Université de Lille

École doctorale MADIS

Centre de Recherche en Informatique, Signal et Automatique de Lille

Thèse

présentée par

Pierrick Pochelu

pour obtenir le titre de

Docteur en informatique

Ensembles de réseaux de neurones répartis et parallèles appliqués au domaine de l'énergie

Ensembles of Deep Neural Networks distributed and parallel for the energy industry

Soutenue le 13 juin 2022 à 15h00 devant le jury composé de :

Présidente du jury	Laurence Duchien	Professeure à l'Université de Lille, France
Rapporteur	Joel Saltz	Professeur à Stony Brook University, USA
Rapporteur	Michel Daydé	Professeur à l'ENSEEIHT, France
Examinatrice	France Boillod-Cerneux	CEA, France
Examinateur	Pierre Sens	Professeur à Sorbonne Université, France
Examinatrice	Cécile Pereira	TotalEnergies S.E., France
Co-encradrant	Bruno Conche	TotalEnergies S.E., France
Directeur	Serge G. Petiton	Professeur à l'Université de Lille, France

Résumé

L'apprentissage ensembliste de réseaux de neurones profonds (Deep Neural Network - DNN) combine les prédictions de plusieurs DNN pour améliorer les performances de résultats obtenus à l'aide d'un réseau unique (DNN) notamment en réduisant l'erreur de généralisation.

Ces méthodes d'ensemble ont le potentiel d'améliorer des applications complexe et stratégique ML en particulier celles du domaine de l'énergie. Cependant, implémenter efficacement ces méthodes d'ensembles soulèvent plusieurs questions: Comment orchestrer efficacement les différentes étapes du cycle de vie des ensembles de DNN (apprentissage, inférence) en vue d'une exécution globale performante tirant parti des infrastructures HPC. Comment entrainer plusieurs DNN indépendants, sélectionner un ensemble performant et exposer les prédictions de l'ensemble construit pour des applications clientes.

Dans ce travail, nous proposons des procédures pour construire un ensemble précis de DNN avec de multiples accélérations des étapes son cycle de vie. Nos procédures garantissent de trouver automatiquement de bons ensembles en se basant sur la diversité des DNN, leur précision individuelle et optimisation du coût de calcul de l'ensemble. En ce qui concerne les deux objectifs qualité des prédictions et la vitesse d'inférence, nous avons découvert que notre procédure construit des ensembles de DNN dont le résultat de prédiction apporte un gain significatif (rapidité, précision des prédictions, reproductibilité, économie en consommation de ressources) supérieur par rapport à l'utilisation d'un DNN unique.

Ces méthodes ont été implémentées au sein de l'entreprise TotalEnergies, sur des cas d'usages concrets et opérationnels (classification d'images, contrôle optimal d'installation par renforcement learning). L'application sur ces cas d'usage, entre dans la stratégie de l'entreprise son orientation vers la production d'énergie décarbonée.

Abstract

Ensemble of Deep Neural Networks (DNNs) combines the predictions from multiple DNNs to improve the predictions over one single DNNs by reducing generalization error. They have the potential to develop the most complex and strategic applications such as developing ML applications for the energy industry. However, ensembles of DNNs raise multiple open questions such as: to run and scale efficiently the steps of their life cycle (training and inference) to obtain global performance leveraging HPC. How to train multiple independent DNNs, select a performant ensemble, and serving the built ensemble's predictions to the client application.

In this work, we propose procedures to build an accurate ensemble of DNNs with multiple accelerations of its life cycle. Our procedures ensure to automatically find good ensembles by searching complementary DNNs, individual accuracy, and computing cost. In regard to those two objectives: accuracy and inference speed, we discovered that our procedure to build Ensembles of DNNs bring high advantages: computing speed, accuracy, and higher reproducibility, save resource consumption.

Those new methods have been implemented on practical applications at TotalEnergies on both supervised image classification and optimal facility control by deep reinforcement learning. Experimental proofs have been performed on multiple energy applications towards a low-carbon future.

Acknowledgement

I would like to thank my esteemed supervisors Professor Serge G. Petiton and Bruno Conche for their supervision and continuous support during the course of my Ph.D. degree. My gratitude extends to the company TotalEnergies for the opportunity to undertake those works.

Besides, I would like to thank the rest of my thesis committee: Professor Dean Laurence Duchien for presiding on the committee, Prof. Joel Saltz, and Prof. Michel Daydé for reviewing my Ph.D. dissertation. Prof. Pierre Sens, Dr. France Boillod-Cerneux, and Dr. Cecile Pereira for participating in the committee.

Additionally, I would like to thank Tanguy Levent for making me discover the reinforcement learning methods and the microgrid energy management challenges. I would like to thank Jerome Gurhem for his help and making discover me *La Maison De La Simulation* laboratory. I also thank Martial Mancip for his feedback to improve the presentation of the Ph.D. defense.

My appreciation also goes to my wife Lisa Darey for her understanding and helping me in my daily life.

Contents

List of Figures

Introduction 1

Recent advances in computer science and machine learning-powered by modern hardware has brought a new level of automation and optimization to various scientific domains and industries.

While rule-based algorithms require tuning a few parameters and a good understanding of input data, machine learning can learn automatically from input data samples and generalize on new data samples by approximating complex functions. Machine learning offers the possibility to automate some interpretative and repetitive tasks using historical data samples or streaming data.

Taking advantage of the big data era, deep neural networks (or DNNs) [86] [78] with billions of parameters can learn from a complex relationships in complex data such as images, texts, and time series. They have produced results comparable to and in some cases surpassing human expert performance [145] and faster time-to-response. Nowadays they have been successfully applied in a wide range of industrial and research projects.

But the path to success is not without pain. Deep learning is an inherent computing-intensive method and requires a high level of parallelism like GPU [78] containing today several thousands of cores and gigabytes of memory to be trained. And more, the construction of a performant model is complex itself: several initial settings must be assessed, which requires multiple training-testing experiments, selecting the most promising one, and deploying it. And more, their training is an indeterministic procedure making them sensitive to random [48]. Finally, because they are data-dependent tasks, some new training data samples require generally a newly trained model, that is why most the machine learning projects are an eternal life cycle, and automation of their construction in parallel is needed for the viability of those methods.

Despite its strength, the high number of parameters may also produce an over-parametrized model which fails to generalize or produce unstable predictions. In those cases, Ensemble of Deep Neural Networks [149] improve significantly predictions quality compared to one single model and have been used lately in several applications. In its simplest form, ensembles consist in interpolating the predictions of multiple independent neural networks. However, even if ensembles produce better predictions, handling their computational complexity and how to build them are unanswered questions. We attempt to answer them in this thesis.

In parallel to the deep learning models field, hardware technology has also quickly evolved in recent years. We are entering the post-Moore era where we no longer enjoy the simple general-purpose CPU performance growth which accelerates all applications. Computes units have reached the maximum attainable frequency due to the power dissipation constraints. To keep developing the applications, we are now entering a new age where the hardware is now more and more specialized with an exponentially increase in the number of cores. The AI accelerators such as GPUs are an excellent example of hardware specialization, which significantly improves the performance per watt for AI workloads. However, efficiently exploiting those new accelerators combined with the fact deep learning models are over-parametrized and unstable requires research and develop new machine learning procedures.

In this thesis, we make the assumption that the HPC infrastructure containing multiple IA accelerators (multi-GPUs) is a promising field to tackle multiple drawbacks linked to the Deep Learning models. Overall, the HPC may efficiently handle automatic construction of neural networks in parallel [110] [133], faster training [177], the ability to handle huge DNNs [68], faster inference [175] [119] [2] and are a promising axis of research for the deep learning workload. Some complete ensemble workflows have been proposed for (non-deep) machine learning [42] [118] [160] [87] [18] but ensemble deep learning and deep reinforcement learning are different context with higher models complexity that have not yet been studied due to their higher computing cost. Some system [49] has been proposed for fast training ensemble deep learning but does not propose a complete ensemble workflow letting the question unanswered: such as how to choose hyperparameters and how to select models in the ensemble? Indeed, the literature is lacking to integrate of all concepts logically and efficiently. This is the ambitious challenge we attempt in this thesis: to bridge the gap between HPC and Deep Learning aiming for practical applications in the energy industry.

This work brings together three contributions organized in three axes. First, we developed new applications for the emerging energy industry, the “clean tech”, by underlying how high power computers may be beneficial to them. Second, we discovered and evaluated novel procedures for automatically training and building Ensembles of DNNs aiming at computing efficiency, qualitative prediction performance, and reproducibility. Third, we propose a novel inference system to build and serve efficiently deep neural network predicting together leveraging multi-GPUs.

Scientific context 2

Computing power, big data applications, machine learning algorithms have quickly evolved in recent years, in such a way that many projects have appeared to create one or multiple machine learning models aiming at the efficient automation of a given task.

In this chapter, we expose the main scientific context of our work: HPC, big data for energy applications, deep learning algorithms, and highlight some problems inherent to automation by deep learning. We then present an overview of the last deep learning methods and their inherent computing cost; in particular, we introduce why HPC is specifically designed to address such issues. This gives a general context to the methods that will be used throughout this manuscript. Figure 6.2 illustrates the scope of this thesis.

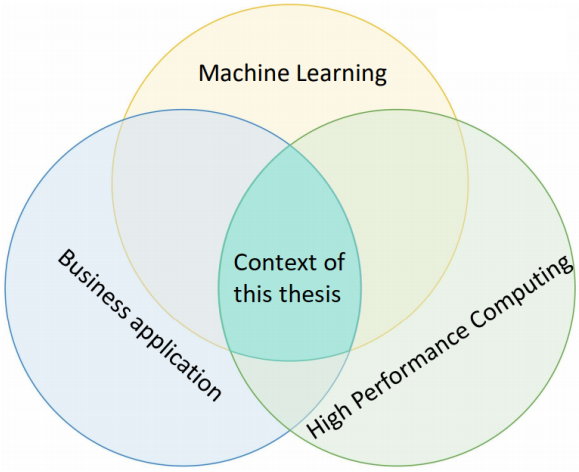

Figure 2.1 – *This thesis is about the intersection of applied Deep Learning last research and the HPC world for diverse energy applications*

2.1 Deep learning promises

Machine Learning has often caught the general attention with some successes such as recognizing images [136], winning against Go world champions [145], and decoding protein structures [75]. This emulsion has already and continues to revolutionize science and industrial processes.

Machine learning refers to algorithms that are trained on previous data and implicitly infer information. It differs from standard programming by the fact the machine learning

model learned itself without a programmer or an expert to explicitly write any rule-based algorithm.

Deep neural network or "Deep Learning", refers to the most complex machine learning models containing a stack of layers and much more trainable parameters than non-deep machine learning algorithms. This stack of layers allows them to learn multiple levels of abstraction from data samples, and therefore they may learn complex relationships between input and expected output. For example in images, they can they may filter the visual noise such as background, and classify the relevant patterns.

The first major advantage of Deep Learning over Machine learning techniques is that machine learning need features extraction to first extract information from complex data (such as images, text, ...) and feed the model with vectors of information. On the opposite, Deep Learning learns itself to extract relevant features and then, returns the associated prediction. For instance, in the image classification task, the convolutional layers extract information vectors from the images and the fully connected layers classify those vector representations.

The second advantage of Deep Learning is the accuracy of its predictions. It can calibrate a large number of parameters on a large amount of data. Deep Learning is a relevant method today taking advantage of multi-cores and the large availability of data samples.

2.2 Deep learning challenges

Deep neural networks (DNNs) are accurate mathematical models but they rise to some challenges linked to their high complexity.

Over-parametrized maths models. More a DNN model is complex and higher the risk of overfitting the training dataset. The overfitting leads to a lack of generalization ability on never seen data samples. In addition to the trainable parameters, deep learning requires also non-trainable parameters named the hyperparameters. The most common hyperparameter include neural network architectures, optimization settings, the data pre-processing.

High experiments need. As a consequence of being highly parameterized models, multiple experiments are run by machine learning scientists with several hypotheses to build a suitable model. Deep learning is known to be sensitive to hyperparameters. Furthermore, because they are nondeterministic methods, multiple experiments may produce different results. The modern computing nodes containing multiple GPUs perfectly fit the need to assess multiple hypotheses at the same time.

One training takes time. The DNN training loop is iterative. It takes thousands of stochastic gradient descent iterations to converge toward an optimum in the objective function with respect to each parameter. Typically, it takes from a few minutes to a few days according to the complexity of the neural network and the task. The machine

learning scientist takes time to perform the experiment and its productivity to test different hypotheses is generally linked to the speed of the DNN convergence.

Inference takes time. After training a DNN we need to predict it. Large neural networks are not only computing-intensive and memory-intensive during the training but during the inference phase too. The machine learning scientist must choose a relevant trade-off between the computing intensity of the training, expected performance during the inference, and the quality of predictions.

Memory consumption. It is admitted that the size of the deep neural networks is correlated to their accuracy. The success of multiple applications is constrained by the required memory such as 3D image recognition [24], text [32], ensemble deep learning [149], embedded machine learning applications... Of course, modern CPUs have generally much more memory than GPUs, but modern CPUs generally fail to get reasonable time performance compared to the GPUs that are designed for the highly parallel operations of the DNNs.

To summarize, deep learning is a computing-intensive method. One single experiment typically takes hours to train a modern deep learning model on one GPU and is limited to the available memory. In practice, multiple experiments are required to calibrate any machine learning model on a given task by multiplying this computing cost. Finally, while the more accurate models may be slower, end-users generally require fast and qualitative predictions.

2.3 Opportunities for Deep Learning using HPC

Historically, High-Performance Computing (HPC) spans a range of application domains including numerical simulation, data processing, and visualization. Deep learning, a new application for HPC, benefit from lessons learned by those other applications: vectorized instructions, GPU Programmation (CUDA [141] and cuBLAS [163]), asynchronicity, cluster management, heterogeneous CPU-GPU computing, and data transfer.

Deep Learning has also a few specific requirements. It needs some specific operations implemented by low-level library cuDNN such as convolutions, activation functions, and max-pooling. Multiple new frameworks and the Python ecosystem make this field unique in terms of applications cluster management and access to external while satisfying cyber-security constraints.

Deep learning operations are modeled as matrix computing to benefit from vectorized and matrix instructions. Recent advances in GPU with thousands of CUDA cores suit the deep learning matrix operation computing needs. CPUs are generally not efficient enough to perform so many calculations simultaneously. We observe the emergence of specific hardware for deep learning applications such tensor cores, specific arithmetic [52] [186] [178], ASICs [1] [17] and AI-specific clusters [115].

[1] https://cloud.google.com/tpu

Neural networks are a function with an input and an output. The deep Learning models field evolves to more and more complex neural architectures to solve more and more challenging tasks. Complex neural architectures prove a better understanding of complex data samples (input) due to large data samples or low signal-to-noise ratio and are also able to prove more and more complex predictions (output) such as a large number of possible classes, multiple tasks ...

More complex is the neural networks and more resources it takes to train, infer, and tune them. In general, deep Learning models chosen by scientists are a trade-off between complexity and accuracy. The figure shows different model architectures 2.2. Lastly, EfficientNet architectures [158] have been proposed showing Pareto dominance compared to previous neural networks on Imagenet. This is a meta-architecture containing hyperparameters to balance between a low number of layers and fast architectures (EfficientNet-B0) and a large number of layers and high accuracy (EfficientNet-B7).

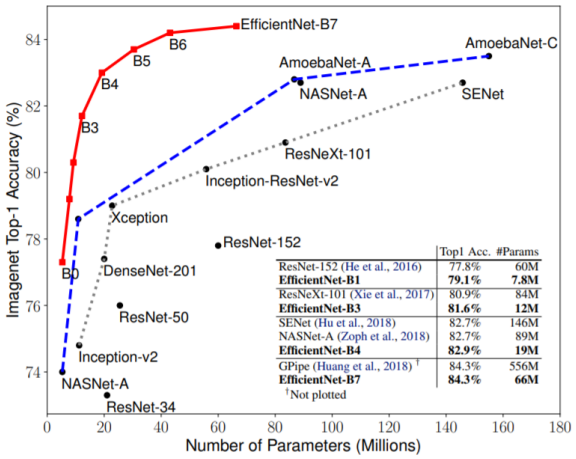

Figure 2.2 – *Correlation between accuracy and number of parameters on ImageNet 2012 [136] for representative state-of-the-art image classification models in recent years. Original image from [158].*

2.4 Presentation of this thesis

The structure of this manuscript is organized as follows. In chapter 3, we introduce the scientific context of this thesis, at the crossroads between energy application, deep learning, and HPC. In particular, we introduce novel deep applications in the energy industry and we conclude on the challenges and how powerful computer power has the potential to alleviate them.

In chapter 4, we present the life cycle of deep learning projects in a very general manner. Then, the last advance in this moving fast field.

In chapter 5, we detail all possibilities that high computing power brings to deep learning applications. We propose a taxonomy of how HPC may be used, enumerate procedures such as hyperparameter optimization, and ensemble, and it allows us to underline the lack of the current methods.

In chapter 6, we propose a performant and flexible HPO and Ensemble procedure to build models. It allows to control of the inference computing power of the produced model and allows to benefit from very high computing power for business applications requiring maximum accuracy. This is our second novel and substantial contribution to the field.

In chapter 7, after having introduced some specifics limitations of Reinforcement Learning we introduce ensembles of reinforcement learning agents. We propose a procedure based on experimental evidence.

In chapter 8, we propose a new inference server to deploy efficiently deep neural networks predicting together on GPU clusters. We experiment with it for any ensemble of deep neural networks averaging their predictions, but it also fits any computational need of neural networks predicting together such as multi-agent, divided-and-conquer.

Finally, in chapter 9, we summarize the contributions brought by this thesis and expose some open perspectives.

Machine Learning and data for energetical applications

3

The field of energy includes many scientific domains in order to gather data, model phenomena, extract knowledge from data, making decisions based on the knowledge.

For instance, meteorology and climatology are interested in measuring, modeling, and predicting phenomena related to the atmosphere: sunlight, wind direction, and wind speed. They are of great interest to match the ambition of Net Zero carbon energy while satisfying TotalEnergies' customers' demand for decarbonized energy. [1]

Geology is focused on studying the outermost solid rock shell of the Earth and geophysics studies its physical phenomenon, two sciences intricate in hydrocarbon discoveries and production. The geology to explore and produce oil and gas is the historical activity of TotalEnergies. Those activities are great to found oil and gas lower and affordable oil and gas. Emergent geology applications include developing Carbon Capture Unit Storage (CCUS) reservoirs allowing to capture of residual emissions.

Biology consists to study living organisms and microbiology. It is a new concern for producing biofuel and biogas, low carbon energy. Microbial CO_2 sequestration is also a potential technology for CCUS and is under investigation at TotalEnergies.

Ecology allows monitoring ecosystems where energy facilities are implemented on earth and seas. Ecologists study the impacts of energy company policy and warn decision-makers to protect ecosystems. TotalEnergies [2] is committed to protect biodiversity where its facilities are located.

Electricity production, storage, and transport concern several scientific domains such as physics, chemistry, and numerical simulation. Facility automation is an active field of work aiming at business productivity under strict safety and environmental constraints. The power grid automatic control systems aim to drive energy production for satisfying the demand every day and 24 hours a day, cost reduction and lowering environment/risk impact. The intermittent sources of renewable energy, volatility of the demand, and low batteries capacity make the design of the power grid system challenging.

[1] https://totalenergies.com/media/news/press-releases/strategy-sustainability-climate-presen

[2] https://totalenergies.com/group/commitment/environmental-issues-challenges/environment-protection/protecting-biodiversity

These scientific domains are of central importance to the present and the future of the energy industry. Data acquisition and data storage evolution allow us to enter the big data era. New machine learning algorithms such as deep neural networks allow automatically extracting knowledge from big data and have multiple applications. They range from providing information to decision-makers (e.g. microscopy object counting with the convolutional neural networks) to full automation of a given proceed (e.g. power grid control with deep reinforcement learning).

In this chapter, we present four novel deep learning applications developed at TotalEnergies.

- Weakly Supervised Object Detection for biology protection (section 3.1). The system was presented at the International Conference on Image, Video and Signal Processing (IVSP 2022) [126].
- Image classification for geology (secion 3.2). was the main subject of three talks [3] [4] [5]
- Object Detection for micro-biology (section 3.3). The whole system including software and patent was deposited.
- Reinforcement learning for electricity control (section 3.4 and chapter 7). was submitted in a scientific conference. It was also the subject of one talk [6].

They are novel contributions to some scientific domains aiming at energy production raising while reducing the environmental impact. The last section summarizes lessons learned from all those projects and allows us to introduce how Machine Learning may benefit from high-performance computing.

3.1 Weakly Supervised Object Detection for biology protection

The energy companies operate all around the world where energy production is possible and may be located where ecosystems need to be protected. Faced with the growing worldwide demand, large-scale industries including Oil and Gas companies are likely to adversely affect the areas left to wildlife. Experts from the Intergovernmental Science-Policy Platform on Biodiversity and Ecosystem Services (IPBES) estimate that 87% of wetlands have been lost since the 18th century as well as 50% of forests since 1990 [15]. Therefore, ecosystem functions and services are now being altered, raising vexing issues about how

[3] "Object Detection with Deep Learning in the Oil and Gas Industry" P Pochelu, J Meng, S Petiton, B Conche at RICE Data Science conference 2019 https://2019datascienceconference.sched.com/event/UZ4a/object-detection-with-deep-learning-in-the-oil-and-gas-industry

[4] "Object Detection with Ensemble Deep Learning" P Pochelu, S Petiton, B Conche at MATHIAS DAYS 2019

[5] "Object Detection with Deep Learning in the Oil and Gas Industry" P Pochelu, S Petiton, B Conche at SIAM HAPPENING VIRTUALLY: Conference on Imaging Science 2020 https://meetings.siam.org/sess/dsp_talk.cfm?p=106760

[6] "Ensemble of Reinforcement Learning for electrical applications" P Pochelu, S Petiton, B Conche at MATHIAS DAYS 2021

to best monitor species and ecosystems over time - and if needed - change the policy to preserve the ecosystem.

3.1.1 Biodiversity monitoring importance and challenges

Through their variability of living organisms, ecosystems yield a flow of vital services ranging from production (food, water, oxygen...) to regulation (soil purification, climate regulation...) through cultural and recreational benefits [43]. Habitat loss and fragmentation, pollution, climate change, species introduction, and disturbance are the current drivers of biodiversity decline. Biodiversity does not only provide services but is also linked to human health. Human health and well-being are influenced by the health of local plant and animal communities and the integrity of the local ecosystems that they form. The COVID-19 pandemic is a reminder that 60.3% [74] of all emerging infectious diseases affecting humans are zoonotic i.e., transferred from animals to humans.

Camera traps have revolutionized the animal research of many species that were previously nearly impossible to observe due to their habitat or behavior. Although not a recent technology, the use of motion-sensor or heat-sensor cameras called "camera traps" has intensified in recent years. The choice of these remotely-activated devices is due to their non-invasive nature, minimal labor costs, high mobility, and battery autonomy. It allows for estimating population trends over time with a continuous, repeatable, and replicable sampling. While being one of the main advantages of the technique, this large amount of data collected also proves to be one of its constraints.

Taking advantage of the big data era, Deep Learning has had a breakthrough impact on many domains. Since its popularity, deep learning was applied to many-colored image datasets often containing several animal classes. Furthermore, deep learning has already shown human-like performance in animal detection. That is why it is a natural choice to help ecologists to leverage the workload.

Faster-RCNN+FPN [99] is specifically suitable to detect small regions in high-definition images but requires high annotation costs including animal localization and animal taxa. By contrast, classification neural networks require only the taxa annotation but they generally fail when images contain small objects to classify and cluttered backgrounds. Based on those previous works, we propose Weakly Supervised Faster-RCNN+FPN to combine those two advantages: image-level annotation cost (Weakly Supervised) and recognition ability similar to the full supervised Faster-RCNN+FPN.

3.1.2 Related works

Over the last 5 years, the landscape of weakly object detection pushed forward the adoption of efficient end-to-end learning frameworks. Some methods extract the saliency map (i.e., object localization probability map) from a CNN classifier trained with an image-level annotation [168]. More advanced methods with the same label requirement consist in combining a CNN classifier and, with a saliency map extracted from it, training another

semantic segmentation CNN [180]. However, they share in common that the majority of works focus on VOC2007 or VOC2012 datasets where objects are rather centered and occupy a large portion of the image.

Another weak supervision method consists in using a standard strongly supervised model but a labelling rule-based function to automatically build labels with unknown accuracy. For example, the JFT-300M dataset was built automatically gathering 375 million images from the web associated with one of the 19 thousand categories depending on web signals. It can be noisy (estimated to 20% error) in terms of label confusion and incorrect labels which are not cleaned by humans. This weak supervision machine learning method is popular when it comes to entity extraction where a label function can be found with some relevant hypothesis.

This label function is domain-specific which is why we propose one to localize animals on camera traps and evaluate their performance. In the 'experimental results' section, we compare the complete workflow compared to some relevant baselines.

Weak supervision is especially useful in video recognizing [172] [146] containing tens of frames per seconds. They often rely on motion cues on images and propagated the objectness information over the neighboring pixels (spatial) and neighboring frames (temporal). Generally, the Weakly Supervised Object Detection in videos assumes that relevant objects are similar from frame to frame which is a not relevant assumption in camera traps data. We are facing a rather different problem: the animal can be present in only one frame or present on multiple frames but with very different positions.

In summary, this work is built on previous research on weak supervision methods and supervised object detection frameworks. The goal is to save the annotation effort on camera traps and to benefit from the state-of-the-art supervised object detection method to spot tiny objects (animals). While most recent initiatives are focusing on how to apply WSOD to localize objects with an end-to-end CNN classifier with some reasonable size objects (VOC2007, VOC2012) or video with multiple similar frames, we focus on building a labeling function applied to small moving objects on a short sequence of images.

3.1.3 The two datasets and goal

Papua New Guinea biodiversity. Images are collected from a monitoring campaign that took place in Papua New Guinea. 8 motion-triggered wildlife camera traps have been used for 15 months with the same settings. They have been operated tied to a tree from 43 to 101 days and took between 474 and 1,272 photographs. When triggered, they shot a burst of three photographs spaced by 1 second, increasing the chances to have at least one good image of the animal. As a majority of mammals are nocturnal, camera traps are also equipped with a flash mode. To decrease potential errors made by our localization algorithm, tuning the settings of cameras is an axis of improvement including the number of shots per sequence and the delay between shots in one sequence.

25 taxa have been captured (dorcopsis luctuosa, bandicoot, scrubfowl, emerald dove...)

into 5400 images with 8 cameras. Each one is fixed to a tree and captures similar background when it is triggered, even if the background may slowly evolve due to leaves falling, weather, flowers blooming ...

Missouri biodiversity. This dataset comes from Missouri campaign [7] [185]. It contains approximately 24,673 camera trap images. There are about 1277 fully labeled images with bounding boxes and classes (squirrel, European hare, red deer, red fox, ...).

Computer vision challenges. Wildlife camera trap automatic recognition present some challenging situation for deep learning methods. First, the animals are small in a cluttered background making it difficult to recognize them. We measure on the Papua New Guinea dataset that the surface of animals can occupy from the entire image to less than 0.2% of pixels of the photo depending on its actual size and its proximity to the camera, with a median of 4%. Additionally, both datasets contain imbalanced classes.

In addition, we summarize the difficulty to recognize an animal in five challenging situations: when the animal is hidden, when the animal is badly framed, when the animal appears blurred (e.g., motion blur when it is running or jumping), hidden behind foliage, unusually small (less than 0.2% of the surface of the image), when the animal is unknown to the training dataset.

3.1.4 The method

Before applying deep learning, the ecologists manually check all images and they input a mapping file that maps the name of the photograph with a taxon number or '0' if there is no animal in the photo. Now, after labeling a few thousand images, the ecologist runs the training phase before running the deep learning inference model on the other images. Because no deep learning model is perfect, to save ecologists' time and reduce errors, all predictions of the systems are sorted in ascending order to their Softmax posterior probability. Therefore, the first images have more chance to contain challenging animals absent from the training dataset. After that, the ecologist goes through the predictions and associated images and corrects the mapping file when needed.

Our model is the fruit of previous works. It uses a meta-architecture Faster-RCNN [131] with Features Pyramid Network [99] and ResNet50 features extractor [58]. It is pre-trained [157] with ImageNet dataset [136] containing 1,000 classes of which 398 are animal classes. Pre-training allows to reduce by about 5 times the convergence speed but it does not make the training converges significantly higher.

One of the main criticism against deep learning is the long labeling time needed for the algorithms to be accurate. The advantage of the proposed classification Deep Learning method is that only one piece of information is needed by an animal in a picture - its taxon - which makes it faster and easier to label training images.

Our weakly supervised Faster-RCNN+FPN method is winning on both sides. First, it can accurately localize and classify a small signal in a cluttered background thanks to the

[7] "http://lila.science/"

FPN module. Second, it requires only classification labeling effort (taxa) and not the usual object detection effort (localization annotations and taxa). The overall proposed method is shown in figure 3.1 and described in the following paragraphs.

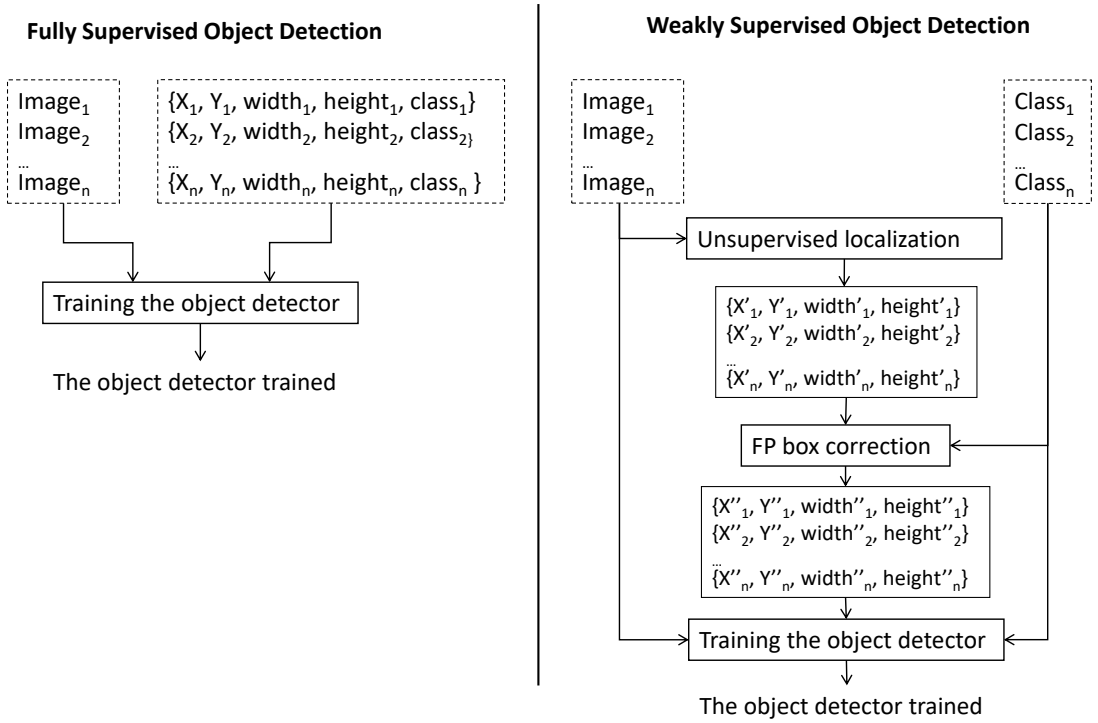

Figure 3.1 – *Comparison between the usual supervised object detection and the proposed Weakly Supervised Object Detection workflow.*

The localization algorithm. The presence of burst mode on modern camera traps makes it possible to capture the same animal in a short sequence of frames in a few seconds when the camera is triggered. Therefore, the animal motion in those frames is possible to distinguish from the background. A motion-based localization allows the ecologists to handle the workload induced by automatically computing bounding boxes of the animals and it enables them to feed the object detection neural network.

Our localization algorithm proceeds following those 6 steps and they are illustrated in figure 3.2:

1. Input a short sequence of images $I_1, I_2, ..., I_n$.
2. It computes a background B computed with median filtering of all the n images of the burst.
3. Motion map $M_1, M_2, ..., M_n$ are euclidean distance between each pixel into $I_1, I_2, ..., I_n$ and each associated pixel in B
4. A binary threshold t is applied on M with t=12%. This new map named T may contain some anomalic values on a few pixels (called “salt and pepper noise”).

5. A morphological opening operation is applied on those previous binary map T. First an erosion operation with kernel 3x3 allows us to erase noisy connected components. Then, a dilation operation with kernel 151x151 ensures all animal parts are connected. It yields D standing for "denoising".

6. The bounding boxes of the largest connected components (if any) is computed from D and returned.

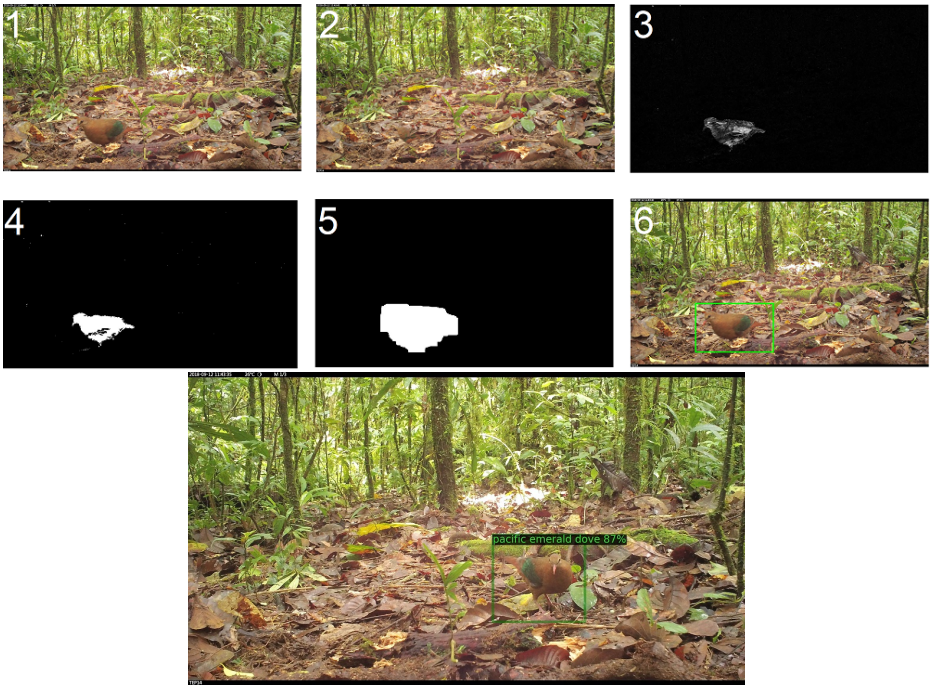

Figure 3.2 – *Our localization algorithm proceeds running step-by-step. The pacific emerald dove is detected despite its camouflage. After training on images like the sixth one, Faster-RCNN+FPN can be accurate on similar images.*

The accuracy of the localization algorithm is 77.6% on the Papua New Guinea dataset test dataset, with 14.7% of false-positive error (motion from raindrops, flying bugs, or wind in the vegetation) and 7.7% of false-negative error (The animal motion is missed).

The FP box correction algorithm. All the FP errors are automatically corrected based on a simple if-then-else code structure. It compares the presence of a detected box and the presence of an animal in the class label to avoid FP errors. Thus, Faster-RCNN+FPN is trained on 92.3% of correct labels.

3.1.5 Experimental results

The accuracy comparison of different workflows is shown in table 3.1. Those workflows, their settings, and their characteristics are discussed in the following text.

CNN Classifier. It is a trained classification deep learning model ResNet50 [58]. It takes as an input the overall photograph and classifies it among 26 classes, 25 animal taxa, or the empty class. Despite we spend time assessing many neural network architectures

and optimizer settings, we have not obtained satisfactory results. It takes only 1 hour to be trained, thus this workflow and RP+Classifier are the fastest to converge.

RP+Classifier. (Region proposals+Classifier) This method uses also a ResNet50 classifier with the same training set as the CNN Classifier workflow. The difference is that it is trained on the region of interests yielded by the localization algorithm presented in section 3.2 and the overall image such as Faster-RCNN and classifier models. An empty class is added because the localization algorithm provides frequent (14.7%) false-positive error motions (again, wind, flying bugs, ...). We explain the performance gained compared to the standard classifier by the fact that the localization algorithm crops most of the cluttered background, thus the classifier part is trained and predicts with a better focus on the moving object.

Auto-Keras. (version 1.0.12) It is an AutoML strategy named Auto-Keras Image Classifier [72]. It is a Bayesian optimization algorithm searching for the best neural network architecture using validation metrics and calibrating the weights of the candidate on the training dataset. We observe a small improvement compared to the ResNet50 classifier after tuning 100 models trained for a maximum of 20 epochs but the training computing cost is about multiplied by 100 (4 days on 1 NVIDIA Tesla V100 GPU). It shows that the famous ResNet50 neural architecture is relevant to our datasets and spending days searching for a specific neural architecture may only improve the results a little bit.

Weakly Supervised Faster-RCNN. We use ResNet50 inside Faster-RCNN and Faster-RCNN+FPN meta-architecture. In the case of the Papua New Guinea dataset, we compare our workflow with and without manually correcting the bounding box annotations. And in the case of the Missouri dataset, we compare our workflow and the original bounding boxes downloaded with the Missouri dataset.

The downloaded dataset annotation of the Missouri dataset ((3) in the figure 3.1) [185] contains inaccurate bounding box annotations. It suffers from about 10% images that contain wrong animal labels or boxes without a visible animal in them. Our proposed workflow can cancel a bounding box when the class of the image is "empty" but it cannot handle when a wrong animal class is given and thus the biggest move (e.g., the wind on the grass) is used as a bounding box associated to a false taxon. We run our Localization based on the motion on this same dataset and provide better annotations ((2) in the figure 3.1).

RP+Classifier works better than classifier thanks to its ability to focus on the region of interest but is still inferior to Faster-RCNN+FPN. RP+Classifier loses contextual information: the relative size of the region of interest is lost because it is resized to the fixed size 256x256 to feed the classifier. And more, the surrounding is also lost it may be a major issue when the localization poorly frames the animal, thus some animal parts are cropped and not given to the neural network.

Finally, we observe on the Papua New Guinea dataset that fully supervised deep learning performs +4.6% on the test compared to the weakly supervised counterpart (again, 7.7% of the training dataset is affected by false-negative error boxes).

Data	Annotation	Method	Presence err. FN	Presence err. FP	Presence OK Taxa error	Presence OK Acc.
PNG	Supervis. classif.	CNN classifier	31.4	6.7	34.3	27.6
	Supervis. classif.	Auto-Keras	28.7	5.1	37.6	28.6
	Weak Obj. Det.(2)	RP+Classifier	7.7	11	10.3	71
	Weak Obj. Det.(2)	F-RCNN	27.7	18.1	12	42.2
	Weak Obj. Det.(2)	**F-RCNN+FPN(ours)**	**2.8**	**1.7**	**13.8**	**81.7**
	Supervised O.D.(1)	F-RCNN+FPN	1.8	1.6	10.3	86.3
MIS	Supervis. classif.	CNN classifier	24.2	12.5	30.9	32.4
	Supervis. classif.	Auto-Keras	20.2	11.7	34.4	33.7
	Weak Obj. Det.(2)	RP+Classifier	19.9	8.5	22.5	49.1
	Weak Obj. Det.(2)	F-RCNN	26.4	10.4	22.6	40.6
	Weak Obj. Det.(2)	**F-RCNN+FPN(ours)**	**15.9**	**4.2**	**20.5**	**59.4**
	Weak Obj. Det.(3)	F-RCNN+FPN	18.1	4.6	27.3	50

Table 3.1 – *Summary of the different deep learning methods tested and their performance on two datasets Missouri (MIS) and Papua New Guinea (PNG). Regarding the method, the localization refers either to class if an animal is present in the picture (in the case of classifiers) or to localizing its bounding boxes (in the case of Faster-RCNN models). (1) our localization algorithm with handcraft corrections, (2) our localization algorithm motion-based (3) localization algorithm by [185]*

3.1.6 Summary and future works

Results obtained in this application show us that Faster-RCNN+FPN suits the ability to recognize small objects in a rather fixed cluttered background. One disadvantage of our workflow is the need for a localization algorithm, even if it shows robust results on two datasets, it is a non-trainable part that cannot properly work if the background contains background motions (such as a city) or if the quality of the overall image change between shots (blur, contrast, ...).

Deep learning has led to progress in image recognition and has been tested extensively to identify animals and yet its application to biodiversity monitoring is still in its infancy. The limitation of the application to a real need comes from the need to have a dataset previously labeled for a given region. The successful application of our weakly-supervised Faster-RCNN with Feature Pyramid Network addresses the need to recognize small objects with a few thousand labeled images. Compared to the fully supervised counterpart we divide by factor 5 the amount of information (the taxon and 4 box coordinates) and the accuracy drops by less than 5%. Now the ecologists can check if the predictions are correct in priority where the model gives a high uncertainty estimate. This allows focusing on the most challenging images or animals absent from the training dataset. It is by this means that in our campaign we discover the Palm Cockatoo taxon visible in figure 3.3 which was absent from our initial training dataset. This method could boost camera traps adoption by tackling the inherent challenges. Additionally, we also hope it will shed light on the benefits of weakly supervised deep learning methods for all disciplinary communities.

Figure 3.3 – ***In absence*** *of a taxon in our training dataset, the classifier will classify them among known classes leading to an avoidable error. The "palm cockatoo" taxon (right) was not photographed during our previous campaigns.*

Bigger datasets. More effort should be provided to increase the volume of the training dataset.

Uncertainty estimates. An important line of research consists in evaluating uncertainty estimates allowing to focus ecologists' attention on the most uncertain predictions (badly framed, blurred, unknown species, ...). Ensemble Deep Learning has already been shown to be not only useful to boost the accuracy but also to produce qualitative uncertainty estimates [82]. However, those methods multiply the computing cost over one single neural network. It seems only experiments and throughout analysis may study the balance between costs and benefits of ensembling.

3.2 Image classification for geology

Lower carbon oil and gas, as well as Carbon Capture Storage (CCS), require a performant geology process. Modern geology companies are used to analyze more and more 2D images. For instance, to date rocks, many microfossils are identified and counted on large microscope images.

Microfossils are extremely useful in age dating, correlation, and paleo-environmental reconstruction to refine our knowledge of geology. Microfossil species are identified and counted on large microscope images and thanks to their frequencies we can compute the date of sedimentary rocks.

To do reliable statistics, a big number of objects need to be identified. That is why we need deep learning to automate this work. Today, thousands of fields of view (microscopy imagery) need to be shot for 1 rock sample. In each field of view, there are hundreds of objects to identify. Among these objects, there are non-fossils (crystals, rock grains, etc...) and others are fossils that we are looking for to study rocks.

3.2.1 Dataset and goal

Microfossils are calcareous objects taken with polarized light microscopy on RGB 2048x2048 images. One image contains hundreds of objects, each object is a region of interest: there are 30% of fossils on average and 70% are other objects (rock fragments, crystals, ...).

Our dataset is a catalog of objects. There are 91 classes (taxa fossils) of 224x224 RGB images (after homemade preprocessing). The classes are imbalanced, we have from 50 images to 2500 images by class, with a total of 32K images in all the datasets. The train/validation/test split is as following: 72% 8% 20%. The F1 score was used and labeled as 'accuracy' on all benchmarks.

3.2.2 Object Detection workflow

To automate these fastidious works, we suggest a hybrid system working with two steps 1) a localization algorithm to localize regions of interest and 2) a supervised Convolutional Neural Network to classify ROIs. This hybrid system presents two advantages compared to the CNN object detection like Faster-RCNN [131]. First, data management of CNN classifiers is more flexible because they take region-of-interest (ROIs) and not on the overall input image, while object detectors require the overall input image. Those ROIs are easier to maintain, extract from geological books, perform data augmentation at ROIs scale and interpret which pixel contributes to the output [132] [103].

The proposed workflow consists in computing large numbers of potential objects using heuristic image analysis. Those potential objects are sent to the CNN classifier previously trained to predict regions. The overall system is shown in figure 3.4.

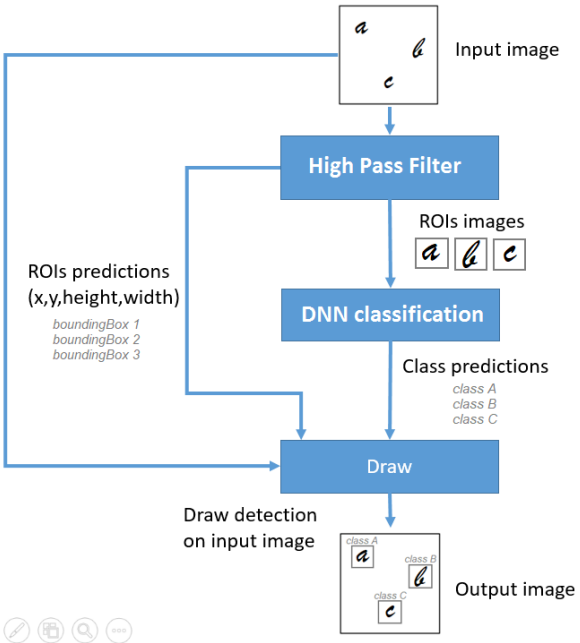

Figure 3.4 – *Micro-fossil detector*

The data management of the classification dataset is lighter and more flexible than a standard object detection dataset allowing geologists to manage easily those data according to their knowledge. All regions of interest (ROIs) are stored (taking 100 pixels squared) and not the entire image photo (taking 2048 pixels squared) this approach is much easier to

	Fscore (%)		Fscore (%) with TTA	
	glorout	imagenet	glorout	imagenet
DenseNet121	89,04	90,27	89.42	91.83
InceptionResNetV2	89,89	89,92	90.72	90.83
InceptionV1	88,89	90,70	89.60	91.16
InceptionV3	88,80	90,11	89.49	90.94
InceptionV3P	85,37	90,12	86.32	91.78
InceptionV4	83,59	90,49	84.32	91.27
Xception	89,99	90,04	90.76	90.94
MobileNet	88,05	89,82	89.31	90.53
resnet18	87,02	N/A	87.55	N/A
resnet50	86,21	89,90	90.23	90.94
VGG16	86,86	N/A	90.23	N/A
VGG19	86,27	89,26	87.42	90.03
Mean Fscore (%)	88.66		89.80	

Table 3.2 – *Fscore of different models with and without test-time-augmentation. They are trained 20 epochs, 15K images spread among 91 classes of micro-fossils.*

manage. And more, they are easier to label: each ROIs are associated with a label without requiring multiple box coordinates and classes [41].

The localization algorithm detects 6% of aggregates among fossils (or 1.8% among all ROIs). Although we cannot detect what aggregates contain, it is not a problem because the goal is to compute quickly a global static on large amounts of fossils. Both localization for polarized and natural light produces negligible amounts of false-negative compared to the classification part.

Fossil/look-alike is between 98,5% and 99% of accuracy. Morpho-group classification test accuracy is 89% with a custom InceptionV3 CNN. This score is 93% on the training datasets, meaning we are facing over-fitting problems.

Fossils are rotation invariant and symmetry invariant, we use this property to reduce the variance of predictions. We exploit Test-Time-Augmentation [143]. One input image from microscopy acquisition contains 8 degrees of freedom (4 rotations x 2 symmetries) given to the input of the models to compute the averaging predictions. The table 3.2 shows the benefit of this method. We experiment it with different state-of-art classification CNN architectures. VGG16 [184], VGG19, ResNet50 [58], ResNet18, InceptionV1 [154], InceptionV3 [155], InceptionV4 [156], InceptionResNetV2, Xception [23], MobileNet [137]. For each architecture we train it twice : one with glorout initialisation [48], second with imagenet pretrained weights [157]. Regarding resnet18 we do not find public imagenet pre-trained weights, and for VGG16 the training with imagenet pre-trained weights diverges.

3.2.3 Deep clustering

Supervised machine learning is very accurate but requires labeled data samples. For example, each time the context change, we require to create a new dataset from scratch which limits the applicability of supervised deep learning. Unsupervised deep learning has the potential to go further by automatically gathering similar images. We expect an unsupervised system to gather fossils images in terms of taxa and not in terms of low-level features (e.g., color, edges, size).

This unsupervised system is trained on data samples without labels. However, to evaluate its ability to make good clusters we use a few labeled data samples in the test dataset. In this work, we measure two scores to measure the quality of cluster predictions. Both of them, give a score between 0 (low cluster quality) and 1 (high-quality cluster):

- **Purity**. It is the percentage of the same class objects that are correctly gathered in the same cluster. This is an intuitive measure, however, this measure may not always be relevant: purity increases as the number of clusters increases. For instance, if each data samples are in a separate cluster, the purity becomes one.
- **NMI (Normalized Mutual Information)** is a less intuitive measure but it corrects the previous purity main disadvantage based on information theory calculation.

We propose a new workflow to evaluate the potential of unsupervised deep learning that we evaluated with both purity and NMI scores. Here are the 3 steps of the workflow:

1. **Pre-training.** A deep neural network is supervised and trained on some microfossils X1. X1 is specific to a certain context (e.g., a certain region in the world)
2. **Data collection.** In a new context, we gather big data samples X2. We label a few fossils images X3, only to evaluate the unsupervised deep learning.
3. **Transfer learning and feature extraction.** We use the previous DNN to extract features on X2 and X3. The next paragraph will allow us to identify after which layer the features should be extracted.
4. **Dimension reduction.** We use a dimension-reduction based on PCA to reduce the size of the features.
5. **Clustering and evaluation of the clustering.** We use a clustering algorithm on X2, we evaluate the cluster computed on X3.

The workflow is illustrated in figure 3.5.

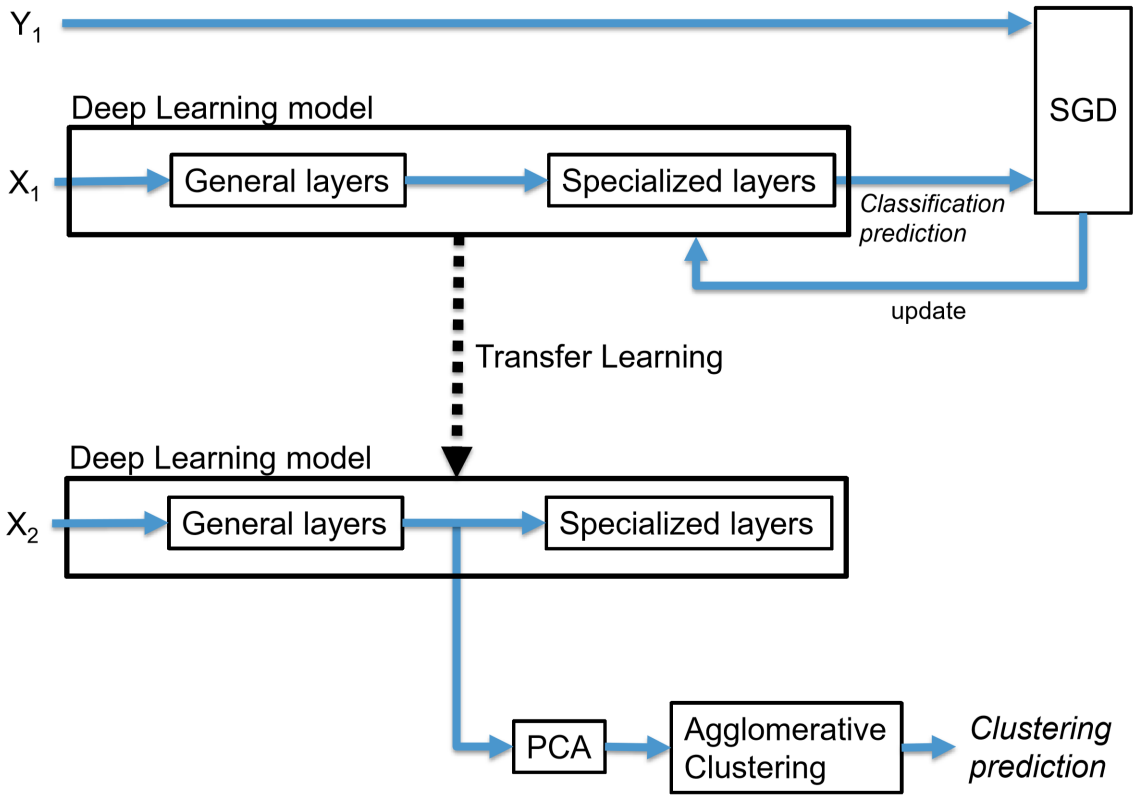

Figure 3.5 – *Image clustering using transfer learning on CNN*

Unlike previous authors [50], we add a dimension-reduction based on PCA between the features extractor and the clustering algorithm. We discovered that PCA has three interests: First, it reduces noise and produces better quality cluster decisions. Second, it accelerates (at least linearly) the clustering execution by reducing its input dimensions. Third, features take less place on the disk that making easier data management.

The number of components of PCA must be carefully configured to denoise data without losing any useful information.

- When n_components is too big, all variance is caught and does not affect the clustering algorithm results.
- When n_components is just fine, a large amount of variance is caught, noise is reduced and the dimension reduction benefits the clustering algorithm.
- When n_components is too small, useful information is lost.

Experimental settings. Our clustering is made by using 515 images distributed among 13 classes with between 20 to 50 images by classes never seen during the pre-training of the neural network. The goal is to see how well our clustering method is able to gather them under the right clusters automatically. In this section, customized inceptionV3 is used and TTA (Test Time Augmentation) is enabled with 4 rotations which increase slightly cluster quality. Agglomerative Clustering [166] (AC) with TTA gets nmi=0.8346 against

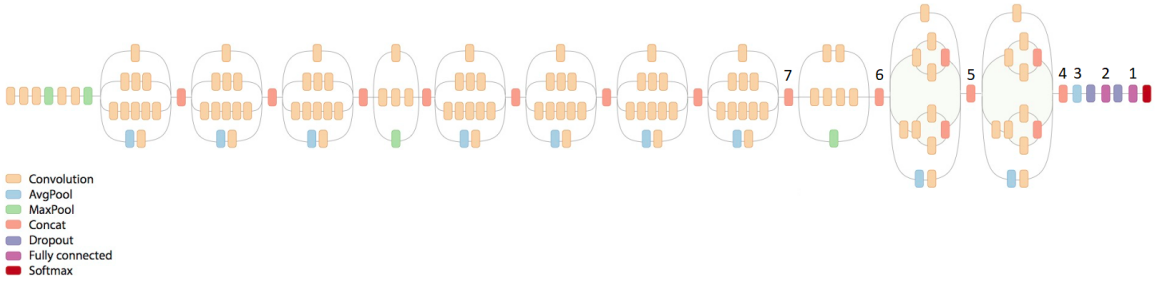

Figure 3.6 – *InceptionV3 neural network customized. We experiment after which layer to extract features: 7 split locations have been assessed identified from 1 to 7.*

Where features are extracted	nb. elements	nmi	purity
After Concatenate (7th split)	110592	0.7330	0.7196
After Concatenate (6th split)	32000	0.8197	0.7625
After Concatenate (5th split)	51200	**0.8346**	**0.8236**
After Concatenate (4th split)	51200	0.7053	0.6871
After Avg Pool (3rd split)	2048	0.7209	0.7058
After Dense (2nd split)	1024	0.7231	0.7037
After Dense (1st split)	91	0.5315	0.4700

Table 3.4 – *Comparison of clustering score after different layers where features are extracted. The split locations are visualized in figure 3.6.*

0.8198 without it. We tried different clustering algorithms and go to the same conclusion: Agglomerative Clustering produces the best clusters.

Clustering tuning. We evaluate three clustering algorithms 3.3 automatically set with the Elbow method to set the number of clusters of AC and K-Means and the epsilon value of DBSCAN. Epsilon controls the maximum Euclidean distance between data points and another one to decide if they belong to the same cluster.

Clutering method	AC [166]	Kmeans	DBSCAN [40]
NMI score	0.8346	0.7496	0.6418

Table 3.3 – *Different clustering algorithms are compared: Agglomerative clustering, K-means and DBSCAN*

We manually tune DBSCAN but at best 33% of the whole dataset is considered noise, and yet we know that all the samples should belong to a cluster with 13 clusters as the optimal number.

Transfer learning. In classification deep learning problems, the last layers are trained to separate data and on the other hand, the last layers are too specialized to be transferable. These two contradictory arguments motivated experiments to find after which layer to extract features to increase the NMI score. The figure 3.6 shows different locations split and the tabular 3.4 shows scores of the split.

PCA tuning. We tried different values to reduce the dimensionality of clustering as shown in tabular 3.5. Initial vectors contain 51,200 elements (5th split). Best results are reached with PCA set to n_components=256. PCA projects data points to 256 elements and

PCA	variance explained	nmi	purity
disabled	100%	0.8346	0.8236
n_components=512	>99.99%	0.8346	0.8236
n_components=256	92.71%	**0.8648**	**0.8496**
n_components=128	82.85%	0.8367	0.8192
n_components=64	72.15%	0.8440	0.8274
n_components=32	61.07%	0.8332	0.8163
n_components=16	49.83%	0.8209	0.8032
n_components=8	37.27%	0.7870	0.7619

Table 3.5 – *Comparison of score with different numbers of components of PCA to denoise data points*

allows to increase NMI by +0.0302 and reduces linearly the computing cost of K-means. We can explain this ease to reduce dimensions because firstly, data points are sparse thanks to the Feature extractor, and secondly because InceptionV3 uses the ReLu activation function that produces multiple 0 values.

Final results. Our clustering system is quite robust. The figure 3.7 illustrates all predicted clusters. Clusters 4, 10, and 12 contain at least 1 image different from the others. The 9 other clusters gather 3 images of the same class. We can observe the robustness of the algorithm: some fossils can be partially destroyed or the image quality can be more or less bad for some optical reasons. For example, the second image of cluster 6 is partially broken.

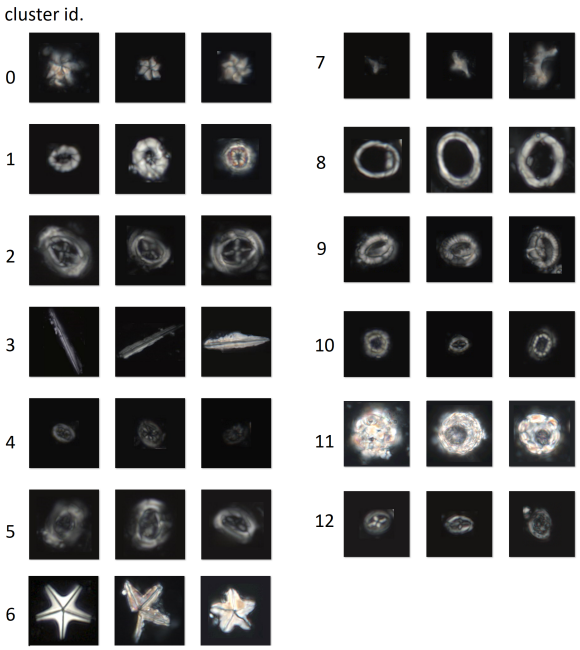

Figure 3.7 – *3 examples image took randomly for each clusters predicted. The global clustering purity is 84.96%*

Similar clustering workflow may be re-used. They require 1 training dataset similar to the target data samples. For example, in microbiology (next section) we may train a

convolutional neural network to classify a few species and apply the clustering workflow to new species.

3.2.4 Summary

We showed image classification can work in this use case too. We can localize accurately 93% fossils from an image. The classifier can classify correctly 88% of fossils but we are optimistic about the outcome of multiple neural network architectures assessed, initialization procedures, and Ensemble techniques.

The possible future works may follow 2 paths to improve results by leveraging a larger amount of computing power:

- **Clustering.** Recent works [8] have shown the benefits of ensemble deep learning to improve the clustering ability in computer vision. Ensemble of deep learning has already shown the potential to tackle the imbalance datasets [22] which is especially beneficial due to the fact most applications have an imbalanced dataset.
- **Semi-supervision.** Semi-supervision consists in using an unlabeled dataset and a few labeled data samples that may be relevant in this case. A key ingredient of the state-of-the-art approach [21] is the use of a big (deep and wide) neural network to pre-train on the unlabeled dataset and tune it on the small number of labeled data samples. Even if ensembles have not been applied yet, due to the fact big deep neural networks are more prone to overfitting, building ensembles should drastically improve predictions [149].

Many supervised applications have those two characteristics of data samples: available unlabeled samples and imbalance between labeled ones. Previous authors [22] [21] shows that big deep learning models (either ensembles or big neural network) are linked to better results in those case. This reinforces our claim that high computing power is important for the applicability of deep learning, not only with academic datasets but in real applications too.

3.3 Object Detection for micro-biology

Carbon Capture Unit Storage (CCUS) aims to capture CO_2 instead of releasing it into the atmosphere. Micro-algae cultivation is a method to perform CCUS. The main challenge of those cultivations comes from the health of the micro-organism to maintain CO_2-consuming cells alive. This is why regular image microscopy and automatic counting based on deep learning have the potential to detect a rapid change in the basin population and quickly take a decision to save the culture. This project has a similar target to the biodiversity one (in section 3.1): monitor living being to analyze how well the population grows. Each microscope image is a 2048x2048 RGB image and may contain from zero to tens of cells, and hundreds of images must be processed to obtain reliable statistics. This application

and the previous one (in section 3.2) are both microscopy projects but aim at a different goals.

Methodology Because the end goal is to reduce CO2 in the atmosphere, we want to create a deep learning model with the same spirit. To do this, we are interested in both objectives: quality of predictions expressed as AP50 [100] to accurately count algae and the energy consumption expressed as kWh.

We use detectron2 [8] framework in this section and ResNet50 as the neural network architecture. Detectron2 proposes default hyperparameters and the number of SGD iterations to apply to a new dataset. We propose a simple procedure to find efficient hyperparameters (Pareto front).

Our dataset contains 200 microscopy images, spread as 180/10/10 as training/validating/testing datasets.

Hyperparameters tuning and computational cost. Note that the Detectron2 framework is evolving fast (like everything in the deep learning field) and default configurations may have been updated. The hyperparameter space is defined as:

1. The batch size {4,8,16,32,64}, default (in the detectron2 config) : 16.
2. The fraction of SGD iterations dedicated to warmup [177]. The warmup can last {1%, 3%, 9%}, default 3%.
3. The warmup factor $10^{-1}, 10^{-2}, 10^{-3}$. Default : 10^{-2}.
4. The L2 regulation rate $10^{-2}, 10^{-3}, 10^{-4}$. Default : 10^{-4}.
5. The learning rate initialization (after warmup) $0.03, 0.01, 0.003$. Default : 0.01.
6. The learning rate is divided three times between the end of the warmup and the termination of the training. The learning rate is divided by a factor after the 50%, 80%, and 95% SGD iterations. The learning rate factor potential values are $4, 8, 16$. Default: 10.

The number of SGD steps is 180K in the default configuration and we propose to bound it to 2500. To get efficient training we randomly evaluate 100 optimizer samples and we select the best one according to the score on the validation set.

It is now well established that the first SGD iterations improve the quality of the model, but this trend slowly declines until plateaus. In some cases, after more training effort the training dynamics may decline to fit the training data samples but fail to generalization on the validation data samples. Indeed, even if training a DNN is a computing-intensive process, a large amount of training effort maybe sometime overfit the data samples too much. This is why a large number of SGD iterations may be a bad usage of computing resources compared to small SGD iterations and multiple assessed hyperparameter optimization.

[8] https://github.com/facebookresearch/detectron2

In the figure 3.8 we illustrate the effect of more SGD iterations on the hyperparameter optimization process. We observe more than 2500 SGD iterations may be not worth more computing effort.

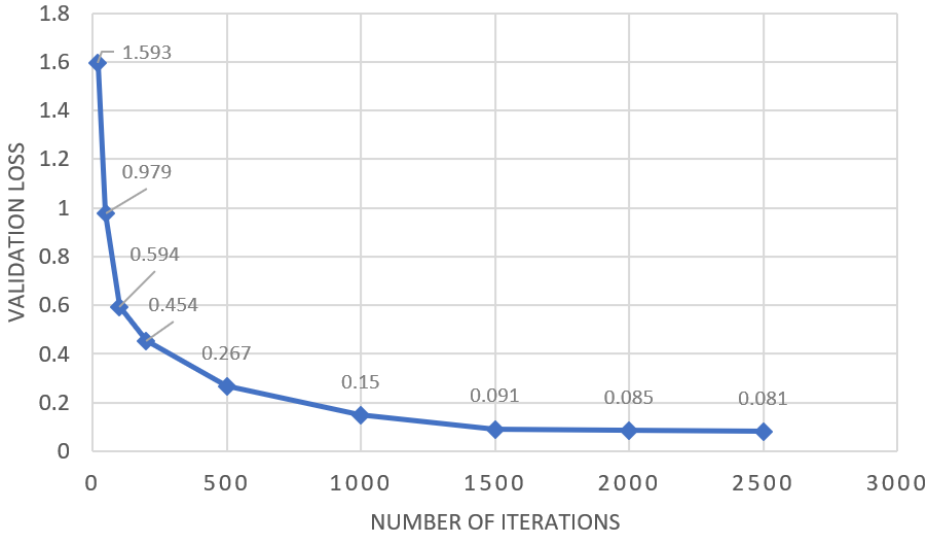

Figure 3.8 – *Effect of the number of training iterations on the best tuned validation accuracy among a population of 100 sampled dnns*

Once hyperparameters have been found we fix them, and evaluate multiple Detectron2 object detection architecures: ResNet50/ResNet101, enabling/disabling FPN[99], enabling/disabling dilated convolutions ("DC5") [179]. It makes a total of 8 different architectures. The pareto-front and the default configuration are shown in table 3.6.

	Training		Inference (2000 images)		
	Time (sec.)	Conso (kWh)	Time (sec.)	Conso (kWh)	AP error (%)
Default R50+FPN	52441	3.979	40.34	0.00306	41.26
HPO R50	551	0.036	26.36	0.00148	28.58
HPO R101	647	0.049	47.34	0.00332	23.11
HPO R101+FPN+DC5	1093	0.088	41.10	0.00355	21.07

Table 3.6 – *We evaluate the default Detectron2 architectures and settings with all Detectron2 architectures and parallel grid search as hyperparameter optimizer. The pareto front between the two objectives AP50 and training kWh is made of those 3 DNNs: R50,R101 and R101+FPN+DC5.*

This table shows that hyperparameter optimization is worth its computing time. If we fix to 2500 SGD iterations and optimize the hyperparameter, the optimized training (e.g., second line) divides by about 100 the computing energy consumed compared to the default settings (first line), and more, it reduces significantly AP50 errors too.

This is the experimental proof that hyperparameter optimization is an essential step of the machine learning life cycle and progress in this field is important for the future of the applications. In other words, it shows how much it is dangerous for performance sake to use the default settings of any machine learning algorithm and software.

Big data. Data samples quantity and quality are of major importance in the machine learning field. This is why we evaluate the importance of gathering data samples, and we attempt to answer if collecting more than 181 images would be beneficial.

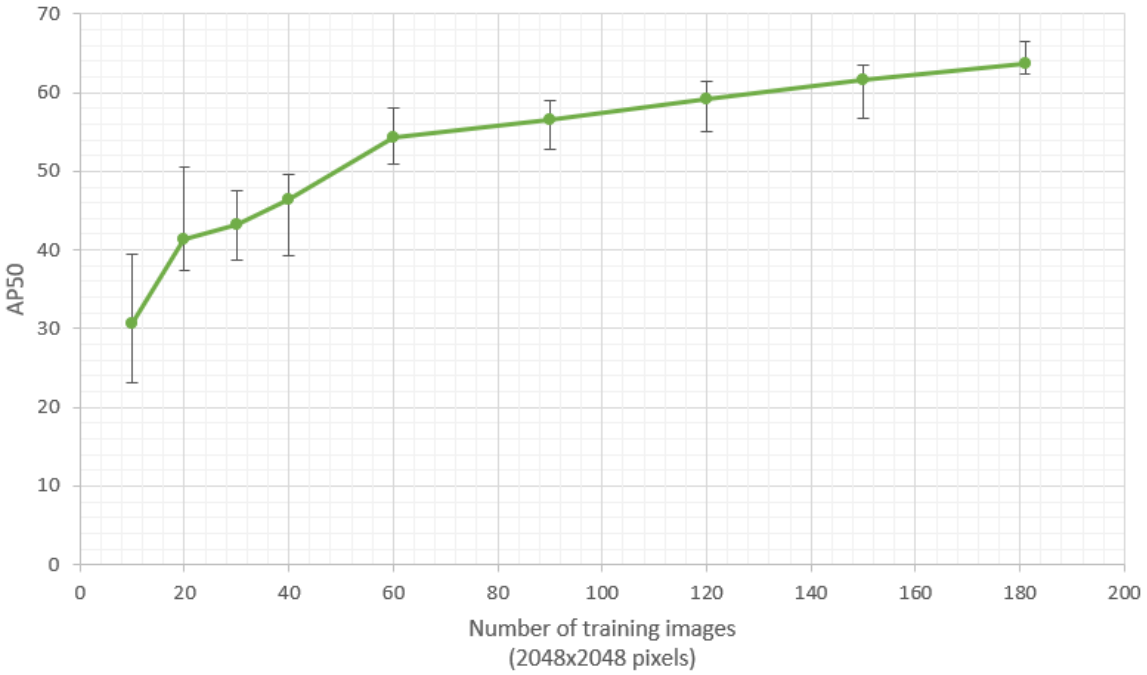

Figure 3.9 – *Effect of the training data size on the test accuracy of ResNet50 DNN. For each assessed amount of training samples we train-test 15 times the DNN we draw the green point represents the mean, the bar represents the extreme values*

We observe that more data improves widely the final accuracy with a slow decline in the trend. And more, we show that more training datasets increase also the stability of the training process (bars are smaller and smaller). However, these experiments may not be ideal because the hyperparameters are tuned for 181 training data samples, if low data samples are available we should tune the optimizer for this volume of data. We will not enter more details regarding data in this thesis to stay focused on deep learning models with the computing angle.

Conclusion. To conclude, the first major observation is that the default settings of any machine learning algorithm and software should be tuned according to its own dataset. Hyperparameter optimization at scale may follow successfully two objectives predictions quality and a computational cost. One must keep in mind that improving the data to feed the machine learning is always beneficial and orthogonal to the tuning of a model. We will not enter into more details in this thesis about the data aspects to be more focused on the model and computational aspects.

3.4 Reinforcement Learning for electricity control

The world lacks safe, low-carbon, and powerful energy alternatives to fossil fuels. Electricity control is a challenging subject due to many aspects: intermittent nature of renewable energy, variations in demand, low storage abilities, [60] [61] significant room for improvement for running electric engines [161]. Deep learning has shown great success in scaling up model-free reinforcement learning algorithms to challenge the Markov Decision Process [109] [145] and are promising methods to solve issues of electricity control [122].

In this ML type, the RL agent learns to control the environment (e.g., the simulator) to optimize a scalar target (the reward). The reinforcement learning loop is formally described in figure 3.10.

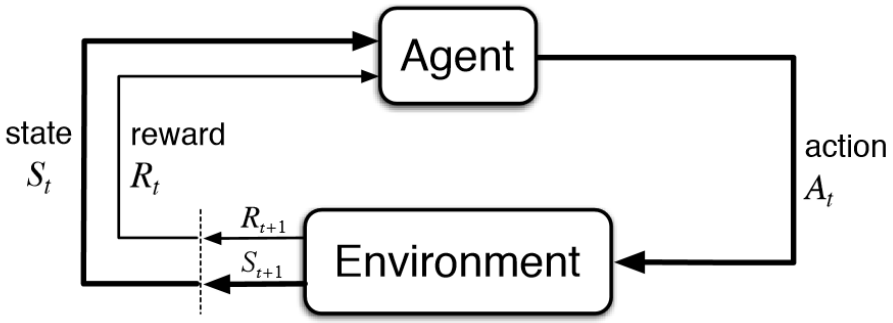

Figure 3.10 – *The reinforcement learning scenario*

A multitude of other RL environments exists and may potentially use our works in this thesis. However, we limited ourselves to the applications where we control electricity in industrial applications with already developed simulators: Pymgrid [60] to simulate microgrids and GEM [161] to simulate the control of an electric engine. Pymgrid is an internal application developed at TotalEnergies becoming open source during this thesis and GEM is an open-source simulator.

3.4.1 Microgrids control

The **Microgrids** are electrical grids that are capable of disconnecting from the main grid (e.g., a national grid). It holds potential in both tackling climate change mitigation via reducing CO_2 emissions and adaptation by increasing infrastructure resiliency. The microgrid simulated in this work is drawn in figure 3.12. The volatility of phenomena such as the sun production, the demand, and the market price combined with the low capacity of the battery makes the control of these systems nontrivial.

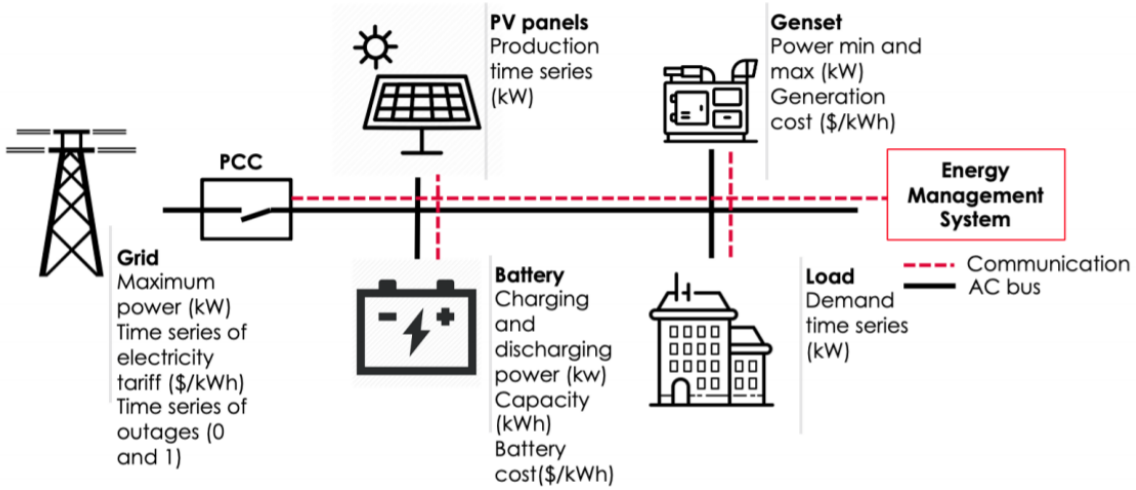

Figure 3.11 – *The Pymgrid digital twin allows to model a microgrid. The Energy Management System algorithm (i.e., RL algorithm) consists in providing electricity to customers by reducing production cost. Original image from [60].*

3.4.2 Electric motor control

The **electric motors** are used in many industrial applications and their efficiency is dependent on how they are controlled. The algorithm (RL) drives the power converter that either directly controls the torque via applied voltage (continuous action space of the RL agent) or by defining the switching on/off the transistors (discrete action space of the RL agent). The goal (reward) is to match the angular velocity of the torque between the motor with the expected torque load by avoiding overheating. The state (not modeled for visibility purposes) is the concatenation of the overall system information and the expected torque.

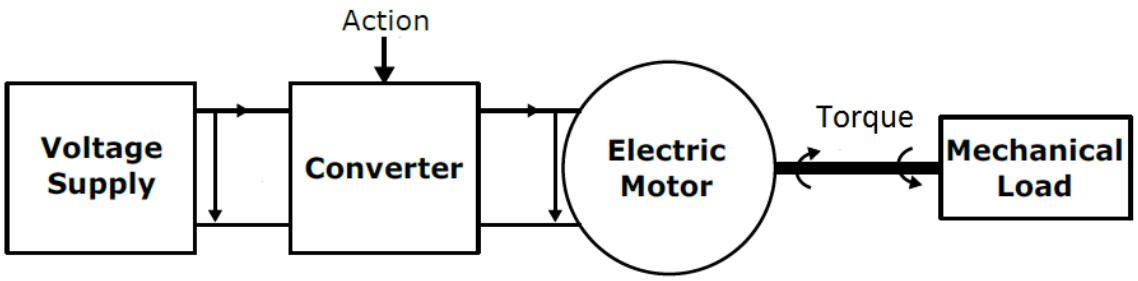

Figure 3.12 – *The GEM simulator allows modeling an electric motor. The goal is to reduce the distance between the torque of the motor and the expected torque. Original image from [161].*

Deep Reinforcement Learning (or just "RL") is gaining popularity for industrial and research applications. However, it still suffers from some key limits slowing down its widespread adoption. First, its performance is sensitive to initial conditions and non-determinism making it unstable and its reproducibility is far from being straightforward [59].

We propose in chapter 7 a new machine learning procedure exploiting distributed

computing, ensemble learning, and hyperparameter optimization to tackle those two strategic challenges.

3.5 Lesson learned from those projects

The development of data acquisition techniques (e.g., camera, captors,...) allows for capturing of accurate real-time data samples, data storage allows storing terabytes of data to feed it to the machine learning algorithms. The development of computing units containing now thousands of core and GBs of memory in the same chipset make possible the training and deployment of deep learning models.

We show in this chapter four projects in the modern and cleaner energy industry that bring out certain challenges.

The increasing number of projects. The increasing number of machine learning projects and their experimental requirement for successful deployment show the need for the automation of the machine learning life cycle. Projects generally have different metrics for success and data preparation requirements (scoping phase). When the goal is specified, it is possible to observe the three steps: tuning, training, and inference.

Data management. Deep learning is a data-centric approach the way we gather data has a significant impact on accuracy. Rigorous data management and managing imbalanced datasets are the daily life of the data scientist. Gathering, storing, loss function design, and over-sampling/under-sampling are not in the scope of this thesis we will rather on the machine learning model life cycle with a fixed dataset and given target metrics.

Multi-goal objectives. despite most of the literature and contests being formulated as some simple mono-objective problems such as a percentage of right predictions, our applications contain generally more objectives.

1. **Technical quality** such as model speed, memory constraint, and power consumption.
2. **Reproducibility** for comparison between algorithm and re-training.
3. **Uncertainty estimates** to automatically trigger a procedure such as asking for human intervention when the algorithm is unsure.

Reproducibility challenge. deep learning is indeterministic and highly parameterized method. They are also data dependent meaning the performant of one model is linked to the training dataset version. And to reproduce any training process, the training must be fully automated, for example, stopping criteria must be fixed in advance before training any model. Therefore the reproducibility of the construction of a model requires gathering all those conditions: the same random seed, the same dataset version, the same training loop code, and no human intervention.

Leveraging hardware if beneficial Like seen previously, deep learning requires generally multiple experiments, and each experiment is computing-intensive. That is why efficient implementation leveraging the underlying cluster of GPUs is needed.

Training and inference are two different projects. The code performing training and inference are generally two different projects aiming at different goals. The training phase search to maximize an objective (e.g. "accuracy"), and some efficient implementation aims to maximize "accuracy-over-time". In the inference phase, it is desirable to keep the accuracy but maximize speed criteria such as the throughput (prediction per second) or the latency (time to get one prediction). Some scientists attempt to blur the line between training and inference by training and predicting a continuous flow of information but online machine learning is out of our scope in the thesis.

More and more initiatives We are facing an explosion in the number of academic and industrial projects in the field of machine learning and especially deep learning. It is now hard to follow those subjects because the state-of-the-art for each problem type is going faster and faster.

1. The platforms are evolving fast in the post-Moore era: development of IA dedicated computing units such as GPU, TPU... Different cloud providers, and on-premise clusters. Computing libraries such as Tensorflow and Torch are one of the most open source projects. Arithmetic libraries such as CudNN, and MKL. Components and instructions set for handling tensor operators.

2. Multiple viable neural networks may solve the same problem. For example, different neural network architectures are regularly proposed and often with multiple versions. For example AlexNet [78] (2012), VGG [184] (2014), Inception [155], ResNet [58] [173] (2016), EfficientNet [158] (2019), EfficientNetV2 [159] (2021)...

3. Generally speaking, the number of scientific works has increased making the literature harder and harder to follow. We assisted recently to an increasing number of scientific conferences in the machine learning field, some ancient scientific conferences have set specialized tracks on machine learning too.

This is why our main contributions are complete workflows made of multiple steps and not a few detailed algorithms. We compare and analyze the effect by varying different steps of the workflows based on experimental evidence but stay in mind it is impossible to be exhaustive. Our workflows are original because they propose novel procedures by aiming both the deep learning prediction quality and a computational cost too by leveraging efficiently multiple GPUs clusters at each step. Furthermore, we assess it on real private applications and compare it all along with the thesis on public testbeds for comparison with previous authors. Finally, we conclude that ensembles of deep neural networks are generally worth the computing cost when they are built with the appropriate workflow in all our projects. Due to the results obtained in this thesis, we encourage future works to follow similar scientific directions: true problems are generally multi-objective problems, and the complete ML life cycle includes model construction, training, and inference.

Now that deep learning applications at TotalEnergies are introduced, the following chapter will describe in more detail the deep learning workload. The next one (chapter

5) will enter more details on the distribution methods across multiple-GPUs computing nodes.

Deep Learning

4

In the mathematical theory of artificial neural networks, the universal approximation theorem [65] states that "a neural network of a single hidden layer containing a finite number of neurons can approximate continuous functions over compact subsets of Rn". In other words, a neural network can learn any continuous mapping from data samples x and their corresponding labels y by fitting its thousands of internal parameters.

The back-propagation [135] was invented and re-invented multiple times in history. It uses the chain rule to be able to derivative multiple stacks of operations, called the layers. The gradient descent updates the parameters in direction of the minimization of a loss (descent of the loss). The loss measures how well the neural network predicts the data x. For example, the formula $(y_p - y)^2$ with y_p the predictions and y the ground-truth labels is a commonly used loss.

The authors discovered that applying backpropagation in multiple small batches randomly drawn from the dataset is much more efficient than training directly on the overall dataset. This phenomenon is explained by the fact that at each iteration we only need to evaluate a partial gradient and not the full gradient which is significantly faster. And more, the stochasticity introduced by a small batch of randomly selected data samples improves the convergence by avoiding being stuck in local minimum and saddle points.

The forward pass and the backward pass are the two operations a neural network performs. In the training phase, forward and backward are called sequentially to compute the loss and then update weights according to the loss. While in the inference phase, only the forward phase is performed.

The application of neural networks for image classification appeared in 1980 [45]. In 1998, LeCun et al. [86] invented the modern Convolutional Neural Network (CNN) and applied them to recognize zip codes on images. He succeeded in both tasks to classify images with CNN classification and object detection by repeating CNN classification without requiring an explicit segmentation into characters. The figure 4.1 the specification of LeNet5, the first modern convolutional neural network.

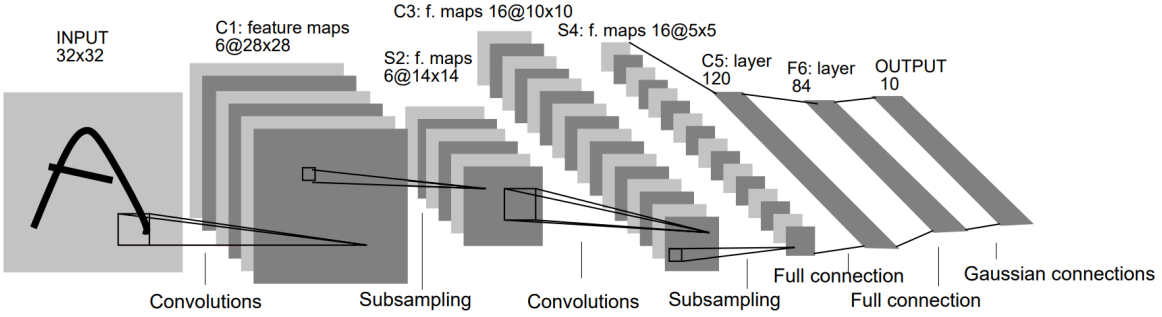

Figure 4.1 – *Architecture of LeNet5, the first modern Convolutional Neural Network. Each gray block are features flowing through deep learning layers. Original image from [86].*

Scientists quickly use the neural network with a large number of layers and more parameters to leverage a large amount of data and improve predictions. Thousands of SGD iterations must be performed to iteratively train efficiently and neural networks making the training loop computing-intensive.

In 2012, the AlexNet [136] architecture implementing a new CNN won the ImageNet challenge mostly because the matrix and vector operations have been parallel on GPUs. It obtained an error rate of 15.3% while the second place was 26.2%. Most operations in deep learning are represented as linear algebra operations and GPUs containing an order of magnitudes more cores than GPUs suit this computation need.

To summarize, Deep Learning is inherited from scalable mathematics where more complexity generally performs better predictions but require also more data samples and more computing requirements. Therefore, this is common knowledge that machine learning applications progress is interleaved with the ability to collect Big Data and the investments in IA-specific clusters.

4.1 Life cycle

The figure 4.2 allows us to put the lexical used in this thesis in perspective.

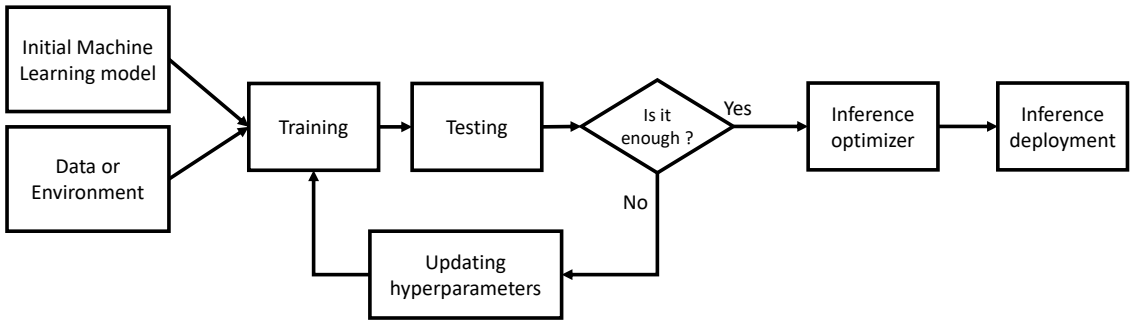

Figure 4.2 – *Standard life cyclic of machine learning model. Data is given for supervised/unsupervised machine learning, the environment is given for reinforcement learning. The updating hyperparameters loop may be performed in parallel.*

Training. Training consists in calibrating the parameters iteratively until the termination.

Evaluation. The most common way to test a machine learning model consists in evaluating its prediction quality on two disjoint samples: validation samples and testing samples. The validation score allows making some decisions on the model such as early stopping or model selection among a library of previously generate a model. For readability purposes, we do not show early stopping or model selection in the figure. And more, some technical performances may be also measured such as forward speed, memory consumption, and energy consumption.

Hyperparameters. We define hyperparameters as non-trainable parameters by the backpropagation which are fixed such as the batch size, the learning rate ... In this thesis, we consider also neural architecture as belonging to hyperparameters such as the number of convolutional blocks, the number of filters per convolution, different activation functions, and so on... Hyperparameters are critical for performance such as generalization performance, error distribution, and technical performance.

Updating hyperparameters. Machine learning training is empiric, due to the fact best hyperparameters are unknown a priori. Therefore, multiple trials are generally required to build an efficient model by sweeping different hyperparameters. The best model on the validation score is selected (i.e., using the accuracy or a Pareto front), then its generalization ability is measured on the test score.

Update of the data, update of the model. The datasets may be periodically updated. When a small update occurs, we may want to train again the neural network to take into account the new dataset version.

Sometimes, all the dataset is entirely replaced. For example, when a new camera investment makes obsolete all previously collected images. In this case, complete hyperparameter optimization may be desired to find a new hyperparameter set that is suitable for the new images. Each time the training data set is updated, the machine learning engineer must ask itself if searching new hyperparameters worth the computing cost, if new training is required, or if the current model suffices. Because the periodic training schedule depends mostly on the application we do not explore them in this thesis.

When a machine learning scientist is facing a new application, the construction of the machine learning model may be more or less automatized. We defined here three scenarios.

1. **Manual tuning**. The machine learning model is taken on the shelf and may benefit from transfer learning to accelerate the process (transfer learning is defined in 4.4). For example, we can take ResNet50 with pre-trained weights on ImageNet. Then, the machine learning scientist trains and tests it for its dataset. If it is not satisfactory, it manually updates it such as changing the learning rate. Multiple experiments may be launched manually in parallel but it may be very time-consuming for the machine learning scientist.

2. **On-the-shelf architecture but the optimizer tuning**. Like the previous scenario, the

machine learning scientist takes an architecture on the shelf and it may benefit from transfer learning. Then, non-architecture hyperparameters are automatically sampled and the experiments are computed independently and asynchronously.

3. **Auto tuning**. All hyperparameters such as neural architectures and optimization settings are sampled and calibrated automatically on the validation dataset. This allows a maximum level of calibration and automation of the process. The machine learning engineer has more time to interpret the produced experiments, tune the evaluation target functions, or may calibrate the hyperparameter space.

Those three scenarios underline that hyperparameters search must be calibrated at any rate. And more, they may be evaluated in parallel automatically to alleviate the scientist to launch manually all those experiments.

4.2 Operations and numerical formats

First, we introduce the three actions we may do with a neural network: forward, backward and parameters update.

Forward. Neural networks are formulated as a stack of layers. The forward phase may be formulated as 4.1 with $a()$, $b()$ and $c()$ the three layers. Each layer takes the data flow as the first argument and their corresponding parameters as the second argument. All operations and data flows are matrices except l which is the scalar loss to minimize.

$$l = c(b(a(x, w_a), w_b)w_c) \tag{4.1}$$

Backward. The partial derivative of all weights with any batch of data x is computed with 4.2.

$$\frac{\partial l}{\partial w_a} = a'(x, w_a) \tag{4.2a}$$

$$\frac{\partial l}{\partial w_b} = b'(a(x, w_a), w_b) * a'(x, w_a) \tag{4.2b}$$

$$\frac{\partial l}{\partial w_c} = c'(b(a(x, w_a), w_b)w_c) * b'(a(x, w_a), w_b) * a'(x; w_a) \tag{4.2c}$$

We may notice that the most advanced neural networks have a more complex topology than a simple stack of layers. Some contains multiple paths [58] [66] and some contains "while loop" operations [131]. However, the main idea stays unchanged on the fact the gradient information is back-propagated from the output to the input.

Parameters update. The parameter update (or "gradient descent") of any layer i is performed with formula 4.3 and the learning rate α.

$$W_i = w_i - \alpha * \frac{\partial l}{\partial w_i} \tag{4.3}$$

More advanced optimizers [76] have shown robust results on a wide range of applications. All the used optimizers rely on the stochastic gradient descent algorithm, that stays one of the most used.

4.3 Numerical formats

The commonly used artificial neural networks are represented as multiple dimensions arrays ("Tensor") of floating-point numbers. They present the advantage to be fast to compute on modern highly parallel devices.

Here are the structures in the neural networks:

- Parameters: they are weights and bias: They are frequently updated during the training phase but constant during the prediction phase.
- Features of the input data samples flowing through the neural network layers in the forward phase from the input to the output.
- Gradients of the parameters. It is the correction of parameters during the learning phase.
- Input data samples and labels: The data to train on or to predict. Labels are used only to train supervised deep neural networks.

All those values are historically stored as float 32 bits. However, this format raises some criticisms on the fact it under-used extreme values from this encoded range and poorly uses computing resources. The precision is especially important during training when the loss is becoming lower and lower and back-propagated through layers but in the forward phase, such precision has been shown less useful. This is why other float format is used or have been invented:

- float32: 1 bit sign, 8 exponent, 23 fraction. It is the most common byte representation for all DNNs data structures.
- float16 (or "half-precision"): 1 bit sign, 5 exponent, 10 fraction. This is a format more and more used for faster computation, saving memory, and consuming less energy. float16 may hurt deep learning accuracy compared to float32.
- uint8 (or "byte" or "char"): integers from 0 to 255. They are often used to quantify image pixel intensity.
- bfloat16: 1 sign, 8 exponent, 7 fraction. bfloat has been introduced by Google. Cloud TPUs are designed to efficiently train a deep neural network with mixed precision between matrix multiplication with bfloat16 and accumulation with float32. More details in section 5.3.4.
- ...

Some criticisms persist with this current formalism. Most operations are performed with values very close to or equal to zero, indicating that the dense operation is maybe not the ideal representation.

However, a new sparse layer [159], well-tuned deep learning, efficient parallel pattern, and low-level code optimization are the lines of research to overcome this challenge. In this thesis, we propose in particular to tune and to ensemble neural networks efficiently.

The brain is also a source of inspiration for the current research:

- **Spiking Neural Network.** (SNNs) are machine learning models which can be applied to some usual machine learning problems. They have the specificity to closely simulate brain functions compared to the usual artificial neural networks (ANNs). However, usual ANNs as such those used in this thesis achieve better accuracy than SNNs with equivalent neural network complexity [152].

- **Neuromorphic hardware.** Neuromorphic devices are physical neural networks to perform neural network computations. Among the most promising SNN dedicated chipsets, Intel developed the Loihi experimental chipset [29]. It is between 0.7 and 5760 times more computing efficient to achieve the SNNs workload compared to a low-voltage Intel 1.67GhZ Atom CPU (2010).

However, due to the lack of systematic improvement, we still use the usual ANNs for this thesis. The Von-Neumann massively parallel chipsets (e.g., GPUs or TPUs) are the most suitable hardware for our ANN workloads. However, we will remain vigilant to the progress in SNNs and Neuromopism toward energy-efficient systems.

4.4 Transfer learning

Transfer learning [157] technics consists in using previous knowledge and using it for a different but related task. Under the transfer learning framework, they are two multiple methods serving multiple purposes: pre-training and fine-tuning.

Pre-training consists of re-using all parameters from a previous task to the targeted task. For example, to classify 224x224 RGB microfossil images, we may use the previously trained ResNet50 neural network on Imagenet instead of random initialization [48]. It is now well established that transfer learning accelerates the convergence speed with a relevant initialization point but it may sometimes decrease the final prediction accuracy compared to the random initialization [38].

Fine-tuning consists of re-using a part of the previous neural network from a previous task and training only a sub-part. For example, we may re-use a previously trained ResNet50 on Imagenet, froze its 49 first layers, and train only the last layer. This allows fast convergence and an accurate model in condition the split how many layers to freeze and how many layers to train is well chosen. A well-chosen split corresponds to well identifying the two parts of the neural network: the general enough layers which are common to two

tasks and should be frozen (such as contour extraction low level), and the layers too specific which are specific to the task (such as the aggregation of high-level features). Because the trainable part may contain a low number of parameters this procedure is known efficient to avoid overfitting and fast to be trained.

4.5 Deep Learning Frameworks

4.5.1 The training frameworks

We see in previous years the emergence of Theano framework followed by a plethora of frameworks like Tensorflow, Torch, Keras, Caffe, CNTK, MXNet... Today some frameworks are still massively used: Tensorflow2, Tensorflow1, Keras, and Torch.

They are 2 API types, either programming with the mathematical operations such as Tensorflow 1 (lower level of abstraction) where 1 line of code is typically one maths operation. Another interface consists in stacking neural network layers (higher level of abstraction) such as Keras where 1 line is directly one neural layer.

Another important distinction is by their execution mode, some or eager others are lazy:

- Eager execution mode. It is the most common code behavior. Each line of code specifies a compute and returns the corresponding result immediately.
- Lazy execution mode. Each line specifies a compute and returns the specification of the building graph. A final instruction run the graph.

The eager mode is simpler to write and debug because the developer can run and assess them one by one. While the lazy mode allows compilation and instruction of the overall graph.

Tensorflow 1 is lazy, Torch is eager, and the Tensorflow 2 developer can switch from eager mode to lazy mode. The eager interface is more intuitive and flexible, but this can be slower performance and deployability. On the simple example of the official documentation, lazy computing may be 75% faster than their eager equivalent. Graph optimization is further discussed for the inference phase in section 4.5.2 and some computational technics may be re-used for the training phase.

Those frameworks are extremely popular with thousands of contributors. Their internal mechanism and programming API are maybe the fast-developing open-source projects with several updates every day, increasing their features, bug fixes, and increasing their performance. This fast evolution of training frameworks mixed with the fast low-level computing libraries evolution and the IA dedicated hardware evolution make the field of the ML platform improving fast.

This evolution in performance makes the training of neural networks easier and easier. This unlocks the possibility of new methodologies such as training multiple

models independently for hyperparameter optimization and ensemble learning that were impossible before.

4.5.2 The inference frameworks

The inference frameworks contain generally two main functions, the "load" function to load a trained DNN from the disk to a targeted device and the prediction function $f(x) \rightarrow y$ with x the data samples and y the associated predictions. The most sophisticated inference frameworks [30], [47], [138], [144] perform post-training optimization such as operations optimization and device-specific optimizations with low or no impact on the accuracy.

A critical part of optimizing the performance of a DNN model is to calibrate its batch size for a target device. It controls the internal cores utilization, the memory consumption, and the data exchange between the CPU containing input data and the device supporting the DNN (if different). That is why some tools [1] scan multiple batch size values and return the fastest one. This Best Batch Strategy (or "BBS") is a relevant mechanism to optimize a single DNN on one device, but it is a naive and rigid method to optimize multiple DNNs predicting together.

4.5.3 The inference system architecture

After training the desired DNN or building the desired ensemble of DNNs we want to infer with it to serve a user application. There are multiple ways to serve this application, either to embed the neural network in the application or delegate it to a server.

The most common metrics are throughput (prediction per second) and latency (time to predict on one data sample). The stability of the latency may be also an important metric in some critical applications such as autonomous vehicles.

Embedded inference. The first way to infer with a neural network consists to embed the neural network in the application. A neural network file may take a few seconds to load into the GPU memory. Therefore, the application must be built by persisting the neural network in the memory to be ready to answer the requests.

This design is simple and applicable in numerous applications. It is also suitable in embedded systems applications in which requesting an external server of DNNs is not always possible. Finally, the absence of middleware between the DNNs and the remaining business application is efficient to improve real-time performance and this is a suitable architecture for real-time application.

Client-server inference. Embedded the DNNs in the business applications is a simple and intuitive design. However, argued architecture choices can separate the DNNs from the rest of the business logic applications.

When a light client machine is connected to a distant powerful machine, it suits the client-server architecture: The light client requests data samples and receives predictions

[1] "https://github.com/triton-inference-server/model_analyzer/blob/main/docs/config_search.md"

from the server. Additionally, a client-server approach can be also desirable for software architecture's sake: It allows centralizing the responsibility, changing, and extending of DNNs into one single application while keeping the code of the business applications focused on the business logic.

The last few years, we show the emergence of software [28] [175], [110], [119] to serve inference framework predictions as a service. Most of them serve wrap prediction in a REST service or sometimes in database management service [3]. They implement often the same features. Ensemble selection allows the client application to choose the model which will answer among multiple applications or the same application but the desired trade-off between accuracy and speed. To improve performance under redundant requests, caching allows avoiding recomputing similar requests. When the amount of requests is low and irregular, adaptative batching allows triggering prediction before the buffered batch is full to improve the latency. We will not enter the details of the engineering of the server to stay focused on the specificity to predict with multiple DNNs together.

To conclude. We stay vigilant to answer those two architectures types when developing an inference system in section 8. This is why we develop it like an intermediate layer between the inference framework (to benefit from computing graph acceleration) and the remainder of the business application (either embedded or client-server). Our proposed inference system is a simple function taking data samples as input and predicting as output by persisting the ensemble of neural networks in the memory. It may be nested in any web server or business logic application.

4.6 Data file format

Deep learning is a data-driven approach requiring a large dataset to be properly trained. In Big Data applications, the entire training set cannot be stored in the memory, so the machine learning algorithm must load it multiple times part by part to be trained. Therefore, machine learning is a reading-intensive method requiring efficient data file format. That is why we are interested to benchmark different dataset formats for energy efficiency sake and to avoid the bottleneck performance of our application. We emphasis also on the simplicity of the API and its robustness to external conditions.

Multiple performances may take into account and no strategy allows for optimizing each one. This is why the best data file format must be evaluated in a given context, in this thesis we focus on HPC applications rather than the cloud. First, the access of data should take into account the acceptability between many small I/Os or a few large I/Os.

In our HPC case, small I/Os may stress the file system for all users. In our HPC environment, the Lustre file system allows parallel access to the content of the files but requires a few best practices such as avoiding storing a large number of files in the same directory and reading a large number of small files.

For the performance's sake when a new application is incoming, we must analyze if the disk is on-premise (multiple 100GBs/sec.) or if a network is bottlenecking and may be

worth compress/decompression time. To make this analysis we compare different reading speeds (including I/O and decompression speed), the compression rate, and the error introduced.

We benchmark and analyze multiple formats in section 5.3.1. We then use the best-suited one for our platform during the remaining of our works.

4.7 Mixed arithmetic

The majority of the deep learning applications use the single-precision floating-point number IEEE 754 (or FP32). Lower precision improves the computing speed but it reduces the arithmetic stability and reduces the deep learning ability to recognize objects. Due to the complexity of the underlying phenomenon and the potential gain in the training and inference phase we dedicate a section 5.3.4.

4.8 Data parallelism

Data parallelism consists in duplicating the neural network on multiple GPUs and dividing the global data batch into local batches. Each GPU takes charge of a local batch resulting in the parallel computing of batches. We may divide the data parallelism into two distinct algorithms according to if we accelerate the inference phase or the training phase.

Data parallelism makes sense only when the neural network handles multiple data samples. For example, it does not suit deep reinforcement learning applications in inference mode where data samples (i.e., "states") and predictions (i.e., "actions") are exchanged with the simulator.

4.8.1 Data parallelism in inference phase.

The inference phase is a long-running process delivering the predictions to the associated business client application whenever it is requested. Therefore, optimizing the inference speed is sometimes considered more strategic than training [33].

Data parallel prediction consists in splitting data, predicting with the neural network in parallel, and combining predictions. The independent nature of predictions makes the task trivial with existing frameworks such as the MapReduce framework [31] and the emergent inference server technologies such as RayServe [110] and Triton [175]. However, those technologies may fail to handle efficiently ensembles of neural networks, this is why we dedicated the chapter 8 with novel and substantial contributions.

4.8.2 Data parallelism in the training phase.

Accelerating the training speed is strategic because it is a computing-intensive step of the machine life cycle that must be repeated each time the dataset is updated. Data parallel

training consists in attempting to improve speed by distributing computation on multiple GPUs at the cost of communication time.

Each worker is associated with a training device (generally a GPU), and iterates on those steps :

1. Loads the local batch of data samples
2. Compute the local gradients (forward and backward)
3. The global gradients are gathered and averaged across all workers. This is a synchronization step.
4. Parameters are updated using the global gradients.

Ideal optimizer settings in data-parallel may not be ideal in sequential training. This claim makes the comparison challenging to formulate if data parallelism really accelerates the training convergence speed.

The fact the question is strategic and complex is the reason we dedicated a chapter 5.3.3. We attempt to evaluate the behaviors under the concept of training convergence, perform general conclusions, and propose directions to follow toward efficient deep learning model construction and deployment.

4.9 Ensemble Machine Learning

Ensembles of deep neural networks may improve significantly generalization ability compared to one single neural network [149].

Ensemble Machine Learning, in its simplest form, consists of 1) training N models independently with different random seeds 2) averaging those N models' predictions to create a better final prediction.

In figure 4.2, we do not make appear ensemble of machine learning models in the standard machine life cycle. Even if machine learning scientists have well understood the benefits of ensembles, the procedure to build them automatically is unknown. And more, predicting efficiently with ensembles is out of the scope of the current inference frameworks and inference servers. This thesis underlines those missing pieces of the pipeline and proposes new procedures for both applications types: supervised deep learning and deep reinforcement learning.

4.9.1 Theoretical explanation

Thanks to a huge number of parameters, DNNs have a lot of freedom to denoise signals, identify patterns and classify them. However, this great advantage becomes a drawback when DNNs fit too much training dataset and cannot generalize on the test dataset. Overfitting refers to the behavior of a model to take too much attention to non-informative noise.

Let's try to discover how an ensemble of neural networks improves the final predictions. We first consider that neural networks have unbiased but noisy predictions of standard deviation σ with 0 as of right answers. We can note the prediction of a neural network following $X \sim \mathcal{N}(0, \sigma^2)$. When n independent neural networks predictions are interpolated the ensemble error prediction becomes $\sigma / \sqrt{(n)}$. The square root function shows that increasing the number of models n is always beneficial and allows to converge predictions towards 0 but the benefits of ensembling are declining according to the square root function.

Indeed, the above formalism hides a lot of complexity. First, each neural network contains an unknown amount of bias error and variance error and each error follows an unknown distribution. There are multiple procedures to build ensembles. Any good procedure should ensure that the ensemble predictions outperform the best neural network in it. Finally, any good procedure should ensure the final accuracy is obtained with a reasonable amount of computing power.

Authors [149] first promotes the same algorithms at different runtime to reduce variance by averaging their predictions. Then, authors [97] [11] [42] promote more and more diversity, not only based on the random effects, but with different Machine Learning algorithms in the same Ensemble to maximize diversity and so promote diverse biases in the same ensemble.

All those previous works have been applied to non-deep learning. The optimal procedure for (non-deep) machine learning algorithms may not be ideal in the deep learning case. This is what we discovered in section 4.9.3.

4.9.2 Challenges

The majority of scientists build ensembles manually by gathering neural networks from a library of models. It allows them to use already implemented neural networks and pre-trained weights. These simple technics have multiple limitations to be applied to a new task. We will attempt to enumerate them here.

- **Large amount of manual experiments**. The number of possible ensembles from a library of models is defined by $n^2 - 1$. For each neural network, we can imagine a bit to decide if we select it, minus the empty ensemble. For example, a library of 10 state-of-art models requires a workload of 1023 ensembles to evaluate.

- **Absence of available library**. First, if the deep learning task is original no library will be available or the on-the-shelf architectures poorly adapted. Of course, hyper-parameter optimization may generate a library of models, but HPO works have been thought to tune the best neural network possible and not the best library to then ensemble them.

- **How combine them ?**. Multiple procedures have been proposed. Multiple questions may be answered. How to ensure some base neural networks are complementary? Is it necessary to give more importance to the best base model in the ensemble

according to the validation score? Do ensemble methods improve accuracy compared to only select the best available neural network in it?

- **High computing cost**. A Deep Neural Network is already a computing-intensive maths model. Ensemble of deep neural networks raises the question of whether the accuracy is worth the increased computing cost. An ideal procedure should allow selecting the trade-off according to the user's preference.
- **Software to handle the parallelism**. Some inference server [175] [119] [2] allows managing multiple DNNs in parallel mostly for multi-user purpose or multi-applications. However, they are not adapted to compute heterogeneous deep neural networks predicting together.

4.9.3 Experimental analysis

Multiple ensemble procedures have been proposed to predict with multiple machine learning algorithms. Our goal is to experiment fairly the most of the Ensemble Deep Learning methods available.

Experimental conditions. The base neural network, optimizer, and database used for experiments are the ones that are proposed in the tutorial of the framework Tensorflow and are drawn in figure 4.3. It is a simple and efficient architecture allowing fast experiments. Models have trained 10 epochs on the CIFAR10 dataset and they perform an accuracy of 70.70 ± 1.65.

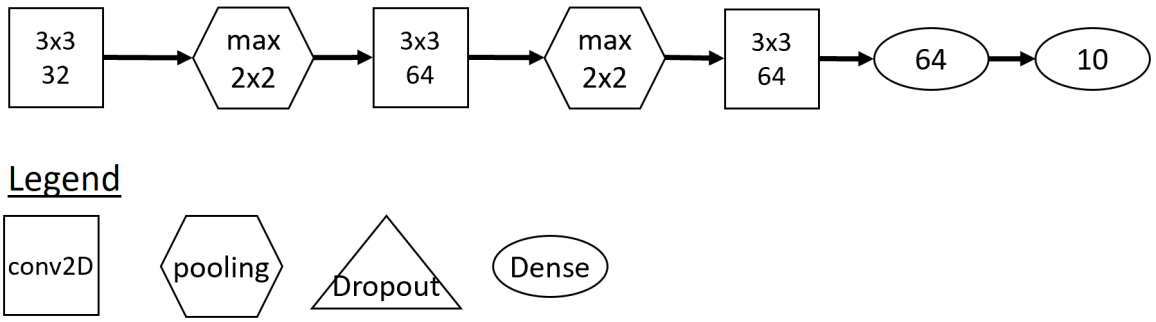

Figure 4.3 – *Simple and efficient convolutional neural network architectures given by the Tensorflow documentation. We use this tiny deep neural network allows to evaluate quickly ensemble methods without advanced parallelism across multiple GPUs.*

All experiments have been done with an ensemble of 4 models. All methods require a combination rule, that is to say, a method taking the base predictions of the base neural networks and computing the final. This algorithm may be the simple average, the majority voting... Some more complex combination rule such as weighted averaging and stacking consists of more complex algorithms which require calibration data, in this case, this is called "meta-learner". The ensemble with a meta-learner can be trained with the 1-split mode or 2-splits mode. In 1-split mode, models and meta-models are trained on the training dataset containing 5000 images by class. In 2-splits mode, models are trained

	Major. vot.	Avg.	W. avg.	Stacking	Gating
1-split mode	74.97±0.49	76.22±0.50	76.83±0.45	76.87±0.25	76.28±0.47
2-split mode	N/A	N/A	76.61±0.20	76.40±0.16	76.51±0.50

Table 4.1 – *Different combiner rules. N/A means the combination rule cannot be calibrated on the validation samples.*

	Bagging	Bag+Stack	AdaBoost	NCL
1-split mode	76.63±0.42	76.92±0.46	76.94±0.40	76.50±0.92
2-split mode	N/A	76.33±0.34	N/A	N/A

Table 4.2 – *Different training procedure in Ensemble*

	Int. Stacking	Int.Stacking2	FS	Snapshot L.
1-split mode	72.64±0.65	76.08±0.34	75.81±0.50	73.36±0.26
2-split mode	N/A	74.14±0.31	N/A	N/A

Table 4.3 – *Accuracy of some deep learning of individual model in Ensemble.*

on 4500 images by class and meta-learner on the 500 remaining images by class. Some algorithms do not have a meta-learner so the 2-splits mode is not applicable.

To test the effect of neural architecture diversity we build and trained 4 models with different architectures until reach $\approx$ 70% accuracy. To classify new data we use the Averaging strategy.

Results with 4 base neural networks. The table 4.3 shows all method implemented and tested. We implement and assess: averaging, majority voting, weighted averaging [55], bagging[9], stacking [170], AdaBoost [140], gating [7], snapshot learning [67], Negative Correlation Learning (NCL) [102], Features pyramid ensemble inspired by [99] (each DNN is independently trained on different resolution of images: 24x24, 32x32, 48x48 and, 64x64)+averaging, bagging+stacking [10].

We may attempt to class them among diverse categories but a complete taxonomy is difficult due to the multiple strategies proposed: implicit diversity (averaging, bagging), explicit diversity (NCL, features pyramid ensemble), priority to the best (weighted averaging, stacking, gating).

Implicit diversity does not guarantee enough diversity and the best one has the same weight as the poorest. Explicit diversity does not guarantee to be the 'right' form of diversity and risks damaging individual accuracy. In the elitist approaches, the data samples used to evaluate which base model is the best may not well generalize and may produce low diversity.

Scale the number of base neural networks. Figure 4.4 shows the effect of increasing the number of models on the accuracy. The meta-learner of the stacker is set to be linear regression, its performance declines when the number of input decline due to overfitting. It shows that averaging is a simpler and more robust method than stacking.

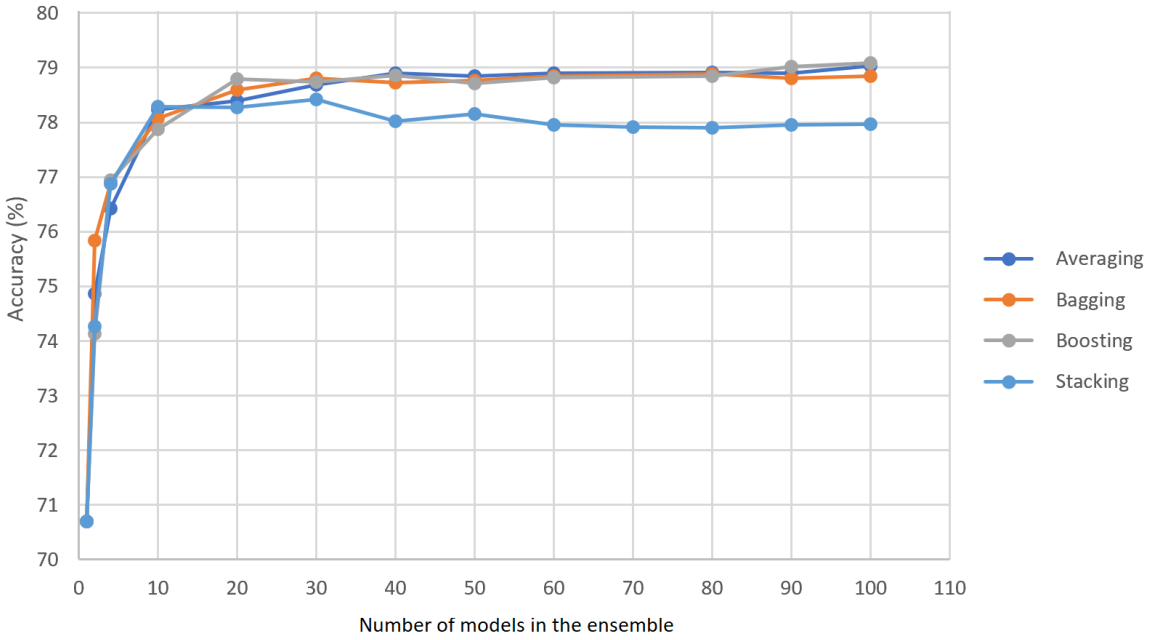

Figure 4.4 – *Ensemble accuracy by varying the ensemble size from 1 to 100.*

4.9.4 Summary on ensembles

To conclude, multiple advanced methods have been proposed for decision trees but our first experiments show that simply averaging predictions deep neural network is simpler, accurate, and robust. First it success to reduce over-fitting, without the need to calibrate a meta-learn on some validation samples. However, there are still a lot of challenges to overcome: such as how to select the model in the ensemble? How ensure that the prediction gain is worth the computing cost? What about reinforcement learning? How to deploy them in inference? We will attempt to all those questions later.

4.10 AutoML

The empirical nature of research in Deep Learning leads scientists to try manually multiple model architecture settings, optimization settings, and domain-specific settings to find the best-suited one for a new task. AutoML or ("Automated Machine Learning") attempts to automatically perform the tuning hyperparameters, training parameters, and testing loop of machine learning illustrated in section 4.1.

The performance of those models is optimal when both parameters and hyperparameters are optimized. They are especially difficult to calibrate in the context of deep learning where the architecture contains millions of parameters to optimize toward qualitative prediction, and tens of hyperparameters aiming at both qualitative and a suitable technical performance.

4.10.1 Definition

AutoML is generally made of at least 3 modules that will be presented in the next sections: the hyperparameter space ("HPS" for short) defining all potential DNNs, the hyperparameter optimization algorithm ("HPO" for short) defining the DNNs sampling strategy, the evaluation phase consisting to compare DNNs.

Some additional modules can be proposed to improve the automation of DNNs construction. For example, in section 6 we propose the idea to store on the disk all assessed models during the hyperparameter search, and we propose a fourth module to build ensembles of deep neural networks.

More formally, any model previously trained of hyperparameter λ sampled from hyperparameter space Λ is written M_λ. The hyperparameter optimization goal defined in equation 4.4 consists in finding the best hyperparamer $\lambda* \in \Lambda$ building a model to reduce the error measured by E. The error is measured on the validation data x_{valid} matching labels y_{valid}.

$$\lambda* = \arg\min_{\lambda \in \Lambda} E(M_\lambda(x_{valid}), y_{valid}) \tag{4.4}$$

In the literature, AutoML algorithms are typically compared based on their results in the evaluation phase. While this may seem intuitive, this field is now facing multiple methodological questions [90] on the relevance of comparing multiple workflows made of different stages and different initial conditions, using different hardware. And more, to fairly compare their robustness, multiple datasets, and multiple random seeds must be reported which is a computing-intensive research area limiting initiatives.

4.10.2 Optimization methods

No Free Lunch theorem [169] proves that no black-box hyperparameter optimization (HPO) can show superior performance to random search in all cases. Nevertheless, methods searching between global exploration and local exploitation have shown a stable performance in diverse applications.

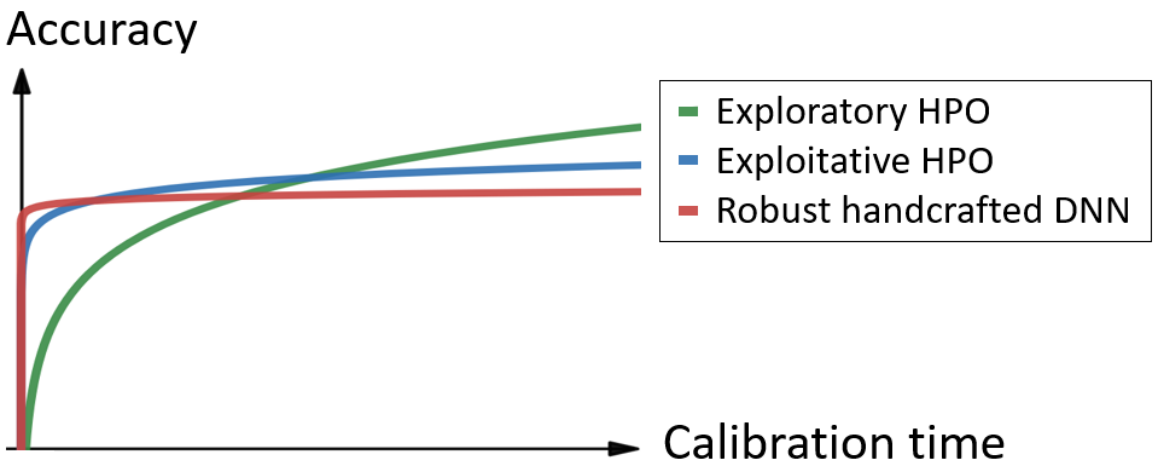

Figure 4.5 – *Sketch of the three families of hyperparameter optimizers: rather exploratory optimizer, rather exploitative optimizer, or use on-the-shelf hyperparameters*

Even if most optimizers balance between exploration and exploitation, some methods are "more exploratory" and others "more exploitative". The more explorative approaches consist in using the previous values to compute the next one, they may converge fast towards an optimum but may be stuck there forever. Due to their sequential nature, exploitative methods generally fail to exploit efficiently multiple GPUs.

On the opposite, explorative methods consist of sweeping hyperparameter space with low attention to the previous values. It may take more time to get reasonable accuracy compared to the exploitative, but in the long term or with a large amount of computing power, they generally outperform because they leverage naturally from multiple computing units and do not fail in a local optimum.

Last but not least, explorative approaches are more robust to the initial condition, while the initial hyperparameter affects critically exploitative approaches. The robustness of the hyperparameter optimizer is of major importance in AutoML because the end goal is to automate the fastidious work to run the life cycle of the model with the low intervention of a human.

A more general approach consists in searching not only neural network architectures but optimization settings and data processing with fixed-length vector hyperparameters. Bayesian optimization, like Sequential Model-Based Optimization (SMBO) with Gaussian model is known to perform well to optimize continuous hyperparameters [147] [63]. Tree-based models are more adapted to the discrete hyperparameters like Tree Parzen Estimator (TPE) [5] and Sequential Model-based Algorithm Configuration (SMAC) [69]. Despite SMBO being inherently sequential methods, parallel versions have been proposed [147] based on successive populations to explore the hyperparameter space by leveraging multi-cores.

Evolutionary methods [129] [176] are a naturally parallel approach running successive generations of trials in parallel. By running them on a large scale on GPU clusters for several days, the authors discovered they are capable to converge very late and very high.

They seem the state-of-the-art to reach the maximum accuracy on complex realistic image datasets like CIFAR100.

Additionally to the exploration/exploitation dilemma some technics have been proposed to boost optimization performance. Early stopping [127] [93] has been also proposed to stop the less promising trials to focus hardware on the most promising trials accelerating the overall convergence process. However, when training dynamics are noisy (such as Reinforcement Learning) we cannot guarantee that an experiment performing badly in short term will be performing badly in the longer term.

Weights sharing [88] consists to perform previously trained parameters as initialization of the previous experiment as initialization of the next experiments. It allows to save computing costs and converge faster. However, sharing weights requires common neural network architectures between experiment which make these technics hard to apply if the neural architecture must be explored. Weights sharing leads also make the reproducibility of experiments difficult.

4.10.3 Hyperparameter space, neural architecture search

The literature proposes two hyperparameters space approaches. Global hyperparameter optimization searching all hyperparameters (optimization settings, neural architecture, domain-specific settings, ...). The second approach is neural architecture search (or "NAS") where only the neural network is optimized and the other settings are fixed.

Global hyperparameters optimization is often made with a fixed-vector length that associates each hyperparameter to its corresponding value (e.g., for example, see 6.1). It is a convenient approach allowing to explore all aspects of the machine learning model and its training but it may be limitations in terms of neural network architecture explored compared to NAS.

In NAS approaches [123] [72] [101] neural networks are build by taking a sequence of decisions to build a specialized convolutional block. This convolutional bock is then repeated a fixed number of times to generate the entire neural architecture.

NAS is more suited to create a specialized neural network with a higher degree of freedom to generate the DNN architecture by taking "micro-decisions" to tune the convolutional block, for example, the number of convolutions per convolutional block, their activation function... However, it cannot take the "macro-decisions" on the overall architecture such as the number of convolutional blocks, the learning rate, and the batch size.

We use a more global hyperparameter optimization with fixed-vector encoding to search hyperparameters, this is a simpler and a more general approach. We exploit this approach in the supervised context 6 and the reinforcement learning one 7.

4.10.4 AutoML with ensembling

AutoML+ensemble proposed workflow is proposed in table 4.4. Due to the strong impact on the business, those frameworks are going fast and are sometimes hidden behind commercial services.

Name	Backend	Type	Optimization	Ens. sel. strategy	Ensemble combiner
Auto WEKA [160]	WEKA	ML	SMAC [69]	Ad hoc 5 models	Voting or Stacking
Auto Sklearn [42]	sklearn	ML	SMAC with Dask	Forward greedy # given	Averaging
TPOT [118]	sklearn	ML	Genetic with Dask	Ad hoc	Stacking
Autostacker [18]	sklearn XGBoost	ML	Evolutionary	Ad hoc	H. Stacking
H2O AutoML [87]	H2O	ML	RS [6] with Spark	the best of each algo. 1 model per family	Stacking
H2O Driverless AI [2]	H2O	DL	??? with Spark	Ad hoc # given	Averaging
Ours (section 6)	**Keras**	**DL**	**Hyperband [91]** **with Ray**	**SMOBF[124]**	**Averaging** **in parallel** **section 8**
Ours (section 7)	**RLLIB**	**DRL**	**RS** **with Ray**	**Ad hoc** **# given**	**Averaging** **in parallel** **section 8**

Table 4.4 – *Comparison of AutoML workflows.*

Some columns are detailed below:

1. **Type**: "ML" stands for (non-deep) machine learning, DL such as image recognition and DRL for Deep Reinforcement Learning.
2. **Optimization**: The optimization algorithm used. The mention "with Spark", "with Ray" or "with Dask" indicates the hyperparameter optimization is distributed with one of those frameworks.
3. **Ens. sel. strategy**: stands for "Ensemble Selection strategy" defining how ensembles are built. The first row indicates how the population is aggregated and the second how the number (i.e., Fixed or calibrated on the validation set ?).
4. The mention "???" means the information is not present in the documentation. H2O Driverless IA is a commercial tool with not enough information.

We observe that most of those tools are applied to (non-deep) machine learning or hidden behind commercial services. Those facts reinforce the claim that our works on chapters 6, 7 are a substantial and original answer to this literature gap.

4.11 Support of large models

The history of deep learning shows that those maths models' quality increase when their complexity increase. Bigger is a DNN and more time it will take to be handled. However, memory is generally the true hard constraint limiting the execution of those models.

This section will describe the most popular method to handle those big models: *model parallelism* and *heterogeneous GPU-CPU memories*.

4.11.1 Model parallelism

Model parallelism consists to split DNNs data structures across multiple GPUs to store DNNs data structures such as parameters, gradients and features flowing through layers. DNNs are generally a sequence of layers and so the cut is performed between layers and generates n portions, each portion is associated with a different GPU among n GPUs.

In general, each portion is computed sequentially, the i^{th} waits for the features from the previous one and gives output to the next one. It makes $n-1$ communications for both forward and backward phases. The "model parallelism" may be confusing, layers are computed sequentially with important communication costs, and no acceleration is expected. To be more accurate, some rare DNN topologies contain different computing branches such that Alexnet and Siamese Neural Networks that may be partially accelerated with model parallelism.

This method succeeds to leverage multiple GPU memories in the condition the computational blocks may be fairly spread across multiple GPUs. For example, VGG16 containing 16 convolution layers cannot model-parallel properly with more than 16 GPUs. And more, the convolution layers may have different settings and different resources requirement.

Multiple approaches have been proposed to tackle the computing time such as data-parallelism and stale gradient mixed with the model-parallelism. Model-parallelism is experimented in more details in section 5.4.

4.11.2 Heterogeneous GPU-CPU memories

As seen in section 4.2 the GPU suits well the deep learning operations compared to the CPU. On the opposite, the CPU memory (RAM) is significantly bigger than the GPU memory (VRAM). An ideal computation design should implement DNNs computing on the GPU and the DNN parameters stored on the CPU with asynchronous marshaling features an exchange between the GPU and the CPU.

We observe no deep learning framework still natively implements this heterogeneous GPU-CPU implementation. However, some tools have been popularized such as Large Model Support [84].

4.12 Recap

We begin this chapter by showing the standard life cycle of a machine learning application is iterative. We then show that deep learning is computing and memory intensive by nature.

Transfer learning is generally a good parameter initialization to fast training but is limited to applications where deep neural networks are used on the shelf.

Then, we see that data-parallelism attempts to increase the training speed by exploiting multiple GPUs. It generally requires a large number of experiments before finding optimization settings to limit the communication time with efficient convergence. This thin equilibrium between communication time and efficient training convergence makes it difficult to generalize on any DNN and any task.

We see that the history of deep learning shows that bigger model complexity is generally more accurate, and memory is the true hard constraint in computers. Model parallelism and heterogeneous CPU-GPU memories are procedures allowing for successfully exploiting the memory of multiple devices, but they may introduce a significant communication overhead.

Both AutoML and ensemble learning aims to improve accuracy. They raise multiple challenges on the exact procedure to apply such as the control of the computing cost, the distribution of computing if the accuracy gains with ensemble worth the computing cost...

We then explore state-of-art inference frameworks that are in active development to increase the deployment and their speed. Although float 32 bits format is the most common format, we observe deep learning arithmetics and format is under active research and development: mixed arithmetic is a simple technic to train faster and convert to float 16 bits for faster inference with no or low accuracy loss. In this thesis, we keep using float 32 bits due to serialization issues when mixing technics with current frameworks.

For the following chapters, we will keep in mind the two important pieces of information. First, the deep learning methods require multiple experiments to be well-tuned. The second one is that the most computing-intensive deep learning models (ensembles or big neural networks) produce generally better predictions. To overcome this computation requirement, optimizing the hardware utilization is strategic for machine learning applications. The next chapter is an analysis and comparison of parallel patterns for deep learning applications. The conclusion of this next chapter is also the stepping stone for the contributions introduced in chapters 6, 7, and 8).

5 Asynchronous and distributed Deep Learning analysis

High computing power allows to efficiently handle each step of the large neural networks life cycle. To give the big picture, we begin this chapter (section 5.1) by defining a taxonomy of the possibility offered by HPC for Deep Learning.

The next sections will follow the defined taxonomy point by point.

- **Independant training.** First (section 5.2) HPC allows training independent DNNs. They are multiple reasons why we may require such as multiple machine learning scientists submitting jobs or building highly accurate models.
- **Building accurate model.** Building a very accurate model is either performed with either with hyperparameter optimization (generating multiple DNNs and selects the best one [6] [123], or ensemble learning where multiple DNNs are trained and their prediction are combined [184] [58] [155].
- **Hudge DNN.** The HPC (section 5.4) offers resources to train and infer very big DNNs (>1B parameters) by splitting its representation among heterogeneous memories and cores.
- **Fast training.** Some technics attempt to use the HPC to train fastest a given DNN (section 5.3) by accelerating its convergence speed. Because time-to-accuracy may be far from straightforward when exploiting multi-GPUs, it requires careful tuning of the optimizer with multiple independent experiments run in parallel.
- **Fast inference.** Last but not least, HPC may allow hosting a performant service to infer with previously trained deep neural networks (chapter 8).

We benchmark and provide a detailed analysis of most of those approaches. However, due to the increasing number of frameworks and computing units for deep learning, it is impossible to be exhaustive.

5.1 Overview of parallelism

Achieving machine learning at scale is a complex notion due to the fact they are projects made of multiple steps: such as the tuning of hyperparameters, the training, and the inference requiring different parallelism patterns and facing different scalability issues. The end goal is generally to maximize the end-user quality of service in terms of qualitative predictions and predictions speed. On another side, the infrastructure has progressed into complex heterogeneous architectures containing CPU, and GPUs making the design of deep learning software more and more complex.

A useful taxonomy should achieve two things: firstly categorize and list all possibilities offered by high computing power, the potential methods, and their expected outcome. Secondly to allow us to spot gaps in the literature, allowing us to explore new techniques or merges methods to overcome individual limitations.

The taxonomy is presented in figure 5.1.

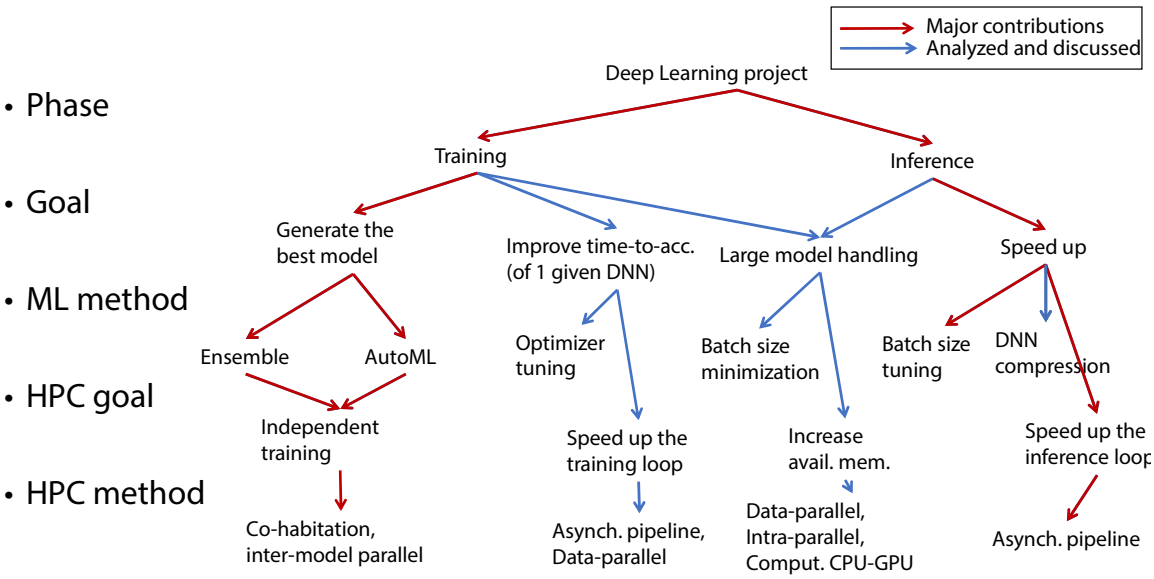

Figure 5.1 – *Taxonomy of works on Deep Learning and HPC*

For instance, we identify that Ensembling and HPO (or "AutoML") aim for the same goal: producing the best models, but comparing both approaches is not possible because they are very different methods: ensemble learning seeks to predict with multiple models, while HPO generates multiple models and select only the best one. They are very different procedures and we ignore a priori which procedures are the most suited for a given task.

This is why we merge both approaches into the same method called "HPO+ensemble". This is the first published contribution to deep learning applications. We dedicate chapter 6

to the first published HPO+Ensemble for computer vision the supervised case and chapter 7 the first one for reinforcement learning.

We also identify that ensemble learning is a well-known method and yet no software allows it to serve them efficiently: no inference server nor inference framework is compatible with the ensemble of independent neural networks. We fill this gap in chapter 8.

This tree allows us to see that the batch size has a lot of impacts on different phenomena. It affects loss convergence, internal parallelism, and memory consumption.

5.1.1 Different distributed patterns for supervised DNNs

Deep learning applications may benefit from computing-intensive infrastructures by implementing different patterns. We attempt in figure 5.2 to be the most exhaustive as possible.

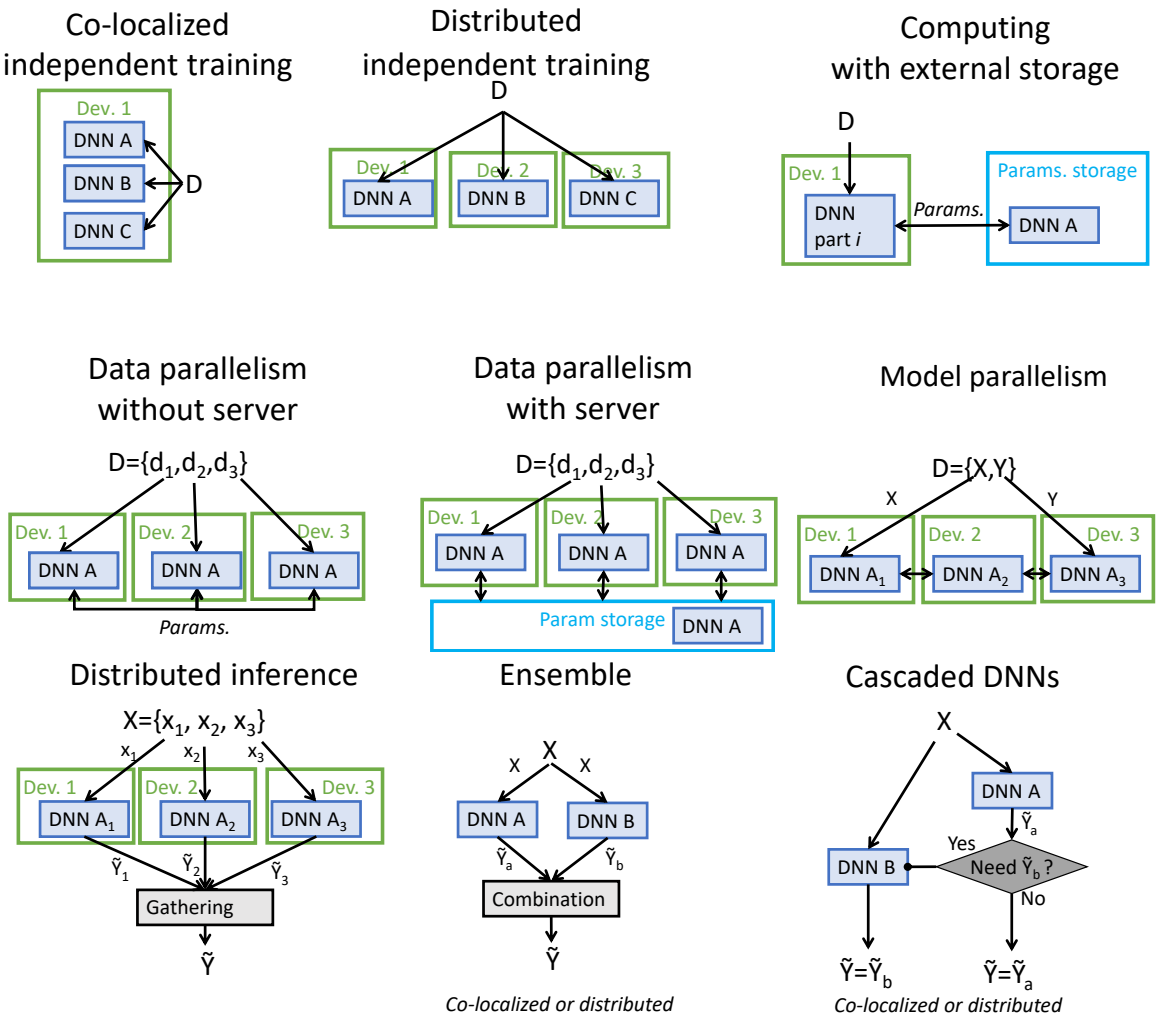

Figure 5.2 – *Different distributed patterns for supervised deep neural networks*

The mapping between applications and computing patterns is flexible. Some deep learning applications may use different parallel patterns. For example, training independent DNNs may be distributed across multiple devices (if available) or can be co-localized into one single powerful device.

On the opposite, some parallel patterns may be useful for multiple deep learning applications, for example, model parallelism may be used in both the training and the

inference phase. In the case of asynchronous data-parallelism with the server, some exploit this pattern to distribute computing to leverage multiple GPUs, but some others [16] train with geographically distant data samples for distributed applications.

Independent training. Independent training of DNNs may deserve multiple purposes: hyperparameter optimization, training an ensemble of deep neural networks, or multiple users submitting different training jobs on an infrastructure. We exploit distributed independent training in this thesis and we enter more details in section 5.2.

Data parallelism. Data parallelism consists of training on different splits of the data samples to attempt to accelerate the overall training. Data parallelism is more in-depth analyzed in section 5.3.3. It is declined in multiple versions: without server [171] where gradients are synchronized with AllReduce operation [31] or with a parameter server [95] allowing to centralize parameters and distribute the computation.

5.1.2 Different distributed patterns for deep reinforcement learning

Multi-threaded Reinforcement Learning training has been proposed under multiples patterns to accelerate multiple algorithms [112] [39] [112] [64].

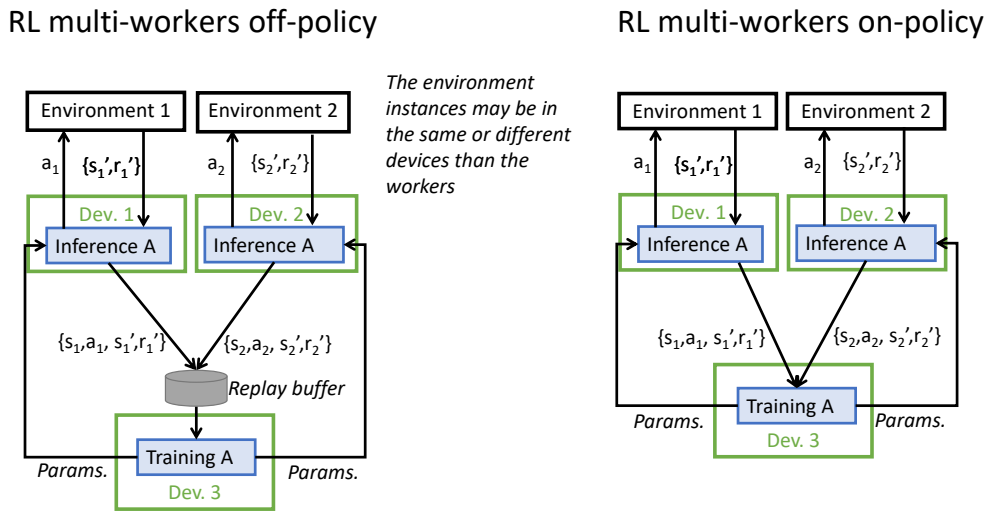

Figure 5.3 – *Two common distributed training patterns for the two reinforcement learning sub-type: on-policy and off-policy.*

We represent in this figure the most common and effective pattern for distributed reinforcement learning. The experience gathering is distributed across multiple cores/-machines with rollout workers. Distributed reinforcement learning is in-depth discussed in section 7.1.2 and compared to independent training when multiple experiments are required. Due to the fact deep reinforcement learning is an experimental field [59] [37],

one should not formulate the workload as training faster one single agent, but instead compute faster a workload of RL experiments.

5.2 Different independent training

This section describes how n experiments may be run on m devices efficiently, with generally $n \geq m$.

The software architecture is typically launched with three steps who launch processes:

- First we launch the **master** (or scheduler). It runs the HPO algorithm and sends jobs (1 experiment = 1 job) to the worked pool. All communications are TCP/IP between client-master and between master-workers. It schedules the jobs on the underlying hardware. It is aware of how much a training consumes resources (typically, one GPU) and it is informed of how much there are resources in the devices.
- Second, we launch a **worker pool** which is parameterized with the number of nodes to use and the number of GPUs to use by node. They are associated with a GPU, when the GPU is free, they request one trial to run to the master. and share memory between worker
- Finally we run **client** which sends Python code to the master. It contains the HPO algorithm and the CNN framework

This general-purpose scheme fits perfectly with the computation need of the majority of parallel HPO algorithms [110] [133] [98].

5.3 Efficient training workflow

Speed up training offers faster iteration times, and the ability to explore more ideas. Increased training iteration speed ("time-to-epoch") is not always relevant if the generalization ability is sacrificed [62]. That is why both time and accuracy must be considered together. Accelerating the training speed may be done in multiple ways: improving I/O speed, prefetching, asynchronous training between I/O and training, data parallelism, and mixed precision. In our applications, accelerating further I/O is not a priority because they are compute-bound applications.

5.3.1 Dataset file format

Supervised algorithms require big data to produce qualitative predictions. In this section, we compare multiple formats and technics on our data at TotalEnergies. Results are summarized on table 5.1.

There is no ideal format because it depends on the context of the application. we enumerate strategic characteristics before choosing a dataset file format:

		Microfossils		Sentinel		Comment
		I/O sec.	Comp. rate	I/O sec.	Comp. rate	
Without compression	numpy	1.54	1	0.67	1	
	pickle	2.99	1	1.41	1	
	hdf5	3.12	1	0.94	1	
Only converted	numpy float16	13.52	2	6.77	2	
	numpy uint8	12.48	4	5.84	4	
Generic compression	numpy zip	32.76	10.45	15.9	8.02	
	lzma	-	-	-	-	Stopped after > 1 hour
	zlib comp=1	26	8.4	12.14	7.16	
	zlib comp=9	23.16	10.65	11.45	8.02	
	bz2 comp=1	194.41	15.63	113.31	9.16	
	bz2 comp=9	217.1	19.27	133.31	12.33	
	blosc	Error	Error	Error	Error	Error "too much bytes"
Compress individually each image	tiff	27.74	17.38	14.18	8.74	
	jpg	24.71	15.62	16.62	16.83	Microfossils MSE=2.3e-5 and Sentinel MSE=1e-4
	jp2	277.47	10.61	Error	Error	Decompression error
	png	37.86	7.1	20.85	5.26	
Compress as a mosaic	tiff	24.92	19.95	N/A	N/A	Microfossils images are stored as a 125*160 mosaic
	jpg	22.96	115.21	N/A	N/A	
	jp2	Error	Error	N/A	N/A	Microfossils decompression error
	png	35.14	20.4	N/A	N/A	

Table 5.1 – *Different file format, reading speed (I/O) and compression factor (C) on two different chunks of training dataset: 20000 micro-fossils small images, and 1 big sentinel image. Some formats performance are data dependant such as zip and jpg*

- Reading time (I/O time + decompression time)
- Mean compression rate
- Mean loss introduced
- I/O pattern. Many small I/Os or a few big I/Os

We evaluate HDF5 [44] in figure 5.4 to parallel the reading with different settings. The best HDF5 settings is still slower than the raw file format.

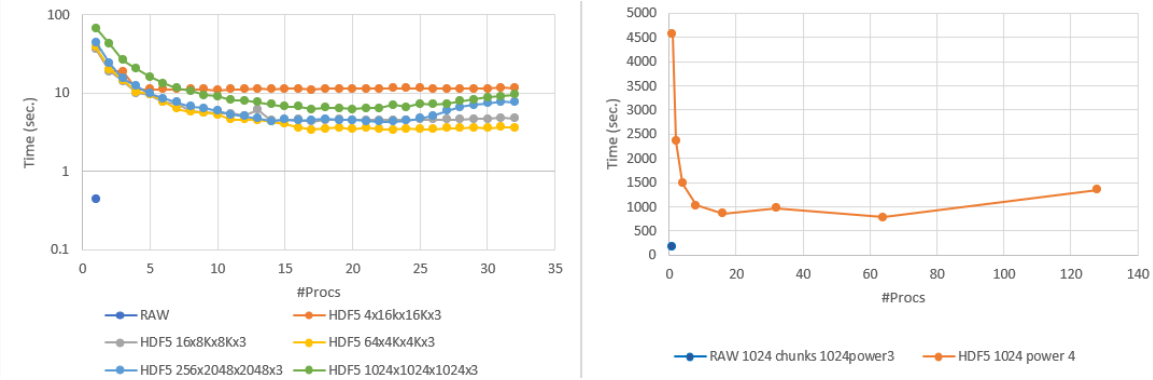

Figure 5.4 – *Format comparison between raw and HDF5 files format. Left the chunk of size* $1024^3 \times 3$ *and the right one is of size* 1024^4

To summarize, the raw file format is the fastest lossless file format. In the case where the I/O is bottlenecking (distant data samples, disk stress) the JPG file in a mosaic achieves a higher compression rate by factor 7 compared to naively stored independent files, without affecting significantly the MSE values. The PNG compression rate is multiplied by 3 compared to the independent files. Those results show that to improve the image compression algorithms, storing images side-by-side in a bigger file is beneficial.

5.3.2 Training workflow

The choice of file format is argued in the previous section and the best one is dependent on the application. In the remaining of this thesis, we keep the original dataset built by experts and copy it for machine learning purposes. The ML dataset must be regenerated each time the original one is updated.

We safely load the DNN based on the "try-except" Python code instructions. We try to load the DNN on a GPU and we catch "out of memory" of the GPU. If it does not fit on a GPU either a large model training approach (section 5.4) or we use another deep learning architecture. In case of a hardware crash or divergence, the DNN is checkpointed regularly on the disk.

The trainer loop is built of multiple threads: The reader performs reading instructions, the preprocessor performs batches and data augmentation on the CPU, and the trainer runs the gradient descent on the GPU. Some additional processes may be implemented such as the evaluatator performing and displaying metrics to the engineer.

The compilation and time-optimization of neural networks training is an active field of research with recent initiatives such as XLA [85].

5.3.3 SGD data parallelism

Data-parallelism aims to accelerate the training time of one candidate DNN (or "trial"), but despite hardware improvement (such as NVLINK [115]) and performant improvement (such as ring-all-reduce [171]), they are challenging to well scale. This is due to the

relative speed performance compared to forward-backward compute on GPUs and the communication of millions of weights at each batch.

Some authors [177] succeed to get good scalability on thousands of GPUs based on hudge batch size to reduce communication and a very careful learning rate scheduling. However, the scalability is hard to reproduce on other DNNs because a huge batch size sharpens the loss function and increases the sensibility of the optimizer, and is prone to overfitting.

Experimental results Figure 5.5 shows the validation accuracy when we vary the number of data-parallel workers and diverse batch size. A complete benchmark should also assess different optimizer settings such the learning rate, the optimizer (e.g. Adam [76], SGD, SGD with momentum [128] ...).

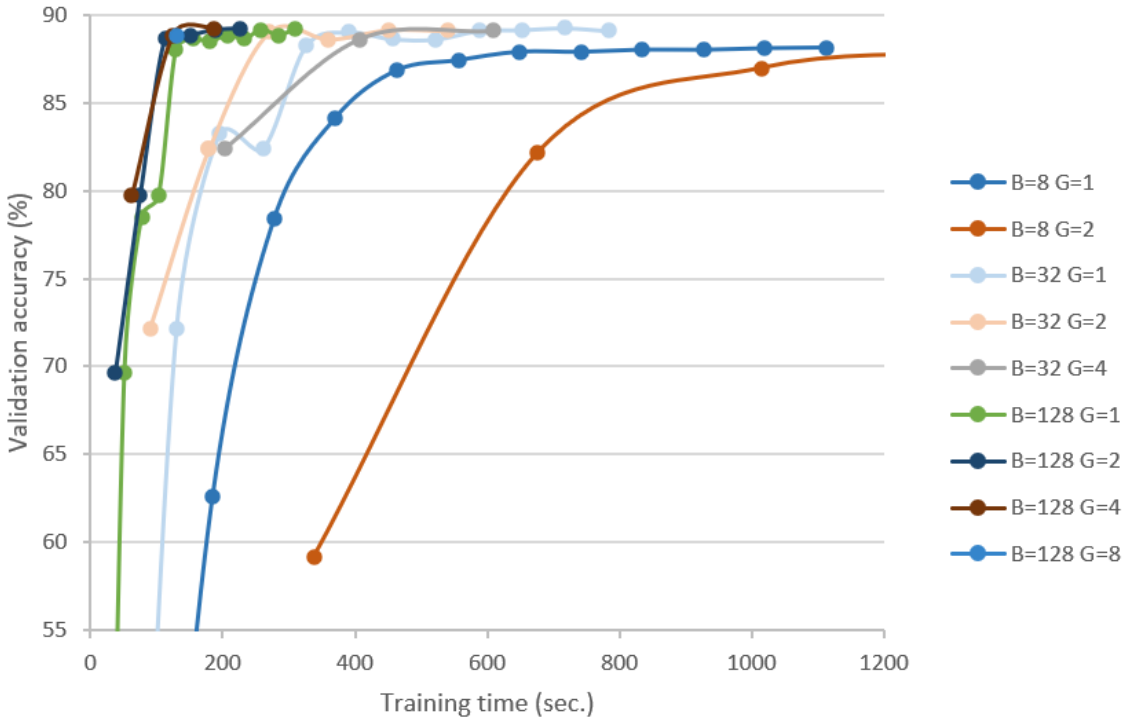

Figure 5.5 – *Effect of data parallelism a ResNet50 on the accuracy-over-time trained on the microfossile dataset with SGD and the 0.001 learning rate. We vary the global batch size ("B"), and the number of GPUs ("G") computing the synchronous SGD. GPU are NVIDIA Tesla V100 in a DGX1 server.*

We run those experiments on a DGX1 computer node. It contains multiple GPUs and implements performant connectors between them to accelerate GPU communication, the "NVLINK". We observe that only a few GPUs are directly connected with NVLINK technology and the other one with PCI-EXPRESS. The complete interconnection topology is presented in figure 5.6. In all data parallel experiments shown in figure 5.5, we exploit contiguously the GPU in ascending order beginning with the GPU 0.

```
      GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    GPU6    GPU7
GPU0   X      NV1     NV1     NV2     NV2     SYS     SYS     SYS
GPU1  NV1      X      NV2     NV1     SYS     NV2     SYS     SYS
GPU2  NV1     NV2      X      NV2     SYS     SYS     NV1     SYS
GPU3  NV2     NV1     NV2      X      SYS     SYS     SYS     NV1
GPU4  NV2     SYS     SYS     SYS      X      NV1     NV1     NV2
GPU5  SYS     NV2     SYS     SYS     NV1      X      NV2     NV1
GPU6  SYS     SYS     NV1     SYS     NV1     NV2      X      NV2
GPU7  SYS     SYS     SYS     NV1     NV2     NV1     NV2      X

Legend:
  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NV#  = Connection traversing a bonded set of # NVLinks
```

Figure 5.6 – *Connection topology between GPUs in a DGX1 server. Each GPU may deliver up to delivers 125 teraFLOPS of deep learning. NVLINKv1 links transfer gradients up to 20GB/sec. PCIe links are 16 lanes of PCIe 3.0 delivering 15.8GB/sec.*

The literature tries different data-parallel methods mainly focusing on the best procedure to optimize leaving out the neural architecture search as a different problem. And yet, the architecture and the parallel optimizer are linked. First, the best data-parallel optimizer and its settings (e.g., batch size, learning rate, number of GPUs) for one neural network may not ideal for another neural network. Second, some neural networks are easier than others to scale [171]. In absence of further explanations, we analyze different neural networks and observe they are different computing/communication rates. The table 5.2 shows the estimation of computation and communication ratio.

Neural network	Forward time (sec.)	Parameters (M)	Comp/Comm ratio
MobileNet	2.12	4.3	0.492
MobileNetV2 [137]	2.20	3.5	0.628
MobileNetV3S	1.56	2.5	0.626
MobileNetV3L	2.18	5.4	0.405
ResNet50V2	2.00	25.6	0.078
ResNet101V2	3.54	44.7	0.079
ResNet152V2	4.83	60.4	0.080
NASNetMobile	1.87	5.3	0.352
NASNetLarge	17.33	88.9	0.195
Xception	5.49	22.9	0.240
InceptionV3 [155]	2.88	23.9	0.120
InceptionResnetV2 [156]	2.08	8.1	0.257
DenseNet169	2.28	14.3	0.159
DenseNet201	2.84	20.2	0.141
VGG16 [184]	3.19	138.4	0.023
VGG19	3.41	143.7	0.024
ResNet50 [58]	2.95	25.6	0.115
ResNet101	4.03	44.7	0.090
ResNet152	5.02	60.4	0.083
EfficientNetB0 [158]	3.08	5.3	0.581
EfficientNetB1	2.46	7.9	0.311
EfficientNetB2	3.34	9.2	0.364
EfficientNetB3	4.52	12.3	0.368
EfficientNetB4	12.01	19.5	0.616
EfficientNetB5	22.00	30.6	0.719
EfficientNetB6	36.76	43.3	0.849
EfficientNetB7	61.97	66.7	0.929
EfficientNetB0V2	1.21	7.2	0.168
EfficientNetB1V2	2.04	8.2	0.249
EfficientNetB2V2	2.54	10.2	0.249
EfficientNetB3V2	3.67	14.5	0.253
EfficientNetV2S [159]	7.81	21.6	0.361
EfficientNetV2M	19.40	54.4	0.357
EfficientNetV2L	32.08	119	0.270

Table 5.2 – *Estimation of the compute-communication ratio. The "compute" is expressed as the forward time to predict with a 64 batch size (An ideal computation measure should be expressed with not only the forward time but forward+backward but we lack time to obtain those figures). The communication complexity is linearly dependent on the number of weights to exchange between data-parallel threads.*

The forward time is used as an approximation of the training time and the number of

parameters is linearly linked to the communication time. The estimated compute/communication ratio confirms what previous authors [171] experimentally observe: ResNet101 generally scales better than VGG16, and InceptionV3 scales slightly better than ResNet101. This ratio allows knowing quickly if a neural network may have a chance to scale without requiring sweeping several hyperparameters. This method does not give accurate information on the ground-truth scalability but it is a rather heuristic that could allow identifying which neural network has the most chance to scale.

The first major observation is that the last invented convolutional blocks are not only more efficient on one GPU, but we demonstrate they have the potential to better scale on multiple GPUs. The invented EfficientNet convolutional blocks (2019) and EfficientNetV2 (2021) are between 0.3 and 0.9, ResNet convolutional blocks (2016) between 0.08 and 0.012, and than VGG convolutional blocks (2014) around 0.002. The second observation is that this rate varies in one given architecture family. It teaches us that the settings of convolutional blocks are also important for scalability's sake.

Now let's quickly summarize the phenomena interacting together when we want to accelerate the training of any deep neural network on multiple GPUs with data parallelism.

- Smaller batch size values generally improve and accelerate the convergence. However small batch sizes increase the gradient synchronization frequency.
- Bigger batch size generally decreases the convergence and reduces the gradient synchronization
- We list multiple deep neural networks and show that they have very different computing-communicating rates.

Data parallelism does not guarantee to systematically speed up the training convergence of a DNN compared to training it on one single GPU. For example, with the Resnet50 we converge faster with small batch size values and 1 GPU. Therefore, another data parallel was introduced to run asynchronous SGD where weights are centralized on a parameter server. This avoids SGD workers waiting for gradients, so they spend more time iterating locally on SGD iterations and regularly load and update global weights from the server. However, due to the asynchronous nature of this process, the workers may update obsolete versions of the server and send non-consistent gradients ("stale gradients"). This is why it is now accepted [19] that synchronous SGD has a faster convergence than asynchronous SGD.

To conclude, as far we know, no data-parallelism method allows to accelerate the training convergence in all cases compared to one GPU. In the next chapters, we argue that asynchronous hyperparameters such as asynchronous hyperband [91] and random search [6] [90] is an efficient way to leverage from multiple GPUs.

Due to the fact we exploit multiple GPUs to perform many experiments in parallel, we do not exploit NVLINKs connections between them. The hyperparameter optimization

workload may be run on any infrastructure containing multiple GPUs regardless of the links between them.

5.3.4 Mixed precision

As previously seen in section 4.2, deep learning applications need to store accurate gradients (precision around 0) but we generally don't require precision for large values. 32-bits floating point (FP32) is the most common format in deep learning but it raises some criticism on the fact it is not adapted to the deep learning workload. Multiple benefits are expected to use a lower precision format: faster arithmetic operations, faster memory exchange, lower memory consumption, and lower power consumption. This is why lower precision has been introduced under different arithmetic formats and different procedures to improve the training and the inference phase.

Here is the commonly used format with the current GPU:

1. **FP16.** consists in storing weights in FP16 and computing FP16 gradients. It may critically reduce the gradient magnitude precision around zero and keep the large number representation unused.

2. **FP16 with loss scale.** [186] This procedure consists of artificially multiplying by a value the FP16 loss to better use FP16 bits. Authors propose to tune it empirically: disable it, fix a loss scale factor empirically or use a dynamic loss scale value according to the current loss. A too high loss scale value may cause an FP16 overflow that is fatal for the training process.

3. **Mixed Precision.** [186] It aims to improve the DNN training speed. It consists to store and computing the forward and backward phase in FP16 by maintaining an FP32 copy of the DNN. Authors and frameworks propose to use the loss scale to attempt to improve the convergence. Finally, the FP16 copy may be used as the final model for the inference phase.

We evaluate them on our microfossil case in figure 5.7 and the NVIDIA Volta 100 GPU. Dynamic Loss scaling is set with the default parameters of Tensorflow. The fixed loss scaling was performed with 1024 which is a common value in some technical documentations.

Arithmetic name	Float 32	Float 16	Float16 LS fixed	Float16 LS dyna.	Mixed precision LS fixed	Mixed precision LS dyna.
validation acc	85.53%	84.58%	84.11%	82.84%	84.00%	84.68%
training time (sec)	2419	1463	1598	1598	1729	1729
inference time (sec)	17.8	13				

Figure 5.7 – *We perform training of ResNet50 on the micro-fossils dataset by varying the numeric format: FP32 training, FP16 training, FP16 with loss scale and mixed-precision training*

When precision is lowered, it seems unpredictable how the training convergence behaves. This is explained by multiple interleaved phenomena:

- Lower precision increases the iteration speed.

- Lower precision may underflow the weights update.
- The noise introduced may regularize the training. For example, Stochastic Gradient Descent and dropout operation [151] are good examples of intentional noise to improve the generalization ability.

We also assess the simple procedure where the DNN is trained with FP32 and then, it is simply converted in FP16 in figure 5.8. This FP16 model produced is the fastest FP16 produced model but it requires an usual FP32 training.

Arithmetic name	Float 32	Float16
validation acc	85.53%	85.42%
inference time (sec)	17.8	13

Figure 5.8 – *Comparison between FP32 and post-training conversion from FP32 to FP16*

Those complex and unpredictable phenomena reinforce our previous claim on the fact that deep learning is a method requiring multiple experiments. Those results illustrate the importance of launching automatically multiple experiments to select the best performing one based on generalization and its computing speed. The numerical precision should be ideally included in the hyperparameter space to let an automatic smart optimization process tune it according to the desired performance. Despite the potential lower precision gain, in the remaining of this thesis, we will only experiment with FP32 due to serialization issues with current frameworks.

5.4 Large model training

The larger deep neural networks have been known as an effective approach to improving model quality for several tasks [68]. To handle large neural networks, authors propose several methods, a few tools emerged to tackle the challenge of training large neural networks but they are still in an experimental stage. Today, no consensus seems to emerge and each method hurts multiple practical challenges. They are different approaches to balance the ability to exploit multiple memory devices and the speed. This section compares those multiple methods.

5.4.1 Methods

Batch accumulation. A deep neural network replicates the entire model across all data samples from the batch resulting in poor memory efficiency. Batch accumulation consists in eliminating this redundancy. The training with a batch size of size n is performed by cumulating n gradients with a fixed batch size of 1. Once the gradients are known, it performs the usual parameters update. This method has two major drawbacks. First, it is

useless if the neural network cannot be stored with a batch size of 1. Second, 1 batch size generally lets a large proportion of unsued cores and so may.

Data parallelism. Data parallelism was previously discussed to speed up training, but it can allow training of large neural networks too. The scalability depends on the batch size, if the desired batch size is large, multiple GPUs can split the global batch across multiple GPUs. On the opposite, the data parallelism fails to handle a large neural network if the global batch size is already 1.

More generally, if the desired n batch size and g GPUs (such as n divisible by g). One GPU must be able to store the neural network with a batch size of size n/g. When n is small, the data parallelism is not a viable solution because it makes it impossible the division of the workload across multiple GPUs. In general, when the neural network is large such that U-net [134] or 3D U-net [24] the chosen batch size value chosen by scientists is generally small: 1, 2, or 4. We may conclude by saying that the data parallelism approach is not a viable method to handle a large neural network.

Model parallelism Model parallelism [78] is described in section 4.11.1. This method succeeds to leverage multiple GPU memories in the condition the computational blocks may be fairly spread across multiple GPUs.

Pipeline parallelism Combine model-parallelism and data-parallelism have been proposed aiming both to train large models and to accelerate the training speed.

Gpipe [68] attempts to increase the training speed by mixing model-parallelism and data-parallelism with synchronous SGD. In our experiments, the data-parallelism fails to accelerate the training in all cases due to high communication costs between workers like previously seen in section 5.3.3.

To attempt to alleviate this, PipeDream [114] has been proposed, it performs model-parallelism and asynchronously SGD with stale gradients. Even if PipeDream accelerates the training iterations compared to GPIPE, better convergence has not been shown in practice.

5.4.2 Experimental results

The table 5.3 shows all those benchmarks.

Model parallelism is the only assessed method that can scale to multiple GPUs to store huge neural networks. However, model parallelism requires the developer to spend time mapping tensors and devices memory, this is quickly painful when multiple neural networks must be managed and multiple GPU counts. Fortunately, some methods exist to perform automatic model parallelism, this is the idea explored in the next section.

5.4.3 Automatic model parallelism experimental results

Automatic model parallelism is a method to automatically split a large neural network to fit it in multiple devices' memory. We use the Gpipe [68] method to automatically allocate tensor operations to devices implemented with the TorchGpipe framework. We disable

	Initialization time (sec.)						
Depth factor	x1	x2	x4	x8	x16	x32	x64
#layers	152	302	602	1202	2402	4802	9602
1 GPU	53.8	105.5	233.2	OOM	OOM	OOM	OOM
1 GPU accum.	107.6	304.8	1104.8	4294.1	OOM	OOM	OOM
1 CPU	112.1	227.2	434.7	913	1939.9	OOM	OOM
LMS	69.6	166.4	470.4	17.6	OOM	OOM	OOM
4GPUs data-//	34.8	66.8	137.1	277.7	583.7	OOM	OOM
4GPUs-model//	77	102.4	115.7	212	579	OOM	OOM
8GPUs-model//	65.9	66.1	99.8	174.5	351	554.3	OOM
	Time to train 1 epoch on 16 images (sec.)						
Depth factor	x1	x2	x4	x8	x16	x32	x64
#layers	152	302	602	1202	2402	4802	9602
1 GPU	1.3	2.2	4.3	OOM	OOM	OOM	OOM
1 GPU accum.	3.7	7.2	14.9	33.1	OOM	OOM	OOM
1 CPU	11	21.2	70.3	122.1	222.4	OOM	OOM
LMS	2.5	4.4	8.4	17.6	OOM	OOM	OOM
4GPUs data-//	2.4	4.7	9.9	21.2	46.1	OOM	OOM
4GPUs-model//	1.2	1.6	2	3.4	6.2	OOM	OOM
8GPUs-model//	1.1	1.2	1.5	2.3	5.5	7.4	OOM

Table 5.3 – *The number of convolutional blocks of the ResNet152 is multiplied to benchmark different methods to handle huge neural networks. The training time is given to train 4 batches of 4 data samples. The mention "OOM" stands for "out of memory".*

ResNet152 conv. multiplier	**x1**	**x2**	**x4**	**x8**	**x16**	**x32**	**x64**	**x128**	**x256**
Number of parameters	**60M**	**116M**	**228M**	**452M**	**901M**	**1.8B**	**3.6B**	**7.2B**	**14.4B**
1 Tesla GPU (no GPIPE)	1.3	2.2	4.3	OOM	OOM	OOM	OOM	OOM	OOM
2 Tesla GPUs	1.5	2.6	4.7	9.1	17.9	38.5	OOM	OOM	OOM
4 Tesla GPUs	1.8	3.2	6.1	10.9	21.8	47.1	OOM	OOM	OOM
8 Tesla GPUs	1.9	3	5.5	10.6	19	39.5	78.4	OOM	OOM
8 Tesla GPUs + 1 CPU	11.4	21	41.8	82.2	166.3	466	666.7	1353.2	2504.3
1 Amper GPU (no GPIPE)	0.4	0.8	1.6	3.3	6.8	OOM	OOM	OOM	OOM
4 Amper GPUs	1	1.8	3.7	7	13.6	27.1	56.2	OOM	OOM
4 Amper GPUs+1CPU	2.2	4	7.4	14.9	28.2	56.9	113.6	207.8	OOM

Table 5.4 – *Automatic model parallelism with TorchGPIPE by multiplying the ResNet152 convolutional blocks from 1 to 256.*

data parallelism because the additional communication cost increases the training time in the majority of cases.

Experimental results. We multiply the ResNet152 DNN convolutional blocks from 1 to 256. Values are expressed as time (sec.) to train 4 batches of 4 images. OOM stands for "out-of-memory'' when at least one device cannot store all the attributed tensors by TorchGPIPE.

Experimental results are given in table 5.4.

Experimental results. We observe multiple phenomenon when we increase the number of GPU in model parallelism:

- **Memory scalability.** Model parallelism successes to scale the memory. For example, 1 GPU may handle ResNet152x4, while 8 GPUs may handle ResNet152x64, a progressive increase in the DNN size should exhibit the 8 factors. We also observe that the memory of the CPU may be exploited but dramatically divide the training speed between 2 and 8.5 according to the computing node type.

- **Communication time.** Computations are performed with sequential dependency between layers, which leads to only one active GPU and letting the other ones idle. The communication overhead may be estimated with a weak scalability analysis. ResNet154x4 takes 4.3 seconds in 1 GPU and 5.5 on 8 GPUs.

- **Alleviate the GPUs.** Splitting the computational graph across multiple GPUs may allow reducing the amount of needed memory per GPU, the memory traffic, and different allocation between operations and cores. In some cases, model parallelism accelerates slightly the training loop. For example, ResNet154x8 takes 5.5 seconds on 8 GPUs against 6.1 on 4 GPUs.

The "communication time" mixed with "alleviate GPU" have opposite effects on the overall time, making the neural network computation time difficult to estimate a priori. The complex relationships between phenomena illustrate that higher speed under memory constraints may be obtained experimentally by varying some parameters such as batch size and parallelism. This is what we do later in the context of the inference in chapter 8.

Conclusion.

To be efficiently performed neural network architecture design must be tuned not only to optimize prediction quality but to target given hardware architecture. For instance, using model parallelism efficiently requires generally that the neural network can be split into fine grains blocks to spread the memory consumption across target devices. Mixing model-parallelism and data-parallelism have been proposed, but data parallelism is challenging to tune in practice due to the large communication cost at each batch and convergence issues.

Another approach is to perform ensembles made of an independent neural network to then, combine their predictions. Training a larger neural network than one hurts multiple performance issues due to the sequential dependency between layers. It is reasonable to think that training multiple smaller neural networks to fit and combine their predictions may be an efficient turnaround. This claim is reinforced by the fact we previously discovered that an ensemble of DNNs from a library of DNNs, is often Pareto superior to the best one regarding the accuracy and computational time.

5.5 Inference systems

It is common knowledge that machine learning tuning and training are time bounded and intense computing activities, while the inference phase serves its predictions for a long period. This is why optimizing the inference time is sometimes considered more strategic.

An inference deep learning framework consists in predicting with an already-trained neural network. This section analyzes the performance of prediction of some frameworks: Tensorflow, ONNX, OpenVINO, and TensorRT benchmarked on diverse computer vision neural networks. For each framework, we gather more than 80 values including throughputs (predictions per second), latencies (time to predict one image), load time, and memory consumption on both GPU and CPU. We provide a comprehensive review of used optimization technics and evaluate them to guide the inference framework choice.

5.5.1 Post-training optimization

As seen in the ML life cycle diagram 4.2, an intermediate step may exist between training and inference to optimize the model speed. We classify possible optimize technics falling into four categories and shown in figure 5.5.

	Graph representation	Low level representation
No effect on accuracy	Fusing [30] [85]	Code optimization [94] [83]
Effect on accuracy	Pruning [54] [117]	Lower precision arithmetic [165] [27]

Table 5.5 – *Different post-training optimization*

These high-level and low-level optimizations are orthogonal to global optimization,

	Width #params/#layers	Depth #layers	Density #jumps	Params	Jump type
VGG19	7.26M	19	0	138M	N/A
ResNet50	0.52M	50	16	26M	Additions
DensetNet201	0.1M	201	98	20M	Concatenations
EfficientNetB0	0.06M	89	25	5.3M	16 Mult. and 9 Add.

Table 5.6 – *Four deep neural network architecture. We choose them diverse in terms of depth (# layers), width (# number of filters) and density (# number of jumps).*

and they can be used together to obtain further performances. We may also notice that some optimization is beneficial to both training and inference such as XLA [85].

A critical part of optimizing the performance of a DNN model is its batch size. It controls the internal cores utilization, the memory consumption, and the data exchange between the CPU containing input data and the device supporting the DNN (if different). That is why some tools [1] scan multiple batch size values of a given DNN. Then, the batch size offering the best performance is used. This Best Batch Strategy (or BBS) is a relevant mechanism to optimize a single DNN on one device, but it is a naive and rigid method to optimize multiple DNNs predicting together.

Inference frameworks [30], [47], [138], [144] allows to load the DNN, to optimize with it and to serve the DNN in production.

5.5.2 Experimental settings

Neural networks. The ML scientists, and AutoML have developped a variety of viable neural networks. We present four of them in table 5.6.

GPU technologies have evolved in computing devices containing thousands of cores and gigabytes of memory, and modern deep neural networks may not guarantee the use of all resources in one single GPU. To match the ML community computing needs, an ideal inference framework should not only efficiently leverage its resources to one DNN but should be performant to run multiple DNNs at the same time. That is why propose the ensembles in our thesis. All values in parentheses are our evaluated test top1-accuracy on ImageNet: VGG19 (69.80%), ResNet50 (74.26%), DenseNet201 (75.11%) and ensemble them by interpolating their predictions performs 76.92%. We choose not to ensemble efficient-netB0 due to operations errors with some frameworks.

Platform. Our benchmarks are performed on two machines. Evaluations are made on only one GPU at a time.

Machine A is an HGX1 equipped with Tesla V100 SXM2 GPUs containing 5120 Cuda cores running at 1312Mhz-1530Mhz frequency and 16G of GPU memory. Its CPU is a

[1] "https://github.com/triton-inference-server/model_analyzer/blob/main/docs/config_search.md"

dual-socket Intel(R) Xeon(R) CPU E5-2698 v4 with a total of 80 cores @ 1.2Ghz-3.6Ghz and 512GB of RAM. The total GPU board power is 300 watts.

Machine B is an in-house machine equipped with NVIDIA Amper A100 PCI-E GPUs, each one containing 6912 Cuda cores running at 765Mhz-1410Mhz and 40G of GPU memory. Its CPU is a single socket AMD EPYC 7F52 with 16 cores running at 2.5Ghz-3.5Ghz and 256GB of RAM. The total GPU board power is 250 watts.

Software stack. All software stack versions are the same on both machines: machine A and machine B. The assessed inference frameworks versions are Tensorflow 2.6, TensorRT 8.0, Onnx-runtime 1.10, and OpenVINO 2021. The needed neural network converters to benchmark those frameworks are tf2onnx 1.9.3, and LLVM 14.0.

Machine A is running on Ubuntu and we use Python 3.9, Machine B is running on CentOS and uses Python 3.8. Specialized OS for real-time may improve latency determinism and latency speed but could lower throughput performance too. We evaluated MLIR (onnx-mlir 0.2 framework [83]) on Machine B but it support only ResNet50.

In all our benchmarks Tensorflow is accelerated with XLA [85] (Accelerated Linear Algebra).

Graph optimization settings:

- **Tensorflow** [1]: The computing graph is frozen (i.e., all weights are put in "read-only memory"). The optimizer "optimize_for_inference_lib" is enabled but it does not impact the performance. XLA is enabled on GPU, disabling it reduces speed by 15%. However, enabling XLA multiplies the initialization time by factor 6. And more, we do not observe performance gain to enable it on the CPU so we let it disabled on the CPU.

- **ONNX-RT** [138]: Caching is enabled, disabling it reduces speed by 3%. The maximum graph level optimization is enabled, using the default optimization settings reduces speed by 8%.

- **OpenVINO** [47]: The NCHW tensor format is enabled meaning all images must be converted in this format. NCHW means tensors follow this format: images, channels, height, and width. The convolution fusing is enabled, disabling it reduces speed by 9%. The concatenation optimization is disabled because it would hurt performance by 1% on DenseNet201.

- **MLIR** [83] (Multi-Level Intermediate Representation from the LLVM project) is still under development and critical issues occur. When it works, the graphs are compiled with "-O3" option.

5.5.3 Experimental results

Not all metrics are equally important for all ML applications, and weighting some more heavily than others is highly application-dependent. This is why we bench multiple metrics.

We identify three main scenarios and associated speed measures:

- **Batch applications.** First, when the neural network predicts the large workload of data samples, such as predicting from a large folder of images, in this case, the batch size must be optimized to provide maximum predictions per second, this is the throughput. The throughput may be expressed as the number of predictions per second.
- **Data flow applications.** Second, when data samples come sequentially such as in real-time embedded systems, Markov Decision Process (deep reinforcement learning)... the latency is measured as the number of milliseconds to get one prediction.
- **Irregular batch applications.** The third scenario is when the data samples come to an irregular frequency such as web service receiving requests from multiple clients, in this case, the batch computing must be adapted to the client requests characteristics [28] and the metric carefully weighted between the latency for reactiveness and the throughput when the service is under a heavy workload of requests.

We may identify other technical measures:

- **Memory consumption.** Lower memory consumption allows for managing more deep learning models and higher complexity ones.
- **Initialization time.** It is the elapsed time from the neural network representation on the disk and its state in memory where it is ready to predict. Some applications may require a fast service setup to satisfy peak demand periods (e.g., elastic cloud service [142]). In all our benchmarks, we enable or implement a cache system to measure the loading time.
- **Power consumption.**

The **power consumption** is an important metric that interests companies and scientists for diverse reasons such as limiting ecological footprint, energy bills, reducing thermal problems... We express here the power consumption as watt-seconds of the inference system to predict a fixed amount of D data samples, with $D = 4096$. Like the throughput, the value depends on the considered neural network, the chosen inference engine, and the batch size. Then, this measures may be converted in equivalent in CO_2 equivalent [2], energy bill or thermal dissipation... We express the power consumption E here as watt/sec to predict a fixed amount of data samples D, we fix here $D = 4096$. E is described with the equation 5.1, with W the mean instantaneous power consumption (watts) during the prediction time T (sec.).

$$E = D \times W \times T \tag{5.1}$$

Some devices such as Nvidia GPUs contains captors to measure power consumption. We use the below command:

[2] https://www.bilans-ges.ademe.fr/docutheque/docs/[Base%20Carbone]%20Documentation%20g%C3%A9n%C3%A9rale%20v13.pdf

```
nvidia-smi -i ${GPUID} --format=csv --query-gpu=power.draw --loop-ms=3000
```

Figure 5.9 shows the correlation between throughput of an inference system and its power consumption.

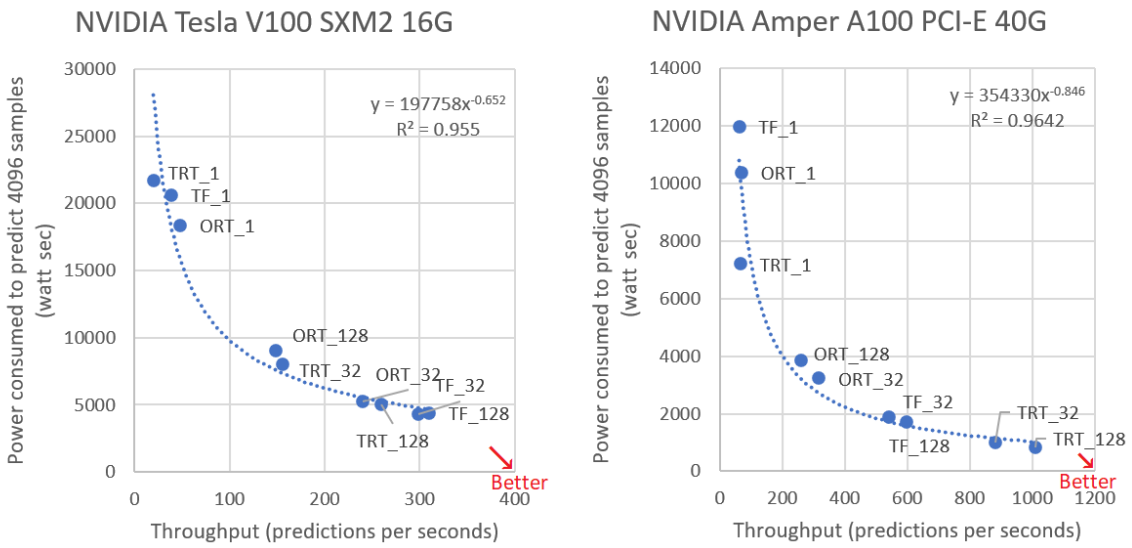

Figure 5.9 – *Plot of the throughput and power consumption of 3 GPU inference frameworks ("TRT" for TensorRT, "TF" for Tensorflow, "ORT" for ONNX-RT) with {1,32,128} batch size. The trend seems to follow the power law with two different coefficients according different GPU.*

The **throughput** results on machine A and machine B are presented in figure 5.10 and in figure 5.11. We decided to not shown the latencies benchmark because it is correlated with the throughput with the batch size of 1.

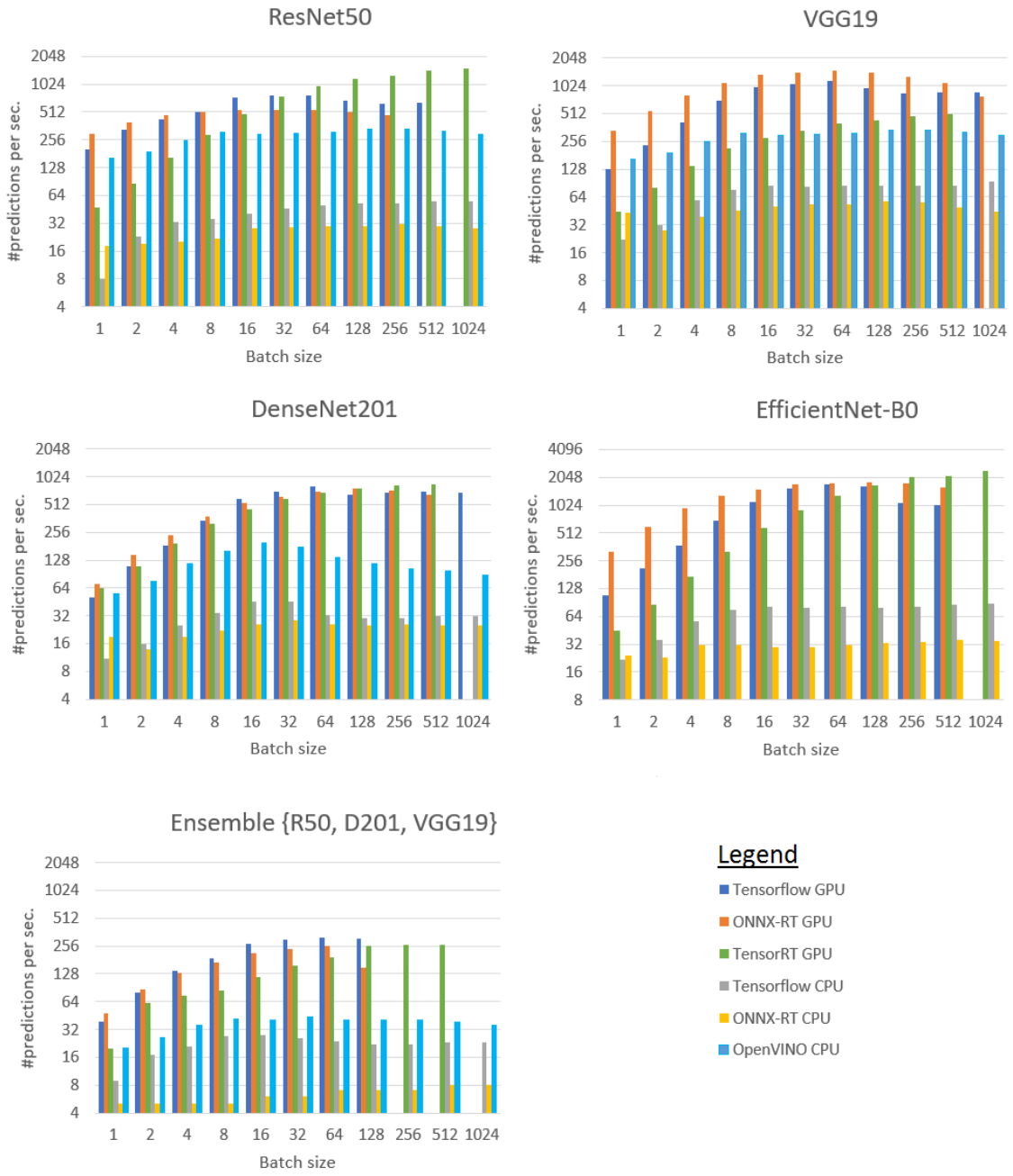

Figure 5.10 – *Inference time (sec.) according different deep learning model and different inference technologies on the machine A. No bar means out-of-memory error conversion fail.*

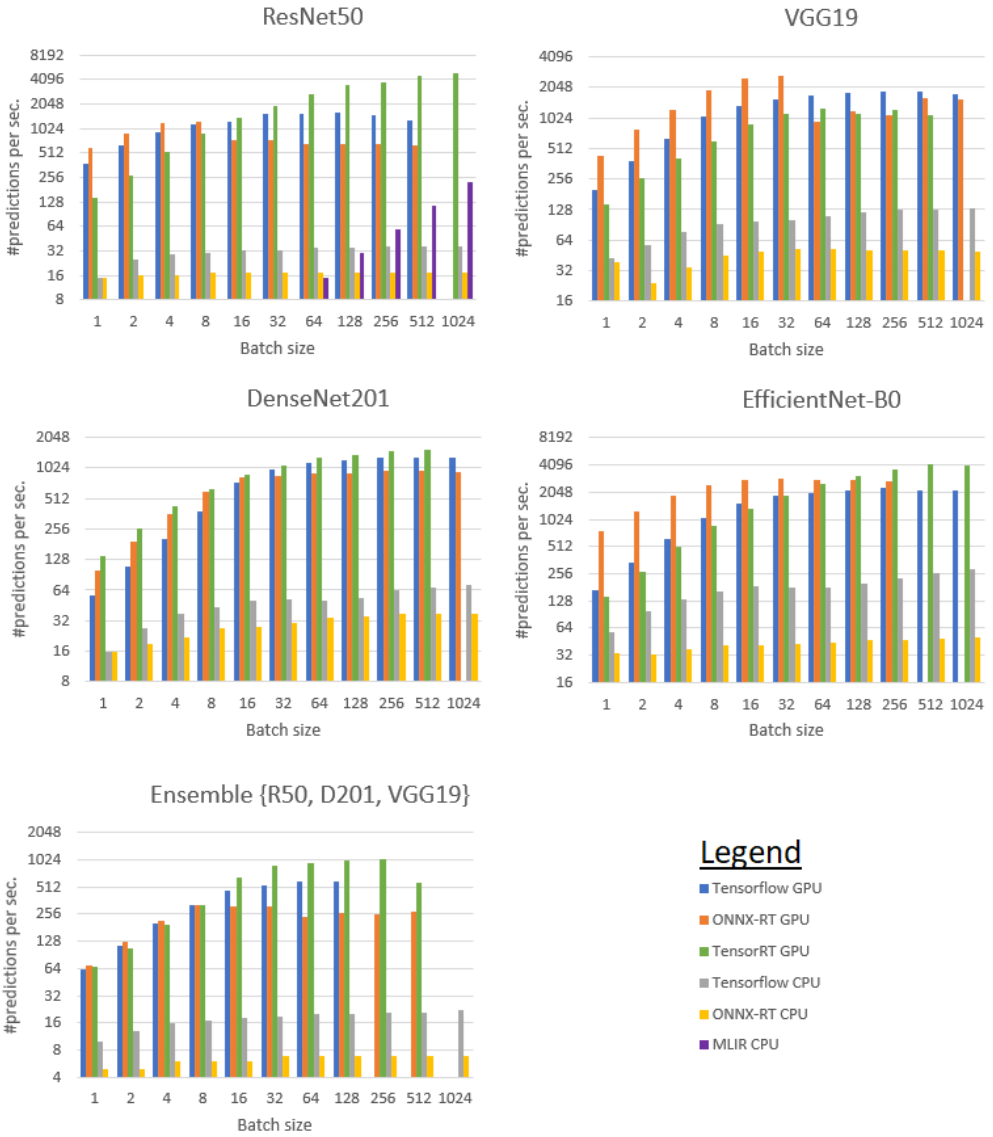

Figure 5.11 – *Inference time (sec.) according different deep learning model and different inference technologies on the machine B.*

The **memory consumption** is expressed as how many the neural network took on the disk plus its current batch. To make a global statistics we measure the ensemble and show it in figure 5.12.

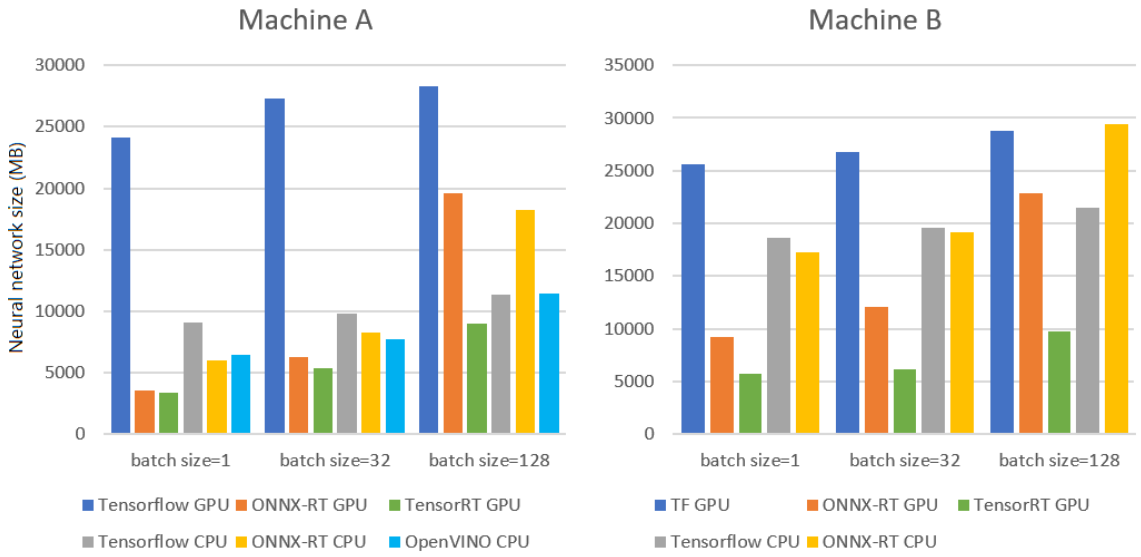

Figure 5.12 – *Memory consumption (MB) of the ensemble with different frameworks varying the batch size. We use this ensemble due to the diversity of topology between neural network: one have wide convolution, one is deep, one is dense.*

The **loading times** (seconds) of the ensemble on machine A in the ascending order: OpenVINO (3.4), TensorRT(3.6), ONNX-RT GPU(5.7), ONNX-RT CPU(8.7), Tensorflow CPU (8.7), TensorFlow XLA GPU (42.5). There are loading times for machine B: ONNX-RT CPU (0.5), ONNX-RT GPU (2.1), Tensorflow CPU (2.9), TensorRT (3.9), TensorFlow XLA GPU (29.2). We observe no clear winner between CPU and GPU technologies but Tensorflow with XLA is especially slow due to the fact we cannot cache the optimized graph on the disk.

5.5.4 Performance summary

Regarding GPU technologies there is no clear winner between TensorRT, Tensorflow, and ONNX-RT, and selecting the best one will depend on the application scenario and the underlying hardware.

ONNX-RT is systematically the fastest in terms of latency and low batch size throughput. TensorRT is faster on large batch size values such as 512 and 1024 but it may perform poorly on wide convolutions. TensorRT is also the framework that consumes the least memory.

If we look at CPU optimization, Intel OpenVINO on machine A is always faster than the others compared to ONNX-RT CPU, and Tensorflow-CPU and MLIR show promising results on machine B even if it is still under development. Intel OpenVINO is also well optimized to save memory.

In terms of power consumption, there is a connection between two phenomena:

- Phenomenon (A). The faster a workload is finished, the less time it has to consume power (kWh).

- Phenomenon (B). The faster a workload is finished, the more hardware piece has been exploited at the same time (cores, memory exchanges). Therefore, more instantaneous energy is consumed by the GPU (watt).

This benchmark reveals that phenomenon (A) is of greater importance than B. In this case, faster means more power efficiency.

When an inference framework runs an optimized graph, it has less time to consume power but, on the opposite, it uses more pieces of hardware in parallel and consumes more energy (watt). Overall, our experiments on GPUs show that accelerating computing is generally beneficial for power consumption.

Finally, these experiments show the need for different frameworks for training and inference because they require different representations and optimizations. As expected, the more optimized and specialized a framework for inference and the more efficient it is. However, some inference frameworks are faster for small batches while some others are faster with bigger ones. Therefore, the best framework depends on the targeted application.

5.6 Handling large deep learning workload

The key to the success of machine learning is that bigger models achieved very high accuracy. We explore in this section different scenarios of large model workloads and the different suitable parallel patterns.

We consider here two big deep learning models type: the first one is a single large deep neural network, and the second one is an ensemble of multiple medium-sized deep neural networks predicting together. We explore a few common scenarios when dealing with GPU infrastructures. The table 5.7 summarizes previous claims of this chapter and illustrates and compares the computing flexibility of the two model types.

Hardware	1 big DNN training and inference	n medium DNNs training workload	n medium DNNs inference workload
1 GPU lacking memory	Swaping GPU-CPU 4.11.2	Schedule them 5.2	Spread GPU and CPU
1 GPU with enough memory	Handle it	Schedule them 5.2	Co-localization
multi-GPU, each GPU lacking memory	Model parallelism 4.11.1, and/or Data parallelism 5.3.3	Schedule them 5.2	Spread them across GPUs
multi-GPU, each GPU have enough	Data parallelism 4.11.1, or sequentially	Schedule them 5.2	Data parallelism, and/or co-localization

Table 5.7 – *The most suitable parallel patterns according to different common situations. Scheduling may not only ensure they fit in memory, but also improve an efficiency measure. For example, the figure 7.15 shows us that handling 4-by-4 the RL agent training workload is the most energy-efficient assignment whereas 8-by-8 is the faster one.*

While a large neural network is a sequence of computation blocks, an ensemble of deep neural networks is made of independent computation blocks with only predictions

transferred between them. This makes the training ensemble workload easily to efficiently parallelize with no communication between blocks.

The table and arguments are focused on parallel patterns. Some methods exist to reduce memory utilization in one single GPU, such as gradient checkpointing, lower precision arithmetic, batch accumulation, or changing the hyperparameters to reduce its complexity... Those technics generally come with either a more computation time increase or an accuracy reduction. More generally speaking, those three characteristics seem impossible to gather: low memory consumption, low computing time, and high accuracy. Only raising the number of GPUs unlocks the available memory for DNNs and the computing power, while ensembles allow to efficiently exploit them.

We will keep in mind for the remaining of the thesis that ensemble methods leverage naturally to the multi-GPUs computing nodes and are more computing efficient at scale. The remaining of this thesis are significant contributions to the designing of ensemble workflows from hyperparameter search to inference. The next chapter deals with a systematic procedure to build ensembles of supervised deep neural networks, chapter 7 the procedure to build ensembles of reinforcement learning agents, and chapter 8 the inference system for the ensembles.

6 Hyperparameter optimization with ensembling

Deep Neural networks (DNNs) are notoriously difficult to tune, train, and ensemble to achieve state-of-the-art results. Hyperparameter optimization with ensembling or "HPO+ensembling" tools provides a simple interface to train and evaluate many ensembles of DNN in parallel aiming at high accuracy and low overfitting. We propose a helpful method for machine learning practitioners to run those different tasks automatically by efficiently using the underlying hardware.

Nowadays, multiple researchers and practitioners have well understood the benefit of homogeneous ensembles of DNNs to reduce overfitting [149] [184] [58] [155]. Then, authors [97] [46] propose heterogeneous ensembles to not only reduce variance but attempt to cancel out their different biases too at the cost of a more complex procedure. Further, several winners and top performers on challenges routinely use ensembles to improve accuracy [53]. However, ensembles of DNNs suffer from three main limitations to be widely deployed in research and industrial applications.

The first limitation is a lack of understanding of which is the best way to build a library of models. How to generate models to build an ensemble of DNNs is still not fully understood [11] [148] but authors generally agreed [46] that the diversity of predictions, individual performance, and numerous DNNs are the three key components. However, we still ignore in which proportion they are important a priori. This is why complete ensemble construction workflows share in common they build multiple ensembles and return the best one on a validation dataset and are detailed in the “related works” section 6.1.

The second limitation is a lack of control of the computing cost of the produced ensemble. In the previous works, authors apply post-hoc ensembling [14] [42] of non-deep machine learning which are efficiently handled by multi-core CPU or GPU [20]. They propose the number of models to put in the ensemble as a threshold between accuracy and computing cost. Those methods require some adaptation to be applied to the deep learning case due to two main differences:

- Each base deep neural network training is a computing-intensive GPU algorithm taking several minutes or hours
- Different base neural networks sampled may have orders of magnitude of resources

requirement (e.g., different number of convolutions, different number of filters per convolution ...).

That is why the number of models constraint used previously is not relevant to controlling the ensemble of deep neural network computational cost. And more, when generating automatically a library of models it is a known [73] no clear correlation between accuracy and model cost, meaning sometimes fast deep neural network can be prioritized without significantly lowering the accuracy.

The third limitation, no inference server enables an efficient inference of heterogenous DNNs by optimizing the exploitation of multiple GPUs. Current inference server allows deploying deep neural networks [175] [119] user of those state-of-art servers attach manually DNNs to GPUs and set the batch size. Ensemble of DNNs is much more complex because we must deal with multiple DNNs sometimes co-localized into the same device and multiple devices. An ideal inference server of DNNs ensemble would allow computing automatically the best localization and batch size settings at the initialization phase. Due to the novelty and the importance of those works for future deep learning applications we detail this contribution in the next chapter 7.

For instance, it is time to address the missing piece in deep learning pipelines between hyperparameter optimization and ensembles. To summarize our contributions, we run extended benchmarks with seven algorithms to compare how best to build the library of DNNs and conclude that asynchronous Hyperband [91] suits this goal. After the library is generated, we propose a new simple algorithm named SMOB (scalarized-multi objective with budget greedy) to build an efficient ensemble based on their accuracy and a desired maximum computing cost. We measured here the computing as the time to predict 2000 images, this metric may be easily replaced by another one such as its power consumption or the memory consumption. Figure 6.1 compare the three workflows: Hyperparameter optimization (HPO), ensembles of machine learning models, and our proposed HPO+Ensemble workflow. We presented this novel workflow as a novel HPC application in the scientific conference HPC ASIA 2021 [124].

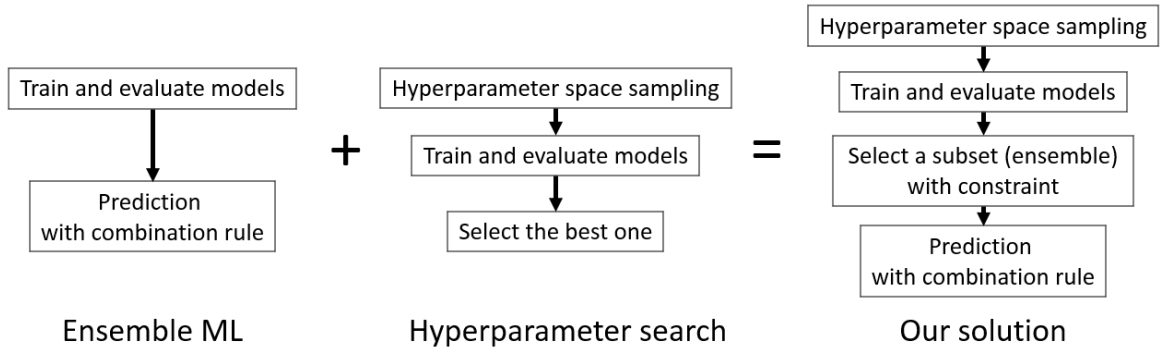

Figure 6.1 – *Simple comparison of the different workflows*

6.1 Related works

AutoML which not only selects the best model but combines them is valuable for domains that require the best possible accuracy. Several benchmarks were performed and AutoML+ensembling is today considered as the big AutoML challenge series winner on data points like AutoML challenges [53] and Kaggle challenges on image recognition. Previous research on non-deep machine learning ensembles showed that over-fitted machine learning algorithms predictions can be averaged out to get more accurate results [149]. This phenomenon is mainly explained by the Law of Large Numbers which claims that the average of the results obtained from many non-biased trials should be close to the expected value. These results are especially interesting for deep learning models. They are the most affected models to random effects (over-fitting) due to their huge number of parameters.

We observe two main trends in AutoML with ensembling:

- **HPO+Ad-hoc ensembling** [160] [87] [49] [18] which consists in searching directly ensembles. The hyperparameter space describes an ensemble of fixed size. In the end, the HPO algorithm keeps the best ensemble and wastes all the others. For example, an ensemble of 2 DNNs with 2 hyperparameters per model, is represented with a size 4 vector.
- **AutoML+Post-hoc ensembling** [14] [162] [42] runs a standard HPO algorithm providing a library of trained models, then constructs ensembles from the library based on a greedy algorithm.

Figure 6.2 summarize 3 different AutoML+Ensemble procedures by vizualizing for each one three possible ensembles sampled.

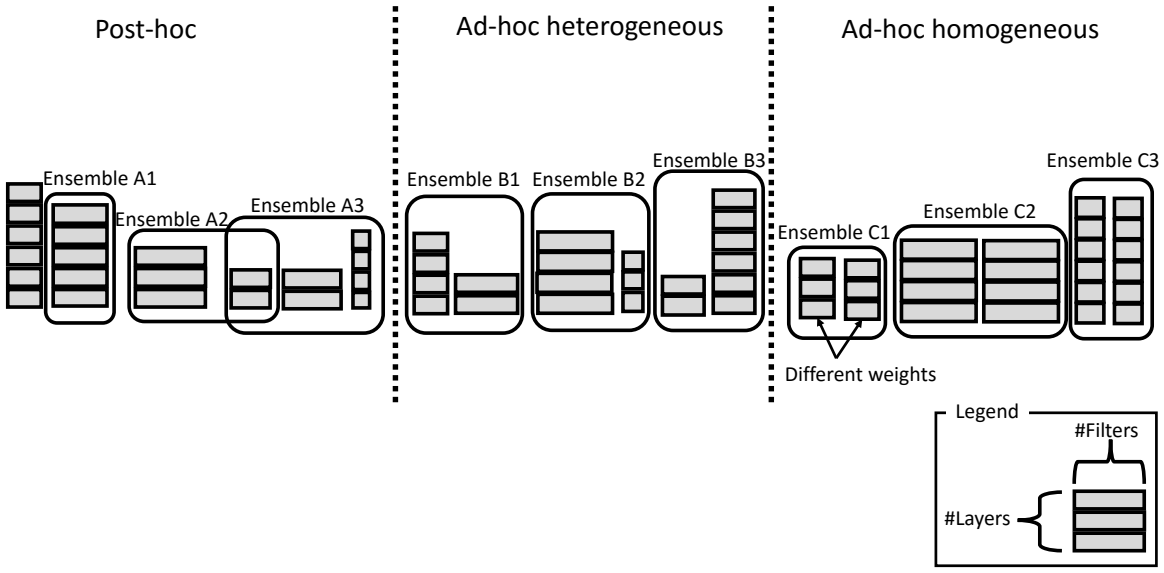

Figure 6.2 – *Three automatic ensemble selection procedure illustrated. Each grey stack represents neural networks with two different hyperparameters: depth and width.*

HPO+Post-hoc ensembling of supervised machine learning has 2 advantages compared HPO+Ad-hoc ensembling: it is more flexible and may evaluate fantastically more ensembles in a few minutes. However, HPO+Post-hoc is not well applicable to reinforcement learning compared to HPO+Ad-hoc ensembling. Those claims are in-depth discussed below.

More flexibility. HPO+Ad-hoc ensembling suffers from a lack of flexibility because the number of models in an ensemble is fixed before building all DNNs. In other words, changing the number of DNNs in the ensembles requires a new HPO runtime. Second, *AutoML+Post-hoc ensembling* [14] [162] [42] runs a standard HPO algorithm providing a library of trained models, then constructs ensembles from the library based on a greedy algorithm. This approach is flexible because we can produce several ensembles with different numbers of models with one library of DNNs.

More ensembles assessed. The posthoc ensembling allows evaluating of more ensembles than ad-hoc because each DNNs model can be used for multiple ensembles. The number of ad-hoc models produced is given with the formula $\lceil \frac{L}{S} \rceil$ with S the desired ensembles size and L the library size. In comparison, the number of possible combinations (without replacement) of posthoc ensembling is the known combination formula given by $\frac{L!}{S!*(L-S)!}$. Let's put those numbers into perspective. Let's imagine we have 100 DNNs and we want an ensemble of size 10. The post-hoc procedure may build 1.7E+13 ensembles against only 10 with the ad-hoc procedure.

Post-hoc procedures and ad-hoc procedures may not always be fairly compared by counting the number of the assessed ensemble when an exploitative hyperparameter optimizer is used. In this case, ad-hoc exploitative optimization will evaluate the next ensemble according to previous ensembles, while post-hoc exploitative optimization will evaluate the next DNN according to the previous DNNs.

However, the exploratory optimizers have not shown performance against exploratory ones in hyperparameter optimization [90], this claim is reinforced by the No Free Lunch Theorem [169] saying no optimizer may outperform another in all cases.

Not applicable efficiently to the reinforcement learning case. In RL applications, the DNN requires interactions with the environment during the validation phase and not fixed data samples. So, validation predictions cannot be cached on disk and thus, thousands of combinations cannot be evaluated quickly. This is why we propose another workflow in chapter .

6.2 Intuition behind complementarity

From homogeneity to heterogeneity. It is well known that diverse machine learning algorithms [97] [46] such as a decision tree voting with a neural network may perform better than homogeneous ensembles of the best base algorithm at different runtimes.

Motivated by those results, we make further analyses to understand how heterogeneous ensembles of DNNs behave. In particular, we are interested to answer if diverse architectures are correlated with diverse DNN predictions.

To make this analysis, we trained 22 DNNs, with two optimization settings and diverse architectures. For example, ResNet18 and ResNet50 with Glorot are very correlated, like VGG16 and VGG19 with Glorot. They are easily reproducible analyses using on-the-shelf deep neural networks. Results are presented in figure 6.3.

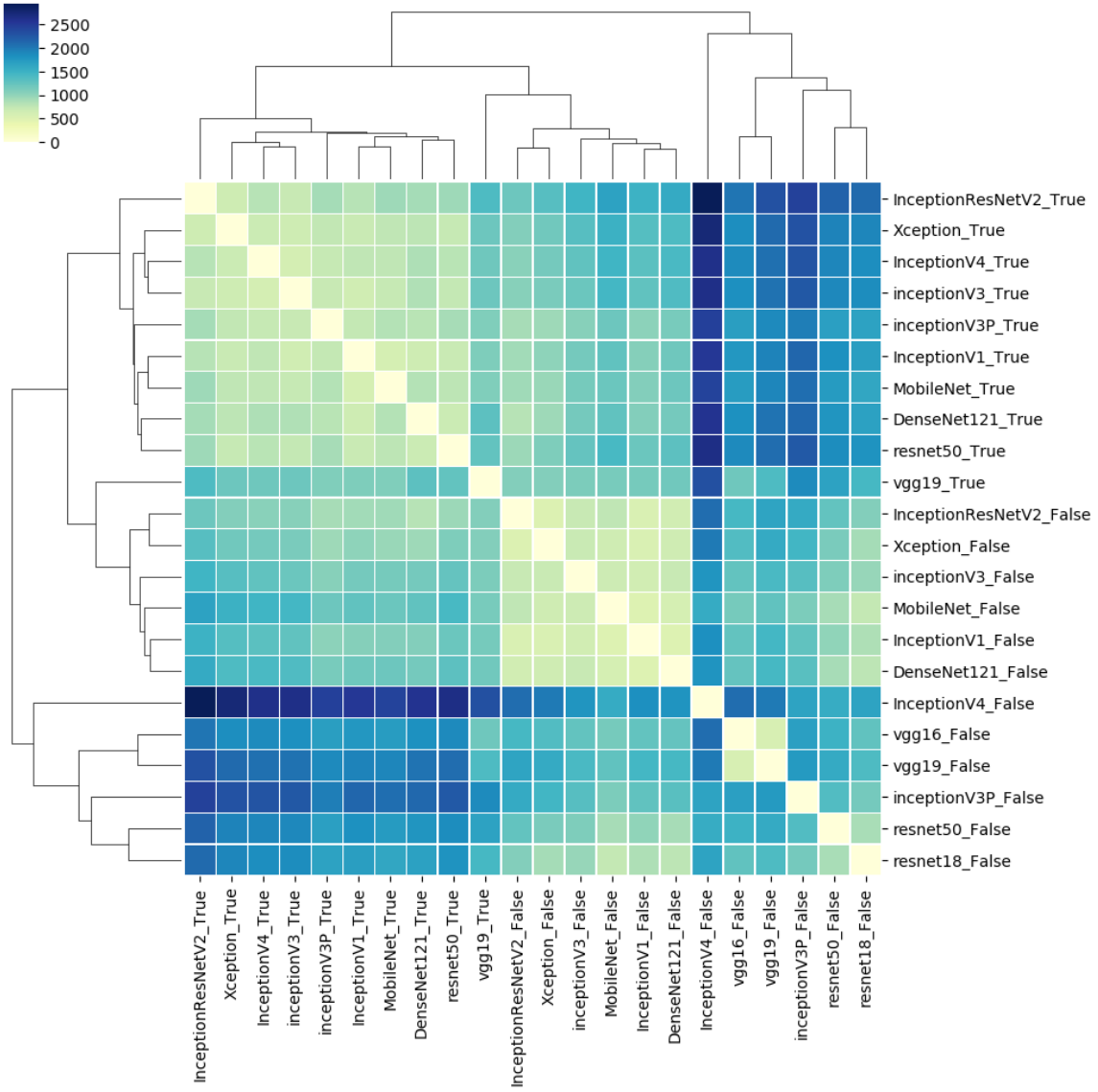

Figure 6.3 – *Distance matrix between predictions of 22 models on the microfossils test dataset containing 6400 images and trained on 25600 images. To measure the diversity, each element is the sum of the Euclidean distance between predictions. Those model names correspond to their architectures and initializers ("True" for pretrained with Imagenet and "false" for Glorout random initialization [48]). InceptionV3P is a homemade architecture based on InceptionV3 [155] with Dropout [151] and one additional dense layer. A binary hierarchy clustering (dendrogram) is used to sort and aggregate them for readability purposes.*

We discover that diverse DNNs produce diverse predictions. In other words, similar architecture and initializer often produce close models' predictions, and the opposite is true. For example, the same architecture InceptionResnetV2 with Glorot initialization and pre-trained with Imagenet [136] are quite distant and yet produce similar scores (respectively

Fscore=89.89% and Fscore=89.92%). Furthermore, we can observe same architecture styles produce more correlated DNNs: Inception architectures, VGG architectures, and ResNet architectures. Surpringlisly, varying the initialization seems to have more impact than varying the neural architecture.

We exploit those results to build our workflow and benefit from this diversity to build better ensembles. We confirm in section 6.7.1 that a heterogeneous ensemble built with an HPO library outperforms a homogeneous ensemble of a robust hand-crafted model such as ResNet50, ResNet152, ...

From diversity to complementarity. Previous authors discover that different machine learning algorithms produce different error distributions [46]. Even if it is well established there is a connection between base DNNs diversity and ensemble accuracy, the base DNNs must be selected with the "right" diversity [79]. Indeed, the measure of diversity itself is a subject of debate, and finding the most accurate subset of DNNs based on their individual error distribution is a non-trivial problem. Indeed, an ideal ensemble should be made of complementary neural networks which cancel out their error when they predict together.

To illustrate the effect of diversity and the complementarity here a quick example. Let's imagine this simple scenario:

- Classifier A outputs the probability vector [0.5,0.1,0.4] answering the first class (with a 50% probability).
- Classifier B outputs the probability vector [0.1,0.5,0.4] answering the second class (with a 50% probability).
- The ensemble (average) outputs the probability vector [0.3,0.3,0.4] answering the third class (with a probability 40%).

This simple demonstration shows that even if we interpolate predictions, the ensemble may provide an original prediction compared to its base components. In other words, even though base machine learning algorithms are wrong, they may be complementarity and return the right answers. Of course, it is also possible models disagreeing produce an incorrect ensemble. For example, in section 4.2, we demonstrated that systematically maximizing the diversity of predictions with Negative Correlation Learning (NCL) [102] is not ideal to build good ensembles because either it hurts individual accuracy or it fails to get the "right" diversity (diverse but not complementary).

There is no systematic procedure to know in advance which models are complementary but to compute their combination and see how well the ensemble is accurate. This is why it is necessary to evaluate thousands of combinations between classifiers to find complementary ones. This challenge belongs to combinatorial optimization problems.

Computational cost of ensembles. The computational cost is an important notion when ensembling deep neural networks. Therefore it is important to be able to measure it to control the final ensemble computing cost. Very simplistic measures such as the number

of parameters or the FLOPS fail to measure accurately the current deep neural network speed because they have evolved in complex computing graphs [130].

Precise estimate speed of an ensemble with our inference system described in chapter 8 is a priori impossible. Measuring the speed of an ensemble with our system takes a long time: to host it, distribute its computations, and benchmark it. This is why, to measure the ensemble computational cost, we rather propose to sum the forward phase speed to predict 2000 images of all DNNs contained in the ensemble. This fast heuristic allows evaluating and comparing thousands and millions of ensembles in a few minutes.

Build efficient ensembles.

To summarize, we now know that to build efficient ensembles 4 key ingredients are required:

- Maximize base DNNs accuracy
- Maximize the complementarity among DNNs
- Maximize the number of DNNs in the ensemble
- Minimize the computational cost

Optimizing all those characteristics is very difficult, and we ignore which important they are important for a given task. And more, optimizing one may impact the other ones. This is why our strategy consists in evaluating a thousand ensembles empirically and selecting the most suitable one.

The procedure to build the best ensembles is an open problem. It requires answering multiple non-trivial problems: how to build the best library of models? How to find complementary models? How control the computing cost of the ensemble? Section 6.3 proposes a novel workflow as a systematic answer to all those questions. The next sections will confirm empirically that this workflow suits the need to build efficient ensembles.

6.3 Proposed workflow

In this section, we give an overview of the proposed workflow 6.4. Then, we will go into further details on the distributed architecture that accelerates these three steps of our AutoML workflow: HPO to construct a library of models, Ensemble Selection to construct ensembles based on their validation accuracy, and computing cost expressed here as seconds to predict 2000 images.

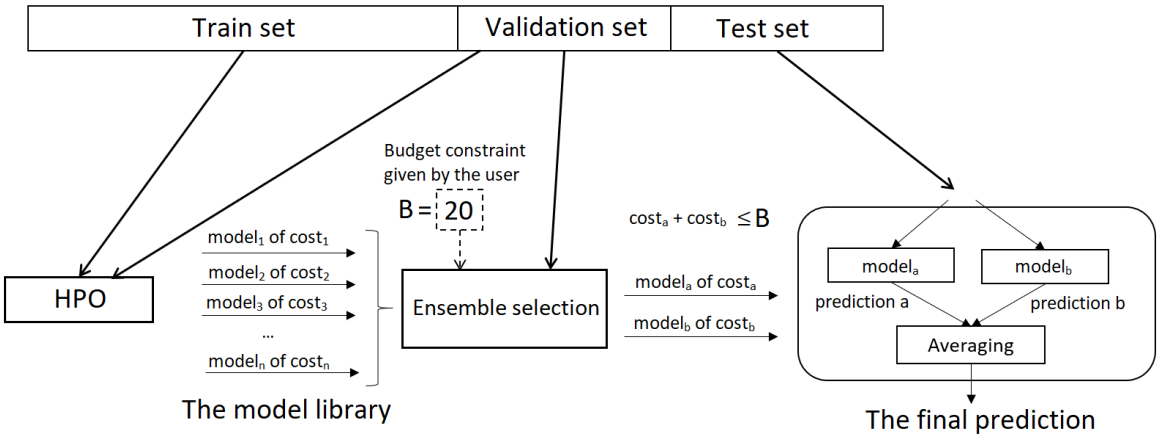

Figure 6.4 – *The proposed workflow runs 3 steps 1) The HPO algorithm generates a library of models. The trials are distributed on several GPUs synchronized by a master process. We recommend asynchronous Hyperband based on experimental results. 2) We propose a new multi-objective Ensemble Selection algorithm to search for the most efficient ensemble honoring a budget given by the user (here B=20). It is based on a parallel greedy algorithm evaluating hundreds of combinations per second on multi-core CPUs. 3) The returned ensemble predicts by combining (averaging) DNNs predictions on new data.*

6.3.1 Detail of the workflow

Hyperparameter optimization (HPO). The choice of the HPO algorithm has several impacts in terms of the number of produced models at a given time horizon, the accuracy of models, and their diversity. After several experiments (section 6.4.2), we recommend asynchronous Hyperband [91] [93] based on our experiments. The main concept of Hyperband consists to train in embarrassingly parallel by keeping all GPUs occupied.

To enter in more detail, asynchronous Hyperband is the fruit of 3 inventions:

- **SHA** (Successive Halving) [70]. SHA is initialized by randomly selecting hyperparameters as a random search but periodically prunes low-performing models. Therefore SHA focuses the computational resources on the more promising experiments.
- **Asynchronous SHA** (ASHA) [91]. One drawback of SHA is that the experiments need to be frequently synchronized to gather validation scores and prune the experiments. ASHA improves the SHA's resource utilization by removing the need to synchronously evaluate and prune low-performing models.
- **Hyperband** [93]. The pitfall of SHA and ASHA is that the pruning rate is sensitive to tune to get good performance. If it is too low, the resources explore the low number of hyperparameters but the pruning is performed on well-trained neural networks. If it is too high, we sweep a large number of hyperparameters but the pruning is based on low trained neural networks. To be more robust, Hyperband consists in invoking SHA or ASHA multiple times with varying levels of pruning rates.

In this regard, Hyperband produces the biggest library of models among tested algorithms because most models do not reach the maximum number of epochs. It is also a

lock-free distributed algorithm until the termination of the algorithm, spending most of its runtime to train models occupying all GPUs and storing them in the library. Then, Hyperband produced also very diverse models due to the initial random sampling of hyperparameters which is desirable in terms of final accuracy. Finally, because more training iterations are given to the best models, Hyperband performs explore-exploit in the time (not in the hyperparameter space) and we observe it builds a better distribution of individual DNNs (regarding the top 10 % and top 25 %) compared to random search on all our runtimes. But we cannot expect Hyperband to outperform or underperforms other exploratory-exploitative HPOs in all cases such as Bayesian or genetic algorithms [169]. A large number of models, hyperparametric diversity, and efficiency are three reasons explaining why Hyberband is robust to build a library of models but we still ignore in which proportion they are important.

Ensemble selection. An ensemble selection algorithm finds the best combination possible honoring the budget given by the user. We propose SMOBF greedy standing for "Scalarized Multi-Objective with Budget Forward greedy".

Forward greedy [14] starts with the empty ensemble and successively adds the best available model to improve the ensemble target metric. The algorithm stops when no available model can improve accuracy or respect the budget.

First methods optimize neural network hyperparameters, not only based on accuracy but a linear combination of accuracy and inference speed [120]. In the case of ensemble constructions, we propose the equation 6.1 to inform the greedy algorithm to favor accurate and cheap DNNs before consuming the overall budget. The current ensemble is a with its computing cost C_a and its predictions y_a. Penalty P_a returns 0 when the budget B is honored ($C_a \leq B$) or an arbitrary large number when it is not to exclude a from the possible ensembles.

The weight w allows controlling the nature of the solution found by the greedy algorithm by placing greater or lesser emphasis on the objectives. The greedy algorithm is run multiple times with different values $w = 0.1$, $w = 0.01$, and $w = 0.001$. Scalarization is a convenient way to handle multi-objective problems by reducing them to a single objective problem, so a simple mono-objective optimization can be performed.

$$score_a = (1 - w) * E(y_a, y) + w * C_a + P_a \tag{6.1}$$

Then, the best ensemble is picked according to its validation cross-entropy loss and respecting the given budget. To improve the robustness of the method on new applications C_a is standardized and it is the rate between the sum of the computing cost of base models (time to predict 2000 images) and the budget B.

The ensemble selection algorithm computes the validation loss of candidate ensembles to evaluate how well a solution will generalize on the test database. Since the data used for validating is taken away from training the individual models, keeping the validating set small is important. Smaller validating sets are however easier to overfit. Contrary to

common AutoML on data points datasets, due to the cost of training and evaluating one model on images datasets, we do not repeat the experience with K-fold cross-validation.

In the library of models, some models diverge or have such poor performance compared to other models that they are unlikely to be useful to improve any ensemble building [13]. Eliminating these models from the library should not reduce the performance and facilitates the ensemble selection task by decreasing the number of non-promising combinations. Pruning works as follows: models are sorted by their performance on the validation metric and only the top X% of them are used for ensemble selection. After pruning, the predictions on the validation set are cached before running the ensemble selection algorithm. This allows handling only predictions vector and not models during the ensemble selection process.

Ensemble combiner rule. We use the simple average as a combiner rule. More advanced methods exist such as "ensemble selection with replacement" [14], weighted averaging, and stacking. Those methods are calibrated on a validation set and thus prone to over-fitting.

Now that the workflow is developed, the final accuracy depends on two settings. The tuning time of the library of models and the budget given by the user to generate a new ensemble to deploy.

6.3.2 Distributed HPO with GPU clusters

To assess large numbers of hyperparameter trials in a reasonable amount of time, a distributed framework using one GPU cluster or several GPU clusters is required. Several trials are distributed with a middleware containing one scheduler and multiple workers, each worker being associated with heavy computing resources. Typically, the scheduler sends a hyperparameter set to the available workers with the number of training epochs to perform. Then, it gathers the scores of the trials from the workers and finally computes the next hyperparameters to send and so on.

This distributed framework suits the need of the majority of optimizers by leveraging GPUs and clusters. Each trial is run independently without slowdown among themselves. Workers can connect or disconnect to the master at run time; however, a disconnection interrupts the current trial. Many HPO algorithms benefit from this middleware such as random search, Hyperband, parallel SMBO, evolutionary algorithms, and also some greedy algorithms for discrete optimization such as SMOBF.

Modern clusters have evolved into a hybrid machine that contains both CPUs and GPUs on each node. These heterogeneous CPU-GPU clusters are particularly useful to accelerate the training loop of one trial. One trial is implemented with two processes, the *preprocessor* which loads and preprocesses the data on multi-cores, while the *trainer* trains the DNNs on the GPU. In case the entire training dataset fits into memory, the preprocessor does not need to load multiple times the data. This is with the data samples are shared between preprocessors of the cluster to avoid copies and useless I/O.

6.3.3 Distributed Ensemble Selection

Ensemble selection is a greedy algorithm evaluating all neighbors of a current ensemble at each iteration. For each ensemble a, the equation 6.1 is performed. This procedure takes one second on one CPU with 100 elements per prediction (100 classes), 2000 validation data samples, and a relaxed budget.

In the case of semantic segmentation images, the class predictions are at pixel scale making this procedure much more computing-intensive. A typical dataset for autonomous vehicle applications [25] contains 256x256 input images, 30 classes per pixel and 500 validation samples. Linear scaling indicates that the computing cost of this procedure would take 1h20 (256x256x30x500 predictions elements compared to 100x2000 predictions elements in CIFAR100 taking 1 second). Because SMOBF is an optimization algorithm we may use a similar distributed framework (previous section) to distribute neighbors evaluation on CPUs or GPUs to alleviate this computing cost.

6.4 Experimental data set and hyperparameter settings

We experiment and discuss our workflow by varying the three steps of the HPO+ensemble workflow on CIFAR100 and the microfossils datasets.

6.4.1 The infrastructure

Experiments were done on IBM Power9 architecture, containing 2 sockets. In each socket, there are 18 cores of CPU with a maximum frequency of 3.8Ghz and 256G of RAM. There are also 6 GPUs by node and are Nvidia Tesla V100-SXM2 with 16G of memory. Hyperparameter optimization framework Tune [98] was used. It runs above the Ray framework [110], it schedules and spreads experiments to run on GPUs and stores results into files. Deep Learning training loop and data augmentation was coded with the framework Keras [51] with TensorFlow 1.14.0 [1] backend.

6.4.2 Hyperparameter configuration space

Table 6.1 shows all hyperparameters properties in this workflow. We use ResNet-based architectures due to their simplicity to yield promising and robust models on many datasets. We explore different residual block versions: "V1", "V2" [58] and "next" [173]. Regarding the optimization method, we use Adam optimizer [76] due to its well-known performance and its lower learning rate tuning requirement.

The simplicity of the ResNet architecture makes this work easy to test by a scientist on its image dataset [181]. Exploring other convolutional block type like VGG [184], Xception [23], DenseNet [66] and EfficientNet [158] are potential improvement which could increase the degree of liberty in DNNs construction and improve again the accuracy of ensembles.

Category	Name	Type	Range
Optim.	Learning rate	Continuous	[0.001; 0.01]
	Batch size	Discrete	[8; 48]
	L2 regularization factor	Continuous	[0; 0.1]
NN archi.	Convolution type	Categorical	{v1,v2,next}
	Activation function	Categorical	{tanh,relu,elu}
	Number of filters in the first convolutional layer	Discrete	[32; 128]
	Multiplier of filters in the 4 blocks	Discrete	[32; 128]
	Number of convolutional block in the first stage	Discrete	[1;11]
	Number of convolutional block in the second stage	Discrete	[1;11]
	Number of convolutional block in the third stage	Discrete	[1;11]
	Number of convolutional block in the fourth stage	Discrete	[1;11]
Data augment.	Max zoom	Continuous	[0; 0.6]
	Max translation	Continuous	[0; 0.6]
	Max shearing	Continuous	[0; 0.3]
	Max channel shifting	Continuous	[0; 0.3]
	Max rotation measured in degrees	Discrete	[0; 90]

Table 6.1 – *The hyperparameter space experimented based on the ResNet neural architecture framework*

The CIFAR100 dataset contains 32x32 images while usually ResNet is adapted to be used on ImageNet (224x244 images). Those different resolutions need some adaptation. Therefore, in the CIFAR100 case, the first convolutional network is replaced from the 7x7 kernel size with a stride of 2, to a 3x3 kernel size with a stride of 1. With equivalent settings, our CNN framework produces nearly the same number of weights between the CIFAR100 and microfossils dataset, but the time complexity is factor 11 different because of different signal resolutions flowing through the layers.

6.4.3 The two datasets used

The CIFAR100 dataset. CIFAR100 [77] consists to 60,000 32x32 RGB images in 100 classes. For each class, there are 580 training images, 20 validation images and 100 testing images.

The Microfossils dataset. Microfossils are extremely useful in age dating, correlation, and paleo-environmental reconstruction to refine our knowledge of geology. Microfossil species are identified and counted on large microscope images and thanks to their frequencies we can compute the date of sedimentary rocks.

To do reliable statistics, a big number of objects need to be identified. That is why we need deep learning to automate this work. Today, thousands of fields of view (microscopy imagery) need to be shot for 1 rock sample. In each field of view, there are hundreds of objects to identify. Among these objects, there are non-fossils (crystals, rock grains, etc...) and others are fossils that we are looking for to study rocks.

Our dataset contains 91 classes of 224x224 RGB images (after homemade preprocessing). Microfossils are calcareous objects taken with polarized light microscopy. The classes are imbalanced, we have from 50 images to 2500 images by class, with a total of 32K images in all the datasets. The train/validation/test split is as following: 72% 8% 20%. The F1 score was used and labeled as 'accuracy' on all benchmarks.

6.5 Experiments and results

We experiment and discuss our workflow by varying the three steps of the AutoML workflow of deep neural networks on CIFAR100 and the microfossils datasets. More details is given in appendix 7.3.

6.5.1 The infrastructure

Experiments were done on IBM Power9 architecture, containing 2 sockets. In each socket, there are 18 cores of CPU with a maximum frequency of 3.8Ghz and 256G of RAM. There are also 6 GPUs by node and are Nvidia Tesla V100-SXM2 with 16G of memory. Hyperparameter optimization framework Tune [98] was used. It runs above the Ray framework [110], it schedules and spreads experiments to run on GPUs and stores results into files. Deep Learning training loop and data augmentation was coded with the framework Keras [51] with TensorFlow 1.14.0 [1] backend.

	9 trials 100 epochs					30 trials 30 epochs				
	acc (%)	duration	speed up	#epochs	gpu (%)	acc (%)	duration	speed up	#epochs	gpu (%)
RS 6GPUs	67.08 ± 3.85	10h13	6.2	900	85	64.90+-0.92	11h40	5.5	900	82
HB 6GPUs	62.58 ± 4.73	5h48	4.4	710	49	62.94 ± 5.28	1h40	5.2	388	62
HB 4nodes*6GPUs		2h00	8.8	710	33		0h48	10.9	388	34
GA 6GPUs	61.85 ± 2.55	14h29	2.8	900	27	65.33 ± 2.33	10h01	4.1	900	57

Table 6.2 – *Limited scale benchmarks of some HPO algorithms. Hyperband was set up with a halving of 3 and the genetic number of trials is divided into 3 successive generations. The columns from the left to the right are: The accuracy error which is not dependent on the number of resources, the duration of the HPO process, the speed up measured compared to the single GPU version, the #epochs is the number of train iteration performed, the mean percentage of occupied GPU during all the entire HPO process.*

6.5.2 step 1 - HPO

Comparison of hardware allocation

Our CNN framework based on ResNet can take between 15 seconds and 11min30 regarding the complexity of the neural architecture assessed (number of filters per convolution, number of convolutional blocks, ...). Our benchmark 6.2 reveals that Random Search and Hyperband occupy more the GPUs than algorithms based on a sequence of population algorithms (such as GA) which is explained by the fact that no intermediate stopping of all GPUs is required. Those benchmarks also confirm that Hyperband terminates earlier than Random Search because the less promising DNNs have been stopped before the maximum number of epochs is reached. The scalability of HPO algorithms and saving useless computations are two important algorithmic characteristics to explore a large number quantity of DNNs.
At the end of any HPO algorithm, the last few DNNs free the GPUs but they take various amounts of time to terminate, this explains why we do measure not a perfect usage of GPUs of Random Search and Hyperband. Hyperband emphasizes this phenomenon because the early stopping increases the variability of time taken between DNNs.

HPO to produce ensembles

We compare in table 6.3 our presented workflow by varying the HPO algorithm to generate a different library of models. Random Search (RS) [6], asynchronous Hyperband (HB) [91], Bayesian Optimization with Gaussian Processes (BOGP) [63], BOHB, Sequential Model-based Algorithm Configuration (SMAC) [69], Tree Parzen Estimator (TPE) [5], Genetic Algorithm (GA). To combine generated models, we benchmark them with the SMOBF greedy ensemble selection algorithm. We benchmark three times each algorithm and we show the median value of each computing. When the budget is relaxed we observe the standard deviation is generally lower than 0.1%. When the budget is lower the standard deviation is lower than 1.5%.

Not surprisingly, when the budget increases the ensemble selection can find better ensembles from any library. This is explained by the fact that the number of available good

Data	Budget	RS	HB	BOGP	BOHB	SMAC	TPE	GA
C100	20	31.26	27.8	31.58	29.24	**24.44**	29.97	24.61
	40	24.24	**22.87**	28.2	25.6	22.97	26.01	22.91
	60	22.27	21.85	26.91	23.53	22.65	25.44	**21.51**
	80	22.69	21.73	26.2	22.58	22.17	24.25	**21.07**
	120	21.63	**21.55**	24.78	21.86	22.17	23.03	21.95
	160	**20.11**	**21.11**	24.24	21.64	21.87	22.47	21.91
	240	20.7	**20.46**	23.76	21.2	21.87	22.55	21.91
	320	20.64	**20.42**	23.76	21.1	21.87	22.46	21.91
Micro	125	13.45	11.93	14.44	12.18	**10.27**	13.91	10.33
	250	10.98	9.82	10.93	11.71	**9.63**	10.66	9.71
	375	10.84	**9.32**	9.93	10.5	9.45	10.49	9.5
	500	10.23	**8.91**	9.93	9.84	9.27	9.06	9.28
	750	9.7	**8.87**	9.21	9.28	9.18	9.3	9.18
	1000	9.69	**8.46**	8.77	8.8	9.09	9.21	9.07

Table 6.3 – *Our workflow error (%) by comparing seven HPO algorithms was run on both datasets for 6 days on 6 GPUs. Each HPO generates a library of models for a given dataset. The budget is expressed as "sum of DNNS times (seconds) to predict 2K images on 1 GPU"*

combinations between them increases. In the table 6.3 some algorithms converge and do not find better ensembles after high budgets such as the Bayesian method and genetics ones. Different run time shows a similar conclusion. On both datasets, the best ensemble is found with Hyperband. 79.44% accurate on CIFAR100 and 91.54% accurate on microfossils. We do not show results of AutoML without ensembling compared to AutoML+ensembling due to a lack of space, but AutoML+ensembling is Pareto dominant for all budgets and all performed run times.

Effect of tuning time

In AutoML we often want to see the evolution of performance over the tuning time, not only the final performance after a time horizon. Therefore, we assess the workflow by varying the budgets until it does not affect the ensemble construction anymore and at different tuning time snapshots. Those figures are presented in figure 6.5 6.6.

First, regarding the benefit of the tuning time, we observe two main trends. When the budget is very small (B=20) model accuracy converges early to 18, 24, or 48 hours reaching the limit of the hyperparameter optimization with a little (or without) ensembling. When the budget is bigger the exploding number of available combinations of ensembles leads to the discovery of better ensembles. The post-hoc ensembling is a promising line of research that deserves more attention and more understanding of how DNNs interact with each other.

Then, we observe that increasing the budget systematically leads to an increasing accuracy (colored lines are rarely crossed) but this trend decline. For example, in the figure 6.5 we show that when Hyperband is finished, the benefit is obvious from 1 to 2

models: +5 point of accuracy percentage, but the improvement is small from B=120 to B=320: +1 point of accuracy percentage.

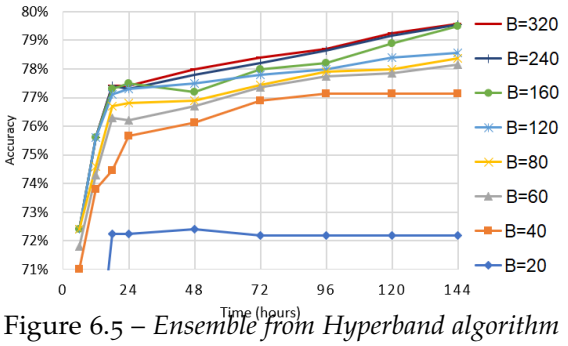

Figure 6.5 – *Ensemble from Hyperband algorithm on the CIFAR100 dataset*

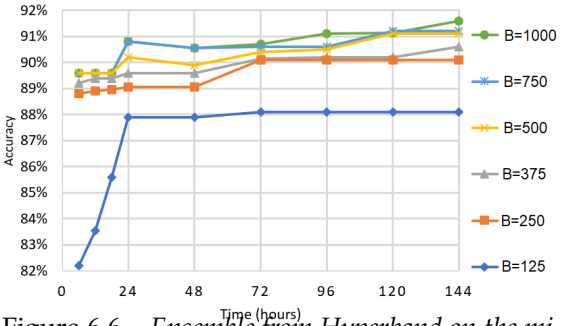

Figure 6.6 – *Ensemble from Hyperband on the microfossils dataset*

Visualization of architecture sampling results

Usually, researchers searching neural architectures look at only prediction quality as the target metric. This is an oversimplification of the problem converging towards either very diverse or hudge neural networks. When a library of models is generated this is crucial to take the best one not only on its prediction quality but as a second criterion to reflect computational requirements. In this chapter, our objective is accuracy under a computing time constraint.

When multiple architectures are evaluated such as a different number of layers, and a different number of units per layer, some neural networks may be one order of magnitude slower than others. Furthermore, there is no or low correlation between the accuracy and computing cost of models generated.

We visualize libraries of models generated with two HPO algorithms, SMAC and Hyperband. We display libraries generated on the two datasets in the figure 6.7 and 6.8. Each point is a model in function of two objectives error rate (horizontally) and its measured computing cost (vertically).

Those 3 phenomena: diverse accuracy, diverse computational cost, and no correlation teach us that extracting a Pareto front is important and reinforces the need for multi-objective optimization. This is even more critical when multiple DNNs are gathered to build ensembles. SMOBF is designed to sample the Pareto front in the space of potential ensembles using scalarization as a sampling strategy.

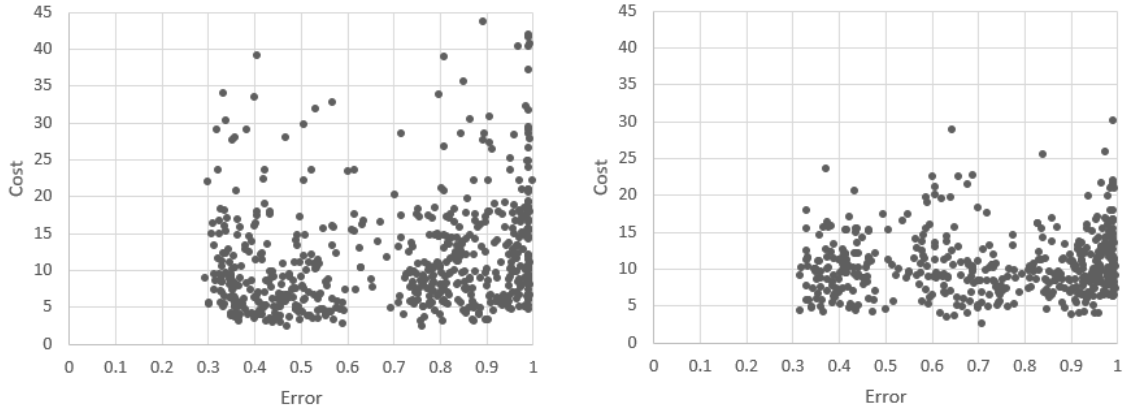

Figure 6.7 – *Correlation between the validation cross entropy (horizontally) and their inference to predict on 2000 images with 32 batch size (vertically). All black points are sampled models on* ***CIFAR100****. Left library produced with asynchronous Hyperband and right SMAC [69]*

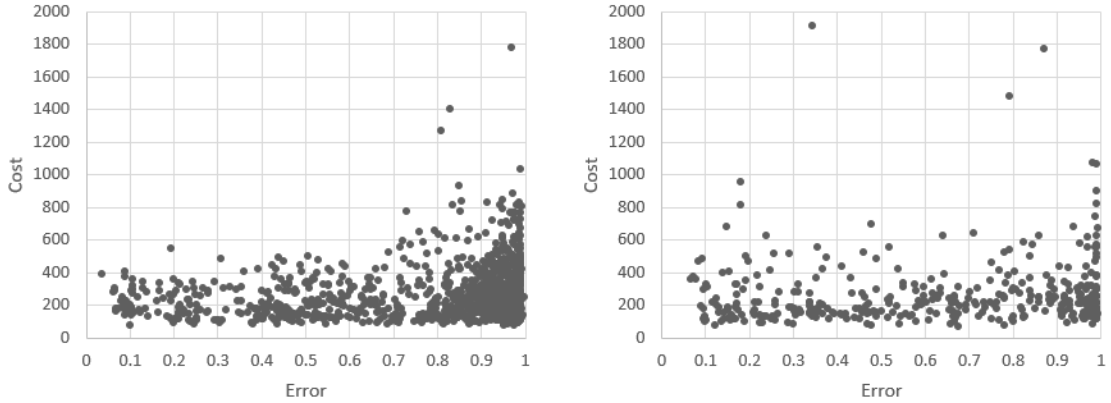

Figure 6.8 – *Correlation computing cost versus accuracy of randomly sampled models on* ***Microfossils****. Left library produced with asynchronous Hyperband and right SMAC [69]*

6.5.3 step 2 - Ensemble Selection

Ensemble selection pruning

Some base neural networks stored in the library of models are so inaccurate they have no chance to be selected by the Ensemble Selection algorithm. We try multiple pruning factors X such that we select only X% of the top X% neural networks stored in the disk-based on the validation score. It reduces the size of the library of models and thus helps the ensemble selection algorithm. When the pruning factor is above 20% it does not reduce accuracy and reduces the ensemble selection time, while the threshold under 15% reduces the target metric in some experiments. For all experiments, the pruning factor is set to 20%.

We attempt to design a similar threshold for computing cost but it shows no benefit. When the budget is relaxed, the most computing expensive model has the potential to be selected and to improve the ensemble's accuracy.

Ensemble selection methods under budget

Because ensemble selection methods was a few empirically explored [162] we propose SMOBF and compare it to multiple baselines. Methods are compared by varying the budget on two libraries of models, one generated with SMAC (table 6.4) and the other one with Hyperband (table 6.5), both trained with the CIFAR100 datasets.

Naive. We evaluate a naive method consists in sorting all models in function of their error. Then naively select best models first until the budget is exhausted. Because it is known that interaction between models are complex and the cost is not considered during the model selection, this method generally produced sub optimal solutions. It illustrates that the ensemble selection is a non trivial problem.

Standard optimizers. Random Search, SMAC (Sequential Model-based Algorithm Configuration [69]), TPE (Tree Parzen Estimator [5]), GA (Genetic algorithm). We formulate the ensemble selection as a function taking a binary vector as input B such as B_i is 1 if the DNN i is put in the ensemble or 0 if i is not. It returns the target metrics s as output or an arbitrary large number if the budget is violated. We apply and compare those optimized to minimize the function $f(B) \rightarrow s$.

Those algorithms are tuned (e.g. numbers of samples, number of generations, ...) to get best results. To be fairly compared they generate ensemble until a fixed time horizon of $\approx$500 seconds. We also evaluate RS, SMAC, TPE and GA during more than one hour but we do not observe measurable benefits. Random Sampling was set by increasing the probability to disable a model regarding the mean cost of models $\hat{c}$, the budget B and the number of models n. Random Sampling have a probability of add a model in the ensemble of $\frac{B}{\hat{c}*n}$ as a rule of thumbs.

Forward with Budget greedy (FB greedy) [14]. The ensemble S is initialized empty. Until no available model reduce the error F, the available model v decreasing the most the formula $F(S \cup \{v\})$ is added to S. A model is said available if it was not previously added and if adding it does not violate the budget. This is similar to [14] but the stopping criteria is not a fixed number of DNNs but the budget.

Backward with Budget greedy (BB greedy). [80] At the opposite of FB greedy, the ensemble S is initialized full. Then models are eliminated one-by-one to minimize F until the budget is respected and no model elimination would minimize again F.

There is a clear evidence of the superiority of the greedy methods. Between FB greedy and BB greedy no one is clearly more accurate than the other one. We observe that FB is better on 4 metrics, BB is greater on 5 ones, and they are tied on 3 ones. FB greedy was widely faster because its empty initial state leads to very fast solution finding when number of models in the solution is small compared to the usage of the entire library as initialization.

SMOFB greedy beats FB greedy on 6 metrics and is beaten 2 times by FB greedy on the error accuracy metrics. Even though SMOFB greedy is 3 times slower (because 3 values

	Budget=20		Budget=80		Budget=320		
Name	ACC	CE	ACC	CE	ACC	CE	Time (sec.)
Naive	33.12	1.4486	26.15	1.0216	23.14	0.8949	< 1
FB greedy	28.2	1.1362	22.02	0.8668	**20.42**	0.7919	33;105;284
BB greedy	29.23	1.1439	22.16	0.8658	20.53	**0.7915**	1311;1298;1447
SMOFB greedy	**27.8**	**1.1316**	**21.73**	**0.8573**	**20.42**	0.7919	99;248;870
RS	30.27	1.1641	23.53	0.9316	22.48	0.8444	
SMAC	28.48	1.1724	23.97	0.9715	22.23	0.849	500±40
TPE	28.58	1.1994	22.32	0.8679	21.45	0.8244	
GA	28.05	1.1553	23.2	0.9194	21.94	0.84	

Table 6.4 – *Comparison of ensemble selection algorithms from a library of models generated with Hyperband on CIFAR100. According different budget (B) and the two metrics: error rate (%) and the cross-entropy validation loss. To a fair comparison RS, SMAC, TPE, and GA have been set to be run with 500 seconds time horizon. All solutions produced honor the corresponding budget B. Greedy algorithms take different amount of time in function of the library size.*

	Budget=20		Budget=80		Budget=320		
Name	ACC	CE	ACC	CE	ACC	CE	Time (sec.)
Naive	**23.95**	**0.8929**	24.78	0.9573	27.08	1.1263	< 1
FB greedy	24.44	0.9458	22.65	0.8679	**22.06**	0.8445	47;94;151
BB greedy	25.9	1.0138	22.65	**0.8543**	**22.06**	0.8445	645;627;566
SMOFB greedy	24.72	0.9402	**22.63**	0.8591	22.12	**0.8397**	138;291;515
RS	28.78	1.1362	23.46	0.9110	23.07	0.8856	
SMAC	25.74	1.059	23.21	0.9062	23.07	0.8739	500±40
TPE	24.68	0.9607	22.82	0.8871	22.7	0.8614	
GA	25.05	0.979	22.97	0.9090	22.68	0.8694	

Table 6.5 – *Comparison of ensemble selection algorithms from a model library generated with SMAC on CIFAR100*

of w are assessed) it is still faster than BB greedy and in addition, ensembles it found outperforms FB greedy on the cross-entropy error.

Some other methods may be invented to automatically identify complementary and fast ensembles with a reasonable algorithmic complexity. For example, unsupervised approach have the potential to discover the most diversified neural network predictions and prune the most redundant neural networks (such as used to compute the dendrogram in figure 6.3).

SMOBF greedy compared to Forward greedy (baseline)

The forward greedy algorithm is the most commonly used ensemble selection algorithm [14] [42], this is what we fix as the baseline. We perform three runtimes of the overall ensemble construction workflow and observe that SMOBF (Scalarized Multi-Objective with budget forward) is Pareto dominant or equivalent to the baseline each time.

The figures 6.9, 6.10 compares those two algorithms in the function of the cost (vertically) and the error rate (horizontally) of produced ensembles with SMOBF greedy (blue) and the baseline (orange) on one runtime. In figures, the mention "BX" means a budget of X and "#Y" means an ensemble of size Y.

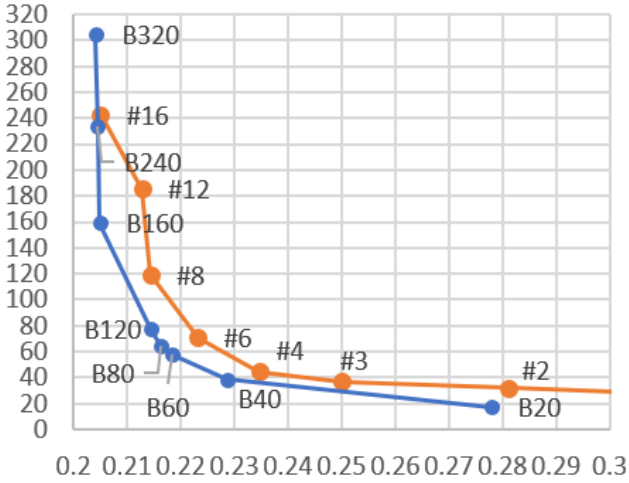

Figure 6.9 – *Ensembles generated from Hyperband algorithm on the CIFAR100 dataset*

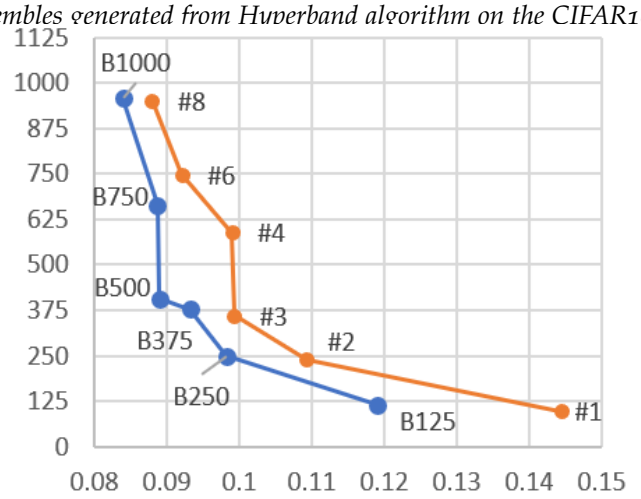

Figure 6.10 – *Ensembles generated from Hyperband algorithm on the microfossils dataset*

The gain of SMOBF is particularly obvious when the budget is small on both criteria, but it is reduced when the budget is relaxed. Indeed, when we target the best ensemble at any cost or any size, the objective of the two algorithms converges. On the opposite when the budget is small, SMOBF informs the greedy algorithm to go toward efficient models before the budget is consumed.

Assessing ensemble combiners

During the ensemble selection process, we try three ways to best combine a candidate ensemble: majority voting, averaging, and weighted averaging. Results are summarized on the table 6.8 and 6.9.

Budget	20		80		320	
Metric	Error	CE	Error	CE	Error	CE
Maj. Voting	**0.2643**	2.8047	0.2261	1.9208	0.2197	1.3622
Averaging	0.278	**1.1316**	0.2202	**0.8668**	**0.2042**	**0.7919**
W. Averaging	0.2808	1.1362	**0.2199**	0.8691	0.2092	0.7921

Table 6.6 – *Forward greedy ensemble selection algorithm over the population of models trained with hyperband. We try three combiner rules: majority voting, averaging, and weighted averaging. We evaluate them on the cross-entropy error ("CE") and the accuracy error ("Error") on the test dataset.*

Budget	20		80		320	
Metric	Error	CE	Error	CE	Error	CE
Maj. Voting	0.2663	2.5460	0.2381	1.6657	0.2314	1.1860
Averaging	**0.2444**	**0.9458**	0.2265	0.8679	0.2206	0.8445
W. Averaging	0.2450	0.9461	**0.2258**	**0.8647**	**0.2204**	**0.8414**

Table 6.7 – *Forward greed ensemble selection algorithm over the population of models trained with SMAC. We try three combiner rules: majority voting, averaging and weighted averaging.*

Majority voting. It counts the votes of all the predicted labels from the models and it predicts the label with the most votes. Compared to simple averaging, it suffers from the loss of information, as it only uses the predicted label.

Averaging. The deep learning literature has already suggested [155],[184] [58] that averaging is a reasonable choice for homogeneous ensembles of models and therefore with similar accuracy. However, the averaging combiner rule can suffer from a major drawback. In an ensemble where base models have diverse performance, the weaker models in the ensemble contribute at the same importance as stronger ones and may intuitively hurt the performance.

Weighted averaging based on ensemble selection with replacement. We experiment with weighted averaging using ensemble selection with replacement, the number of times a model is taken by SMOBF in the ensemble corresponds to its weight in the ensemble [14] [162]. Then, weights are normalized to ensure that the sum is equal to 1. We observe no benefits to using weighted averaging, we may interpret it by the fact validation data information is already used by SMOBF, and calibrating weighted averaging is not beneficial. Of course, we could extract data samples from the training set to create a second validation set, but it would reduce the number of data samples for the training phase, and would increase the complexity of the procedure (e.g., how many data samples are required ?).

Machine learning model as combination rule. A meta-learner such as stacking [170]) is a model taking all base predictions as input and returns the final ensemble prediction. The meta-learner is expected to combine the strengths of all the base learners. We do not evaluate meta-learner here due to multiple reasons. First, the meta-learner is generally calibrated on a validation split and we observe that a very simple weighted averaging combination rule (previous paragraph) already overfits it. Second, it would increase

Budget	20		80		320	
Metric	ACC	CE	ACC	CE	ACC	CE
Maj. Voting	**0.2643**	2.8047	0.2261	1.9208	0.2197	1.3622
Averaging	0.278	**1.1316**	0.2202	**0.8668**	**0.2042**	**0.7919**
W. Averaging	0.2808	1.1362	**0.2199**	0.8691	0.2092	0.7921

Table 6.8 – *Forward greedy ensemble selection algorithm over the population of models trained with hyperband. We try three combiner rules: majority voting, averaging and weighted averaging.*

Budget	20		80		320	
Metric	ACC	CE	ACC	CE	ACC	CE
Maj. Voting	0.2663	2.5460	0.2381	1.6657	0.2314	1.1860
Averaging	**0.2444**	**0.9458**	0.2265	0.8679	0.2206	0.8445
W. Averaging	0.2450	0.9461	**0.2258**	**0.8647**	**0.2204**	**0.8414**

Table 6.9 – *Forward greed ensemble selection algorithm over the population of models trained with SMAC. We try three combiner rules: majority voting, averaging and weighted averaging.*

the ensemble selection running time that requires assessing thousands of combinations, training meta-learner for each combination is not possible. Third, a machine learning model as a combination rule would also significantly hurt the time in the inference phase, especially on the latency metrics.

To conclude, it appears that naively averaging predictions is simple and performs as well as weighted averaging them with no risk to overfit the validation dataset. The weighted averaging combiner performs very similarly to the averaging combiner. We can explain this result by the fact that the validation information is already used during the ensemble construction, so the weights tuned on this validation set are over-fitted.

6.6 Baseline comparison

We report in figure 6.10 results of authors running until convergence multiple AutoML methods with diverse theoretical backgrounds [89], [129], [176]. We run the Auto-Keras framework version 1.0.12 with one single GPU and default settings, except the max_trials set to 10. We notice no improvement when max_trials is set to 20. We report our proposed workflow results and some intermediate scores over time.

The comparison between AutoML methods is known to be difficult because authors explore different search spaces, with different initial conditions, different data processing, and ensembles, however, we observe two main trends. The first block of rows AutoML methods converging fast in several GPU hours which often explore the neural architecture space sequentially on one GPU. The second block is evolution-based algorithms that converge later, their population-based approach allows to easily leverage of GPU clusters. Asynchronous Hyperband is still not widely used.

Method	#GPU	hours	Cum h	GPU name	#DNNs	Test(%)
RSPS [92]	1	2	2	GTX1080TI	1	52.31±5.77
DARTS-V1 [101]	1	3	3	GTX1080TI	1	15.03±0.00
DARTS-V2	1	10	10	GTX1080TI	1	15.03±0.00
GDAS [35]	1	9	9	GTX1080TI	1	71.34±0.34
SETN [34]	1	10	10	GTX1080TI	1	58.86±0.06
ENAS [123]	1	4	4	GTX1080TI	1	13.37±2.35
Auto-Keras [72]	1	2	2	Tesla V100	1	69.57±0.53
LSE [129]	250	264	66K	?	?	77.0
CNN-GA [176]	3	320	960	GTX1080TI	1	77.97
CNN-GA+cutout [176]	3	320	960	GTX1080TI	1	79.47
Ours with B=320	6	6	36	Tesla V100	4	72.39
Ours with B=320	6	24	144	Tesla V100	18	77.35
Ours with B=320	6	144	864	Tesla V100	34	79.44

Table 6.10 – *The comparison between AutoML algorithms in terms of the classification accuracy (%) and GPU hours on CIFAR100 benchmark dataset. The "?" mention means the information is missing in the paper. Column from left to right are: the name of the method, the number of GPUs used, duration of the algorithm (hours), cumulated time, GPU name, number of models (1=no ensembling) , mean test accuracy*

As evolutionary methods, our method can benefit from high computing power. In comparison, asynchronous Hyperband does not require intermediate stopping of all GPUs which makes it more suitable to use GPU clusters compared to a sequence of generations. Also, in the large-scale case where the number of available GPUs is superior to the number of trials, Hyperband can run all trials at the same time, while genetics is limited by running all trials of the current generation.

Furthermore, Hyperband has low settings requirements. It uses early stopping which reduces the sensibility to the #trials/#epochs dilemma. Also, the exploration/exploitation of the hyperparameter space is balanced by stopping a fraction of less promising trials, so only the halving factor is needed. Evolutionary methods require many more initial settings such as the number of generations, the crossover operation, and the mutation operation. They make this algorithm sensitive to the initial choices and so the settings calibrated for an application cannot be suited for a new application.

We try our best to do a fair comparison by using the same dataset. However, different data processing, different hardware, and different initial settings can have an important impact on experimental results. We do not perform cutout and yet it seems effective processing on CIFAR100. Finally, we recall that with the given time horizon of 36GPU/hours visible in figure 6.9 and figure 6.10, Hyperband+SMOB greedy has not yet converged.

6.7 Replace the HPO tuning

Our proposed workflow is made of a few modules. We attempt to replace the construction of DNNs based on Hyperband with different technics: homogeneous ResNet architectures

ran at different timesteps and on the shelf library of population models such as InceptionV3, VGG16,...

6.7.1 Replace HPO with homogeneous ensemblings

Retraining the same deep learning architecture from scratch can yield significant distance in different run times, that is why we compare different HPO strategies and different popular ResNet architectures.

Table 6.11 show the workflow error by replacing the library from the HPO algorithm with a library from a fixed ResNet architecture. For a fair comparison with the previous benchmark, libraries have been generated at different run times for 6 days and 6 GPUs.

CIFAR100 dataset						
ResNet archicture	R18	R34	R50	RX50	R101	R152
#weights per DNN	31.6M	53.1M	53.1M	13.1M	95.8M	128.4M
Cost per DNN	10.83	14.35	14.05	11.87	21.14	28.56
#1	**35.30**	35.77	37.63	42.28	35.85	37.82
#2	32.98	**21.70**	33.60	38.67	32.77	34.86
#3	29.88	**28.58**	31.29	36.16	30.30	32.16
#4	28.66	**27.63**	29.76	35.09	29.74	31.65
#6	27.70	**26.88**	28.16	32.15	28.93	30.26
#8	26.98	**26.21**	27.33	31.86	28.26	29.33
#12	26.59	**25.88**	26.82	31.44	71.85	29.46
#16	26.66	**25.74**	26.80	31.44	28.15	29.45
The microfossils dataset						
ResNet archi.	R18	R34	R50	RX50	R101	R152
#weights of 1	31.6M	53.1M	53.1M	13.1M	95.8M	128.4M
Cost per DNN	120.53	169.71	164.96	118.64	254.45	354.29
#1	**13.17**	14.76	14.35	15.09	15.78	14.91
#2	12.26	**12.23**	12.46	12.74	13.96	13.39
#3	11.78	**11.45**	11.52	12.07	12.46	12.34
#4	11.66	**11.29**	11.37	11.98	12.26	12.09
#6	10.90	**10.50**	10.73	11.73	11.76	11.32
#8	10.87	**10.41**	10.61	11.27	11.60	11.20

Table 6.11 – *Homogenous ensembling accuracy based on ResNet architectures. We vary the dataset, the ResNet architectures to build the library of models, and the ensemble size. We stop assessing to add models when it plateaus. Values are expressed as error percentage on the test dataset.*

ResNet architecture is robust on-the-shelf neural architecture, but producing homogeneous ensembles underperform the ensembles built with an HPO library. This may be explained by the fact homogeneous hyperparameters create similar predictions, this lack of diversity makes it impossible to find complementary models. This reinforces our previous

claim (section 6.2) that diversity is a necessary condition to find complementary ensembles (but not always sufficient).

6.7.2 Replace HPO with a library of on the shelf models

SMOBF may be widely applied to any library and supervised applications. To confirm the performance of this idea, we run the SMOBF algorithm on the shelf deep neural networks trained on ImageNet from the "keras.applications" Python package [1]. SMOBF may also generalize to any prediction quality measure (e.g., accuracy, top5 accuracy, ...) and computing cost (e.g., number of parameters, power consumption, memory consumption, inference speed ...)

Experimental conditions Catalogue of architectures and their accuracy of models on Imagenet are showed in figure 6.12.

[1] https://keras.io/api/applications/

Deep learning model	Inference speed (speed)	# parameters (M)	Top 1 acc.	Top 5 acc.
VGG16 [184]	3.1851	138.4	0.6957	0.8907
VGG19	3.4092	143.7	0.6980	0.8903
ResNet50 [58]	2.9540	25.6	0.7428	0.9165
ResNet101	4.0294	44.7	0.7546	0.9242
ResNet152	5.0209	60.4	0.7592	0.9267
ResNet50V2	1.9967	25.6	0.6936	0.8911
ResNet101V2	3.5429	44.7	0.7155	0.9039
ResNet152V2	4.8311	60.4	0.7226	0.9071
NASNetLarge [187]	17.3257	88.9	0.8249	0.9596
NASNetMobile	1.8671	5.3	0.7357	0.9140
Xception [23]	5.4912	22.9	0.7883	0.9438
InceptionV3 [155]	2.8771	23.9	0.7765	0.9379
InceptionResnetV2 [156]	7.1731	55.9	0.8030	0.9513
DenseNet121 [66]	2.0806	8.1	0.7435	0.9192
DenseNet169	2.2796	14.3	0.7559	0.9273
DenseNet201	2.8423	20.2	0.7657	0.9320
MobileNet	2.1173	4.3	0.7077	0.8967
MobileNetV2 [137]	2.1996	3.5	0.7205	0.9063
MobileNetV3S	1.5649	2.5	0.6725	0.8740
MobileNetV3L	2.1847	5.4	0.7501	0.9216
EfficientNetB0 [158]	3.0769	5.3	0.7353	0.9165
EfficientNetB1	2.4584	7.9	0.7610	0.9295
EfficientNetB2	3.3445	9.2	0.7724	0.9359
EfficientNetB3	4.5246	12.3	0.7931	0.9478
EfficientNetB4	12.0068	19.5	0.8097	0.9547
EfficientNetB5	22.0013	30.6	0.8201	0.9591
EfficientNetB6	36.7563	43.3	0.8278	0.9637
EfficientNetB7	61.9721	66.7	0.8239	0.9619
EfficientNetB0V2 [159]	1.2126	7.2	0.7802	0.9387
EfficientNetB1V2	2.0405	8.2	0.7926	0.9458
EfficientNetB2V2	2.5375	10.2	0.7995	0.9493
EfficientNetB3V2	3.6709	14.5	0.8143	0.9552
EfficientNetV2S	7.8051	21.6	0.8368	0.9655
EfficientNetV2M	19.3997	54.4	0.8493	0.9715
EfficientNetV2L	32.0848	119	0.8537	0.9721
Ensemble A	27.2048	76	0.8517	0.9725
Ensemble B	78.7326	288.2	0.8589	0.9742
Ensemble C	113.5	240.1	0.8584	0.9746

Table 6.12 – *Library of deep learning models. Ensemble A contains {EffV2M, EffVS} ; Ensemble B contains {EffV2L,EffV2S,EffV2M,NASLarge, Mobile} ; Ensemble C {EffV2L, EffV2M, EffB7}*

Ensemble A and Ensemble C are produced aiming for" top 5 accuracy" under inference time budget. Ensemble B is produced aiming "top 5 accuracy" under the number of parameters.

Once an ensemble is built, the weighted averaging combination rule instead of averaging

improves slightly the top 5 accuracy (up to +0.1% accuracy gain). In this case, due to the quantity of validation data available (25K samples), it seems weighted averaging does not overfit this dataset split.

Experimental results We vary the budget of SMOBF in figure 6.11 taking the catalogue presented in figure 6.12. We SMOBF procedure with two different computing costs: inference time and the number of parameters, showing our ensemble selection SMOBF algorithm is flexible.

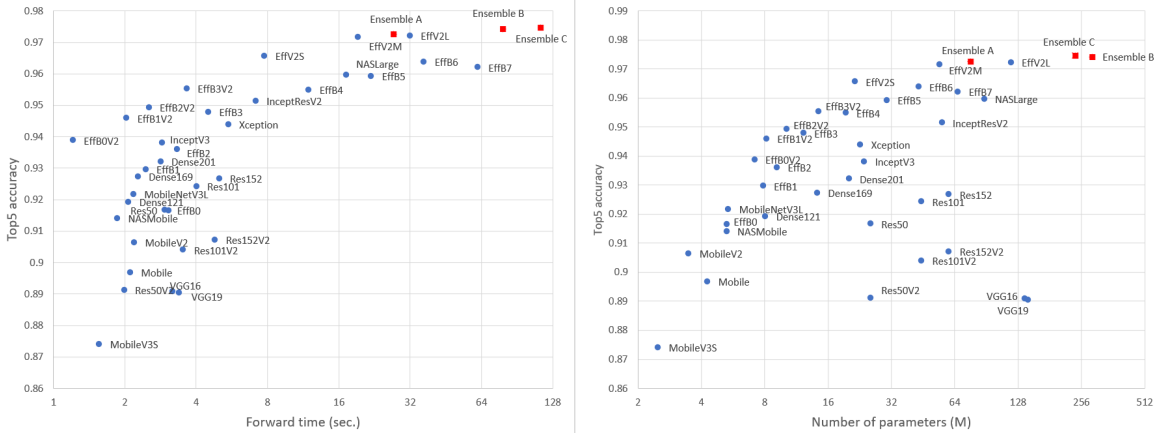

Figure 6.11 – *SMOBF taking as input on the shelf deep neural networks applied on Imagenet dataset with different budgets. Imagenet contains 1000 classes, for each class 1,300 images of training, 25 images of validation, and 25 images of test. We also showed performance between simply averaging predictions and stacking the ensemble. For each experiment, the number of models in the Ensemble is indicated. We observe small prediction quality improvement to use the stacking combination rule (orange) instead of the averaging one (blue).*

The first major observation is that ensembles are generally Pareto superior to the most efficient neural network of the library. This is a major discovery that shows the benefits of ensembling worth their computing cost when they are combined with multi-objective combinatorial optimization. And more, SMOBF is general enough to find a more efficient solution on any library of models and success in discovering ensembles Pareto superior compared to the state-of-the-art deep neural networks. By performing those experiments, we also observe that SMOBF is flexible enough to return only one single model of the library (i.e., an ensemble of size 1).

6.8 Summary

Due to the increasing number of new Deep Learning applications and datasets, Auto Machine Learning (AutoML) methods are an important line of research. We propose a workflow capable to calibrate hyperparameters, training multiple neural networks at the same time, and extracting an ensemble of DNNs that is a subset of the library of trained DNNs. We aim to fill the gap between Machine Learning research, the new GPU clusters, and the end-user application quality of service. To go toward this direction, we formulate

the problems by aiming at the accuracy, the ensemble inference speed, and the flexibility of the underlying heterogeneous infrastructure.

For the first time, we do not assess the hyperparameter optimizer to find the best neural network, but to generate the best library of models to then combine them. We presented the experimental results demonstrating that asynchronous Hyperband is suitable for generating a large number of diversified models thanks to 3 inherent characteristics. First, the uniform sampling of the hyperparameter space allows for sampling statistically diverse hyperparameters. Second, efficient asynchronous computing to keep GPUs occupied by the workload. Third, the early stopping mechanism purses both goals: reducing the overall tuning time and maintaining the most promising neural networks trained.

We then propose a novel Ensemble Selection strategy, SMOBF, that optimizes the ensemble selection by controlling the final ensemble computing cost of heterogenous DNNs. When the budget is relaxed, our algorithm offers high and robust accuracy compared to other AutoML workflows. We observe that SMOBF generally found Pareto optimum ensembles compared to the individual neural network in its library of models.

All those researches show that efficient hyperparameter optimization should be mixed with ensemble construction to find efficient ensembles. The next chapter will present hyperparameter optimization to compute ensembles in a very different context: the reinforcement learning case. Chapter 8 deals with deploying those produced ensembles in inference mode by leveraging modern computing nodes.

Reinforcement Learning and ensemble

7

In the previous chapter we propose a novel workflow to integrate all the steps of the automatic construction of ensembles for supervised deep learning applications. As seen in chapter 3 we see the emergence of energy applications formulated as Markov Decision Process leveraging reinforcement learning [109].

Reinforcement learning cannot fully benefits from the proposed workflow of HPO+Ensemble in the last chapter. The nature of the problem to solve and some previous choices applied to the supervised context cannot be generalized in the RL context for two reasons:

First **it is not possible to assess thousands of potential ensembles** in a reasonable amount of time. In the supervised case, we could store all DNNs predictions of the validation data samples, it allows us the ability to assess potential ensembles on a multi-core CPU in a reasonable amount of time. This hypothesis is not possible in the case of reinforcement learning where future states/predictions/rewards will depend on the ensembles' interactions with an environment. For example, to evaluate 1000 ensembles of 4 bases agents on GPUs each one taking 1 hour to be trained, it would require 4000 GPU hours.

Secondly, in the previous chapter, we use the Hyperband algorithm to prioritize the most promising experiments during the training time according to the validation score. However, RL algorithms are much more unstable than supervised ones, so comparing the learning curves is not informative due to their high volatility. The instability of the training at different runtimes may also ask our-self if the homogeneous ensemble procedure is already diverse enough base agents compared to the more complex heterogeneous ensemble procedure.

The reproducibility is also a desired characteristic of major machine learning algorithms. To maintain scientific progress in ML and its applications, it is required to get confidence in the reproducibility of the results of the experiments and be able to fairly compare different algorithms. Reinforcement learning methods are especially sensitive to random effects and hyperparameters tuning [59] making them difficult to use in practice. This is why to

compare the predictions of different methods it is mandatory to compare them using the mean score and the spread of the score at different runtimes.

To alleviate this, we propose and compare multiple workflows of ensembles in the reinforcement learning case. Ensemble learning is now well-known for producing qualitative predictions by using the instability of the same learners at different run times [149]. In the context of RL, it has the potential to improve the robustness and quality of reinforcement learning trajectories by taking a better sequence of local better decisions.

We discovered that an ensemble of RL agents overpasses the simple baseline consisting to select the best single agent in it. And more, ensembles are generally worth the computing cost on recent hardware without systematically requiring GPU investment. Their independent nature allows us to run them in embarrassingly parallel on multi-GPUs computing nodes and be efficiently deployed in the inference phase too.

In the last few years, some ensembles of reinforcement learning have been proposed and demonstrate good abilities [36] [81] to increase the diversity of samples gathered to improve the stability and to accelerate the training process compared to one single agent. However, there are still open research questions to be investigated: which procedure is better to build them? Does ensemble worth the computing cost compared to selecting the best model in it?

In this chapter, we aim to answer them. We will not limit ourselves to comparing the training dynamic of one ensemble compared to one single agent. We instead sweep hundreds of hyperparameters to fairly compare ensembles by splitting training/validation and test phases to reflect the need to measure how well they generalize. Furthermore, for the first time, we compare multiple ensemble procedures: homogeneous [149] and heterogeneous [97] [11] ensemble construction procedures. And different combination rules, averaging[149], weighted averaging [55], and select only the best base agent (i.e., no ensemble). Then, we evaluate the computing cost of the building phase and the inference phase running on modern computing nodes, and we discuss whether the ensemble is worth the computing cost with a suited parallel implementation.

This chapter first demonstrates experimental evidence that homogeneous ensembles with averaging as a combination rule are more performant and more stable than the other evaluated procedures. Second, we perform extended experiments by increasing the number of agents and conclude that it significantly improves the stabilization of the cumulative reward without widely increasing the computation time of modern GPU computing nodes. Indeed, GPUs have evolved into chipsets with thousands of cores that efficiently run multiple small neural networks in parallel. Finally, due to the simplicity of the proposed procedure and the stabilization effects, our experiments are easily reproducible and adapted can be applied beyond our assessed environments and RL frameworks.

We experiment with diverse ensembles of RL agents constructions on two industrial applications linked to the electricity control: Pymgrid [60] and GEM [161]. Both are available with a continuous and discrete action space. We evaluate ensemble workflows on both RL algorithms: DQN for discrete environments [56] [167] and DDPG [174] for

continuous ones. However, those results may not generalize well in some RL frameworks where RL agents lack exploration in the training phase, the interpolation of too many specialized agents may drive the ensemble in some trajectories where agents have low experiences.

Our chapter follows this structure: In section **??** we show the progress in what we call "collective performance": multi-agent, multi-module agent, multi-threaded agent. We discuss what is beneficial to ensembles and what they bring to those fields. In (**??**) we introduce the workflow we use to perform ensemble benchmarks. In (7.3) and (7.4) We perform multiple benchmarks to assess different ensemble construction procedures in terms of test score and stability of the test score.

7.1 Collective performance in reinforcement learning

This section is about collective performance in the context of reinforcement learning: multi-agents, agents made of multiple modules, and distributed agents to run faster the training. For each pattern, we will compare them with an ensemble of reinforcement learning, explicit the difference and how they can be mixed with the ensembles.

7.1.1 Multi-agents and multi-modules agent

Multi-Agent Reinforcement learning (or "MARL") was mentioned multiple times in literature with different variants. In general, multiple agents [12] in the same environment perceive different states and act at the same time. On the opposite, in the ensemble of agents, the decisions are made collectively by all agents perceiving the same state. In other words, in MARL each step is as follows: n partially observed states by n agents predicting and returning n actions. In the ensemble of agents, there is 1 state observed by n agents and returning 1 final action. MARL and Ensemble do not aim to solve the same problem but some authors [71] succeeded to apply them together. MARL allows partitioning of the input space and the decision space to simplify the DNNs tasks, and combines multiple DNNs on each partition of the environment allows to improve the predictions but those works lack in-depth analysis of the ensemble procedure evaluation.

Reinforcement learning agents may be broken down into sub-modules to be more efficient. Some architectures are separately trained modules such as actor-critic neural networks [153] [108] and hybrid model-based and model-free [111]. On the opposite, some architectures are neural network trained end-to-end with specialized streams such as one auto-encoder stream and LSTM stream [116] or dueling DQN splitting advantages stream and values stream [167]. Our proposed ensemble benefits from all those works, for instance, we ensemble multiple DDPG agents each one being actor-critic methods and we do the same with dueling DQN agents.

7.1.2 Distributed agents

Distributed agents training [113] [108] [39] have been proposed to train faster RL agents and use efficiently the underlying parallelism of the computing nodes. The training of one agent may be break down in 4 sub-program parts:

- (A) The parameter server containing the agent parameters.
- (B) The rollout worker collecting experiences. It performs iteratively the forward of the neural network and the next step in the environment.
- (C) The replay memory storing thousands of experiences for off-policy RL and only a batch of data samples for the on-policy RL. One experience is a transition $\{s\_t, a\_t, r\_t + 1, s\_t + 1\}$ with s_t the current state and a_t the action performed on this state, $r\_t + 1$ the reward and $s\_t + 1$ the new state after the action a_t.
- (D) The learner performing updates of the neural network based on the replay memory.

Previous works proposed diverse mappings between multiple processes and those 4 roles. They are considerable flexibility in the number of ways the RL training loop may be parallelized. We give below a non-exhaustive list:

- In GORILA [113], there are Multiple (A), multiple (B) on multiple servers called "shards", multiple (C) implementing asynchronous SGD, and multiple (D).
- In A3C [108] there are 1 process (called "master") implementing (A) and (C); no (B); Multiple (D).
- In Apex [64], there is 1 process (called "learner") containing (A), (B), (C); Multiple (D), we will name below the "the rollout workers".
- In IMPALA [39] there are multiple learners, each one containing (A), (B), (C), and multiple (D).

Different strategies are balancing between computing parallelism versus communication complexity. Some of them use synchronism with consistency [39] versus asynchronism with outdated information [109] which has already been discussed in the supervised case. Those works share in common that they use a pool of rollout workers to accelerate the overall training loop which is generally the main bottleneck. However, the communication overhead may not be worth the time saved by interacting in parallel. A separated process (B) needs to communicate regularly its experiments (C) and be synchronized with the global parameters (A).

We assess rollout workers in figure 7.1 7.2 and observe that the parallelism allows us to divide the computing time quite easily on modern devices but it is not ideal due to communication overhead between the learner process and the rollout worker. We also

observe that according to the neural network architecture the GPU is not systematically faster than the CPU. It shows that more in-depth analysis must be done to accelerate the hyperparameters sweeping workload.

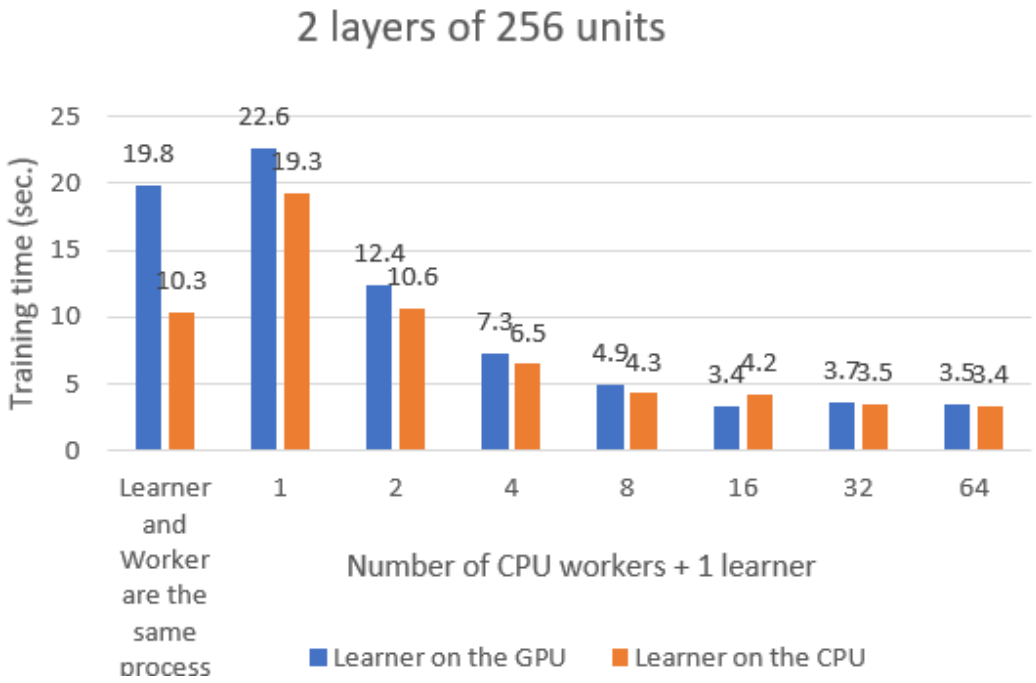

Figure 7.1 – *Distributed RL training over 3 episodes. More workers increases widely the training time (a few minutes), drawing theses bars would reduce the readability.*

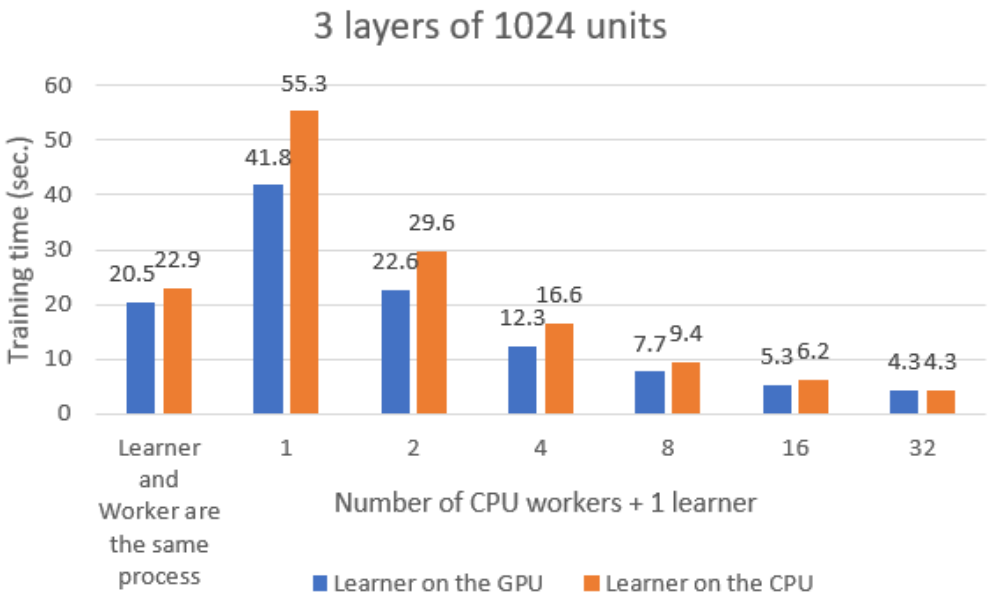

Figure 7.2 – *Distributed RL training over 3 episodes. More workers increases widely the training time (a few minutes).*

In our design, we have one single process by agent implementing all 4 tasks, but we benefit from hardware parallelism by training multiple independent agents with different

random seeds. Therefore, we reduce communication between processes to its minimum and cores utilization rate to its maximum due to a greater number of trials than available cores. We detail further this choice in section 7.2.1.

7.2 The proposed ensemble of RL agents

To solve a given control problem, Random Search to generate multiple Reinforcement Learning hyperparameters given an RL framework is a simple and robust method [90]. In this chapter, hyperparameters will not encode only one agent but the ensemble of agents.

Offline RL agents have two different policies depending on whether they are in the training or evaluation phase. The agent is trained with an explorative-exploitative behavior to learn to control the environment. Once the agent trained (i.e., inference phase), the agent acts with its best guess in the environment (exploitation only). Multiple training and evaluation are needed to measure the performance and the robustness of one given hyperparameter.

The procedure we follow is described as follows:

1. Hyperparameters are sampled with Random Search to build hundreds of ensembles. There are multiple variants we assess: either homogeneous ensembles or heterogeneous ensembles, and different combination rules.

2. In each ensemble, all base agents are trained independently and evaluated individually to produce an individual validation score.

3. Ensembles may be calibrated according to the individual validation score to calibrate the combination rule. Then, the validation score of the ensemble is computed.

4. The ensemble performing the best in step 3 is taken and deployed in production. We simulate the final performance in production by testing it the last time. It is the test score of the ensemble.

7.2.1 Training and tuning speed

In their construction phase, ensembles require the independent construction of base agents. Hyperparameter optimization with Random Search consists also to run multiple independent trials by sweeping some hyperparameters and in the end, taking only the best one on a validation score. More generally speaking, Hyperparameter optimization and ensemble both should leverage from a large amount of computing available resulting in bigger ensembles, well-tuned, and trained during more time.

All agents are trained in embarrassingly parallel. It is ideal when the number of trials is large, which is preferable due to a large hyperparameter space of more than 10 dimensions. We also observe that in the training phase GPUs may handle more efficiently embarrassingly parallel neural networks. And more, an embarrassing parallel of execution is generally a simple method available for all machine learning frameworks.

	Time (sec.) to train them			
Number of DQN RL agents	1	10	100	1000
Sequence of distributed trainings with rollout workers	3.4	34	340	3,400
Sequence of distributed trainings with rollout workers	3.5	35	350	3,500
Embarrassingly parallel CPU trainings	19.8	74.5	138.2	783.6
Embarrassingly parallel GPU trainings	10.3	29.6	84	732.8

Table 7.1 – *Time (sec.) to run different workload of RL training according the parallel pattern on the CPU. The initialization time is not measured here.*

Table 7.1 compares two different parallel patterns to run the workload. We use the time using 32 rollout workers and several independent trials with the equivalent workload. The workload consists of training on Pymgrid 1, 10, 100, and 1000 DQN RL agents of 2 layers and 256 units:

- Sequence of distributed training, each training is distributed across 32 rollout workers and 1 learner.
- Embarrassingly parallel the training, each training run in the same process plays both roles: the learner and the rollout worker.

We compare a few parallel settings to accelerate the training of distributed RL agents in a heterogeneous computing node. The possible parallel settings are endless: CPU/GPU, number of dependant trials, number of rollout workers... But our experiment is still incomplete, an ideal parallel setting should be tailored also a function of the nature of workload (small or large neural networks), the number of experiments, and the memory consumption per experiment.

7.2.2 Training, validation, and testing

Many reinforcement learning projects applied to some tasks such as video games or playboards are only interested in maximizing rewards during the training loop and compare RL approaches based on the training dynamics. We rather want to train our agents on a given runtime and measure how well they generalize. The method must be robust to the initial conditions such as different random seeds. We also want to avoid overfitting the training transitions stored in the replay buffer and avoid overfitting the time series data associated with the environment (Pymgrid contains some time series but not GEM). This is why we propose that the base agent training loops are independent with no sharing of the experience memory. No sharing of collected data samples allows them to build their own experiences and statistically maximize their diversity.

It is well known that most supervised machine learning projects have a training phase and two evaluations phase: the validation and the test which are a different subsets of data samples from the training phase. We use the same terminology to make good quality decisions in the context of Reinforcement Learning.

- The validation score allows making decisions on the neural network, like selecting the best agent given this score, calibration of the ensemble ...
- The test score allows to simulate the true score when it will be deployed in production. It may provide different values from the validation for multiple reasons: different random seeds, different number of states to evaluate ("different horizon"), and different time-series data (if any).

We name "R" the evaluation score ("test score" when it is not precised). This is the cumulative rewards during the evaluation phase. $R = \sum_{t=1}^{h} r_t$ for each time step t returning the reward r_t until the horizon h is reached.

In the build library of agents, we observe a strong correlation between the validation score and the test score which shows that a decision taken on the validation generalizes well on the test. It can be observed in figure 7.3.

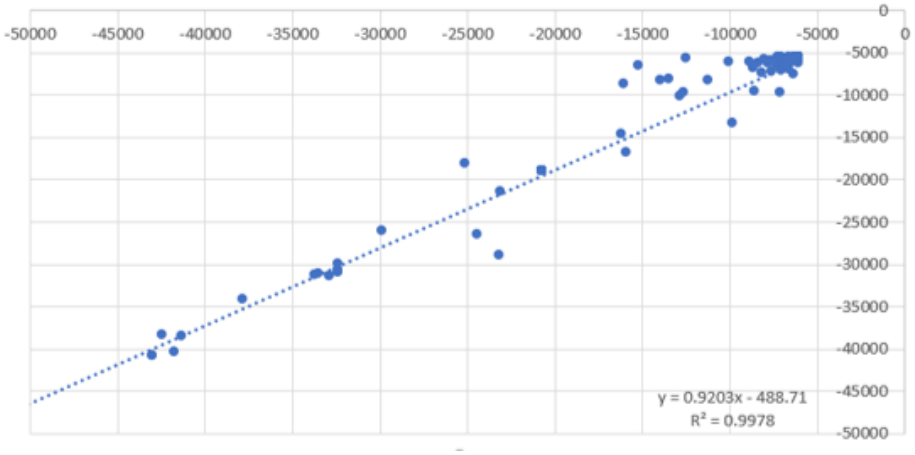

Figure 7.3 – *Correlation between validation score (horizontally) and test score (vertically) on Pymgrid even if they are performed on different time period.*

7.2.3 Ensemble construction

Homogeneous ensembles are built with n hyperparameters: each agent shares the same n hyperparameters but different random seeds. In heterogeneous ensembles of m agents, the ensemble has $m * n$ hyperparameters.

Table 7.3 and 7.2 show individual hyperparameters of our RL agents. The hyperparameter range has been set according to common literature values and preliminary experiments on Pymgrid without ensembles. For example, our preliminary experiments teach us that less than 2 layers underperform, and more than 8 is an uncommon value. Hyperparameters importance are more in-depth discussed in section 7.4.5.

We learn from a massive amount of RL experiences the importance to process the rewards on the cumulative reward. The processed reward r' is used only to train our agents, when we compare different agents we compare them based on the accumulated raw rewards.

DQN [56] hyperparameters used		
Variable name	Comment	Value range
width	Units per layer	[32;1024] log2 space
depth	Number of layers	[2;8]
reward_power	p of the reward processing	[0.5;2]
reward_scale	s of the reward processing	[0.01;100] log10
nb_previous_states	Temporal window size	{1,2,4}
batch_size	Training batch size	[16;256]
lr	Learning rate	[1e-4;0.1] log3
gamma	γ in the bellman equation	[0.2;1]
explor_rate_start	First exploration rate	[0.06;1]
explor_rate_end	Last exploration rate	[0;0.05]
dueling	Is dueling DQN enabled ?	{Yes;No}

Table 7.2 – *The hyperparameter spaces used for DQN agents*

DDPG [174] hyperparameters used		
Variable name	Comment	Value range
width	DNN width of actor and critic	[32;1024] log2
depth	DNN depth of actor and critic	[2;8]
reward_power	p of the reward processing	[0.5;2]
reward_scale	s of the reward processing	[0.01;100] log10
nb_previous_states	Temporal window size	{1,2,4}
batch_size	Training batch size	[16;256]
actor_lr	Learning rate of the actor	[1e-4;0.1] log3
critic_lr	Learning rate of the critic	[1e-4;0.1] log3
gamma	γ in the bellman equation	[0.2;1]
explor_rate_start	First exploration rate	[0.06;1]
explor_rate_end	Last exploration rate	[0;0.05]
tau	Soft update of target parameters	[3e-4;1e-2]

Table 7.3 – *The hyperparameter spaces used for DQN agents (up) and DDPG agents (bottom)*

We use this formula: $r' = sign(r) * s * abs(r)^p$ with r the raw reward given by the environment, s a scale factor and p which allows smoothing ($p < 1$) or increasing ($p > 1$) the relative importance of the raw rewards. The $sign(r)$ function return -1 or 1 according the sign of r, and $abs(r)$ is the absolute value function. They allow to process reward even if the environment produces negative values.

7.2.4 The ensembling procedure

After training independent agents their predictions are combined to improve their ability to make better local decisions in the same environment and improve the overall trajectory as well. Figure 7.4 illustrates our ensemble procedure.

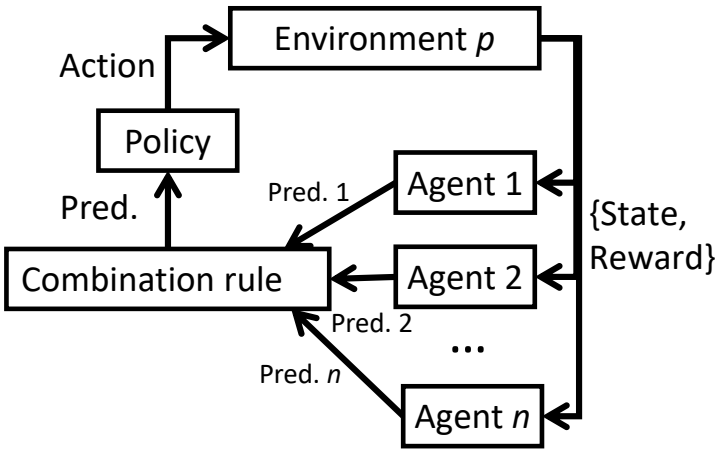

Figure 7.4 – *In the ensemble, the agents interact in parallel with the same environment. Rewards are only used to evaluate the ensemble.*

7.2.5 The combination rule

Multiple combination rules have been proposed by authors to combine neural networks predictions. The averaging combination rule simply interpolates predictions giving the same importance to each base agent without using the validation information. The weighted averaging [55] combiner uses the validation score to give more weights to the best models on the validation data but it may overfit validation information.

The Soft Gating Principle [139] is a combination method that has been proposed in supervised regression tasks to control the relative importance of the best estimators on some others. We observe it is also applicable to the reinforcement learning task with any number of base agents and both discrete and continuous action space.

The Soft Gating Principle is described by formula 7.1. It computes the weight w_j of base agent j based on its validation score R_j in the ensemble of size J. $\epsilon = 1e-7$ is a small value above zero to avoid division by zero cases. Then, computed weights are scaled such that the sum of weights is always equal to 1. The β value is a hyperparameter and repeated experiments will allow calibrating the combination rule. It controls the importance of

better models in the final prediction. It unifies averaging ($\beta = 0$), selecting only the best agent ($\beta \to +\text{inf}$) and weighted averaging (the intermediate values for β).

$$w_j = \frac{\sum_{i=1}^{J} R_i}{R_j^{\beta} + \epsilon}, \beta \in \mathbb{R}_0^+ \tag{7.1}$$

The effect of β on the relative importance of two agents is illustrated in figure 7.5 according to their different validation cumulative rewards. We show here only two agents for visualization purposes but it generalizes to more agents. For example, if agent 1 gets 150 validation cumulative rewards and agent 2 100 with $\beta = 1$, agent 1 will contribute to 60% of the final prediction, and agent 2 will contribute to 40%.

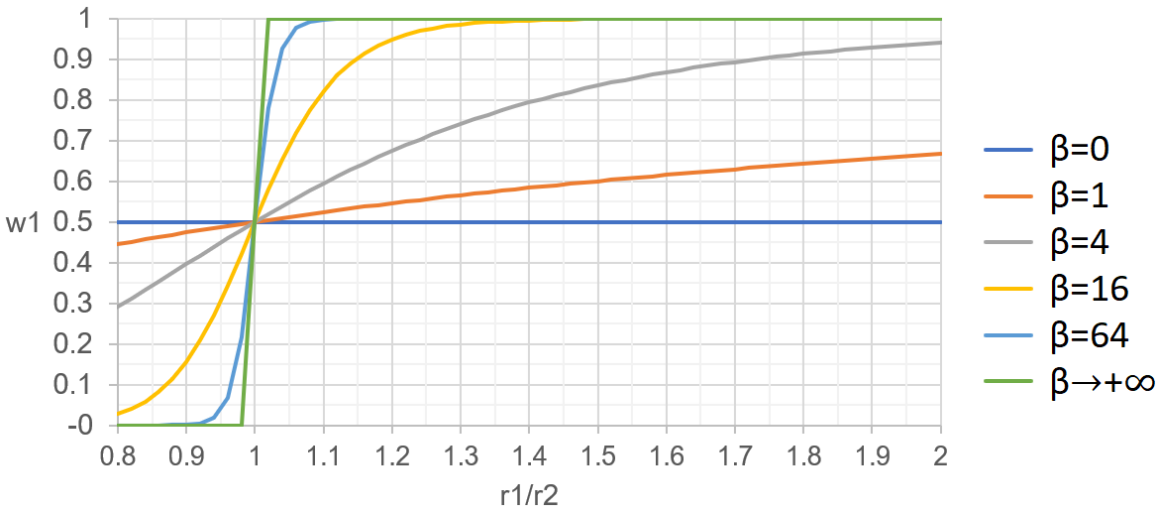

Figure 7.5 – *Agent 1's weight (vertically) by varying to its relative score compared to agent 2 (horizontally) and different value of β. β allows to give more importance to the best agent according the formula 7.1.*

7.3 Experimental settings

In the first analysis, we arbitrarily choose the ensembles containing a fixed number of 4 base agents. We evaluate multiple ways to create ensembles: homogenous/heterogeneous and different β values. This section details experimental settings. We use GEM [161] and Pymgrid [60] environments. Both are available with a continuous and discrete action space. We use DQN for discrete environments [56] [167] and DDPG [174] for continuous ones.

GEM environment (Gym Electric Motor) is a simulator of various electric drive motors. They contain components such as supply voltages, converters, electric motors, and load models. The goal is to maximize the performance of the engine under safety constraints. If a state exceeds the specified safety limits, the episode is stopped and the lowest possible reward is returned to the agent to punish the limit violation. In all our benchmarks we use the Permanent Magnet Synchronous Motor ("PMSMCont-v1" and "PMSMDisc-v1" code name). We set the horizon to 100K steps. All our agents have been trained 30 episodes in

those environments. Validation runtime and test runtime simply run one episode but with different random seeds.

Pymgrid environment (PYthon MicroGRID) is a simulator of microgrids. Microgrids are contained electrical grids that are capable of disconnecting from the main grid. Automatic control allows for minimizing production costs by honoring the demand. The reward produced are negative values to maximize given by formula $r = -1 * (c + p)$ with c cost expressed as dollars and p a penalty term when the demand for electricity experiences a power cut. In all our benchmarks we use the second microgrid available. This microgrid has multiple components connected: the demand, the solar panels, the battery, the generating set, and an unstable link with the main grid allowing to buy/sell kWh. This simulator is mixed with real sunlight time series data. The training performs 5870 steps, 438 validation steps, and 2453 test steps. Each phase receives different sunlight time series. Finally, all our agents have trained 500 episodes in those environments.

The software stack. We use RLLIB [96] and Ray [110].

The platform. We train and evaluate each population on one computing node identical to the Oak Ridge Summit node. It contains 6 Tesla-V100 GPUs and a Power9 AltiVec 2 sockets CPU, each socket containing 72 virtual CPU cores. Inference experiments have been performed on an HGX-2 containing 16 Tesla-V100 GPUs and an Intel Xeon Platinum 8168 2 sockets CPU, each one containing 48 virtual cores.

7.4 Experiment results

7.4.1 Ensemble construction comparison

We measure the performance of ensembles according to their construction (Homogeneous or Heterogeneous) and combination rules ($\beta in \{0, 1, 4, 16, 64, +\infty\}$) and the two simulators (Pymgrid and GEM). They are represented in figures 7.6, 7.7, 7.8, and 7.9. Each column in the figures represents a population of 100 ensembles with randomly drawn hyperparameters in the hyperparameter space described in table 7.3. The whisker boxes show the 10th, 25th, 50th, 75th, and 90th percentiles, and the circles the performance of the atypical ensembles superior to the 10th percentile and inferior to the 90th percentile.

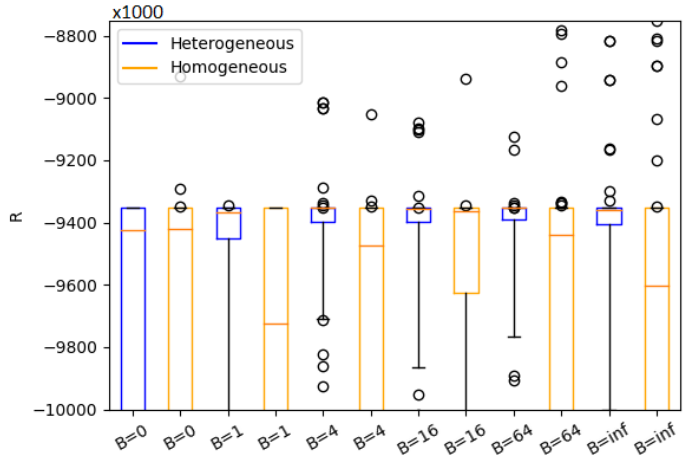

Figure 7.6 – *Pymgrid with DQN.*
Top 3 produced ensembles: Homogeneous with $\beta = inf$ (-8.75M), Homogeneous with $\beta = 64$ (-8.78M), Heterogeneous $\beta = inf$ (-8.82M).

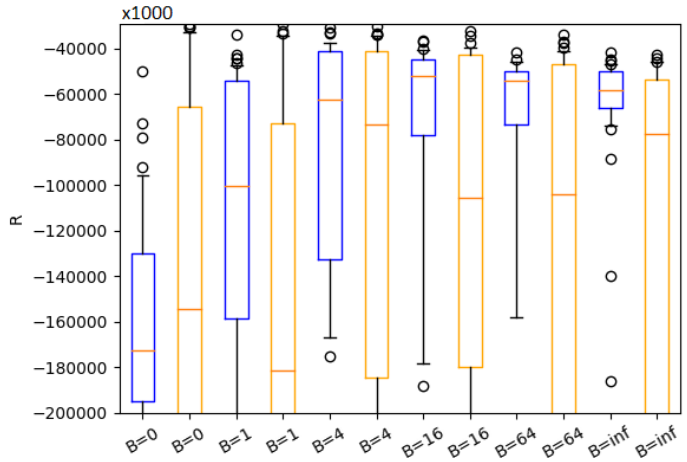

Figure 7.7 – *Pymgrid with DDPG.*
Top 3 produced ensembles: Heterogeneous with $\beta = 4$ (-30.74M), Heterogeneous with $\beta = 1$ (-33.7M), Heterogeneous with $\beta = 16$ (-36.3M)

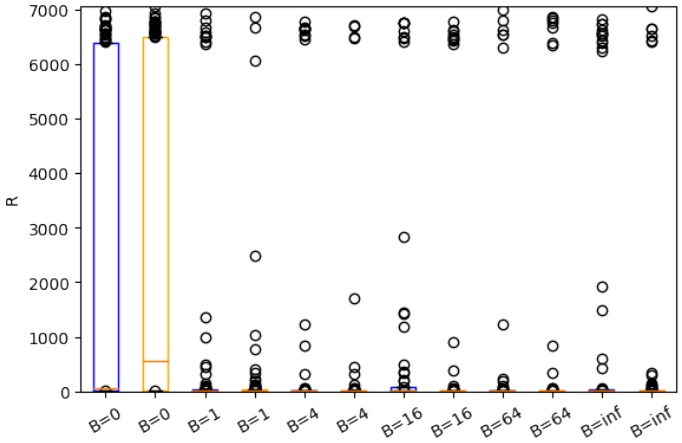

Figure 7.8 – *GEM with DQN.*
Top 3 produced ensembles: Homogeneous with $\beta = inf$(7064), Homegeneous with $\beta = 0$ (7057), HeBo(6974)

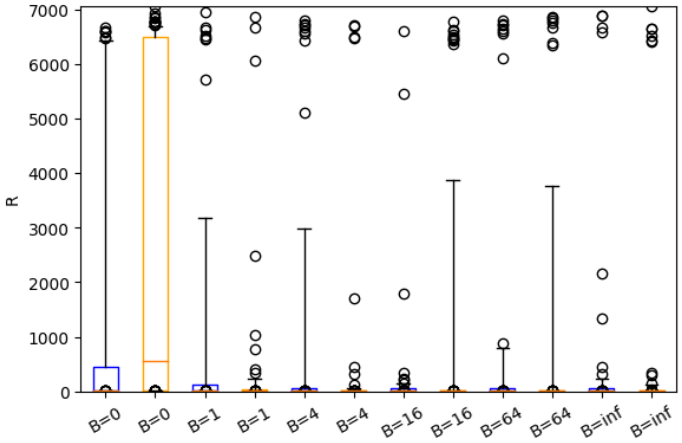

Figure 7.9 – *GEM with DDPG.*
Top 3 produced ensembles: Homogeneous with $\beta = inf$ (7065), Homogeneous with $\beta = 0$ (7056), Heterogeneous with $\beta = 1$(6958)

Hetero. Homo. distribution comparison. The first major observation is that different ensemble construction procedures yield a very different distribution of the performance.

RL agents are unstable by nature and running the same training at different runtime produces already diversified predictions. Heterogeneous ensembles combine hyperparametric and parametric diversity yielding again more diverse base agents. This diversity is maybe not useful, because it comes with the cost of much more dimensions in the hyperparameter space and is much more difficult to calibrate. In most cases, heterogeneous ensembles have a tighter distribution of R, for instance, the difference between the 1st and the last quartile is smaller or equal to homogeneous ensembles in all our run times.

Homogeneous is a more risky construction. If a homogeneous ensemble gets a good hyperparameter set, all four base agents have a chance to well perform but the opposite is also true: a poor hyperparameter set means all poor agents. In practice, this is not a

problem that several ensembles are performing poorly, because only the ability to generate good ensembles matters.

Combination rule comparison. Another important observation is that homogeneous ensembles with $\beta = 0$ provide the best decile or equivalent decile each time compared to other β values. This shows that the simple procedure consisting to build homogeneous ensembles and simply averaging predictions is a robust ensemble procedure.

In practice, the goal is to find the best ensemble and discard all the others. However, evaluating the procedure to build them based on only the best one is noisy information, this is why the decile is used as a measure of the robustness of the automatic ensemble construction.

Top 3 comparison. We observe no specific pattern when we are looking only at the top-3 best ensemble constructions due to random effects. There is a total of 12 available positions (Top 3 with 2 environments and 2 action spaces, 3*2*2=12). 7/12 are homogeneous compared to 5/12 heterogeneous. Furthermore, 4/12 has been build with $\beta = +\infty$, 3/12 with $\beta = 0$ and 5/12 with intermediate β values.

Distribution in discrete action space. A last piece of information is that multiple ensembles may follow the same trajectory and accumulate the same rewards in the discrete action space. For instance, it is possible that 1st decile=1st quartile in shown figures.

7.4.2 Stability analysis

The previous section shows the final performance distribution with different ensemble construction procedures. It is intuitive but it is often an insufficient analysis in critical applications where stability and reproducibility are needed. Scientists want to ensure its reproducibility for transparency, confidence, and sharing experiments. After a minor update of the environment and re-training of the RL algorithm, it is expected to find similar performance.

In table 7.4 the ensemble construction procedure is evaluated. First, top-3 performing hyperparameters on the validation score is selected. Then, ensembles are trained/validated/tested with 4 different random seeds. Finally, the mean and stability of the test score of the top3 ensembles are computed. More formally, they are computed with $mean(mean(R_{1,1}, R_{1,2}, ...), mean(R_{2,1}, R_{2,2}, ...), ...)$ and $mean(std(R_{1,1}, R_{1,2}, ...), std(R_{2,1}, R_{2,2}, ...), ...,)$ with $R_{i,j}$ with $R_{i,j}$ is such that i is the i^{th} best hyperparameter (from 1 to 3), and j the different run times (from 1 to 4).

Comparing only the standard deviation between construction procedures is not fully relevant to observing the stability. For example, the bottom right corner experiments get a low standard deviation score compared to the others, this is since the ensemble produces a low mean score too. We propose instead the Relative Standard Deviation (RSD) given by $RSD(x) = std(x)/mean(x)$. In this case, we observe that $\beta = 0$ gives generally lower RSD. It also appears that between the homogeneous and heterogeneous procedures no one seems more stable than the other in all cases.

β	DQN Pymgrid Homog.	DQN Pymgrid Heterog.	DDPG Pymgrid Homog.	DDPG Pymgrid Heterog.
0	$-9.21M \pm 35K$	$-9.36M \pm 7K$	$-30.2M \pm 0.75M$	$-5.2M \pm 5.85M$
16	$-9.23M \pm 53K$	$-9.14M \pm 99K$	$-33.7M \pm 1.87M$	$-3.8M \pm 1.93M$
∞	$-8.87M \pm 183K$	$-9.02M \pm 144K$	$-46.9M \pm 4.14M$	$-5M \pm 5.88M$

β	DQN GEM Homogeneous	DQN GEM Heterogeneous	DDPG GEM Homogeneous	DDPG GEM Heterogeneous
0	6193 ± 1066	5816 ± 1131	3559 ± 3177	5487 ± 1140
16	5837 ± 1032	5551 ± 1147	1675 ± 1346	3436 ± 1182
∞	5258 ± 2155	4496 ± 2286	2423 ± 1775	186 ± 138

Table 7.4 – *Reproducibility of the top ensembles. The blue figures indicate minimum Relative Standard Deviation value for each environment and action space.*

We can also observe the stability gained by ensembles compared to its base agents not only after a horizon but after each training episode as shown in figure 7.10. The ensemble learning dynamics is much less noisy and may be useful to perform decisions such as early stopping [127] of all base agents at the same time.

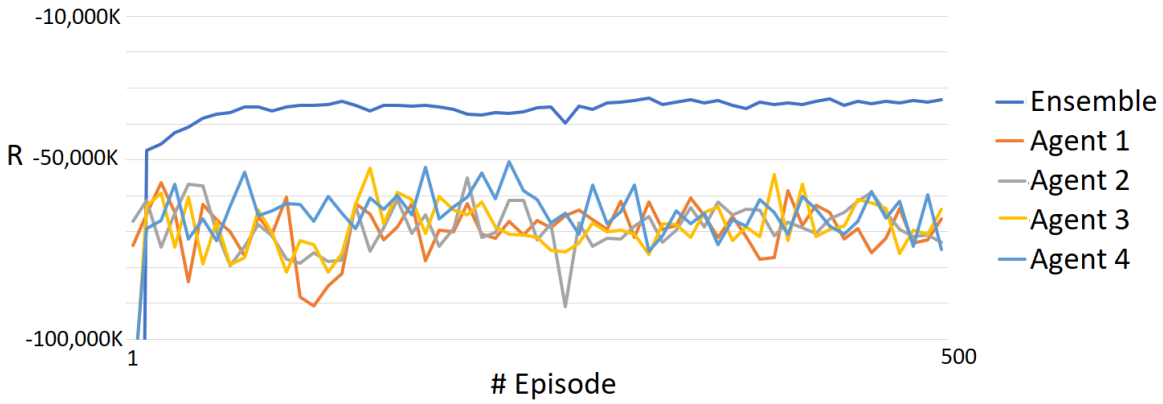

Figure 7.10 – *Performance of the ensemble compared to its individual agents over 500 training episodes. After each training episode, we assess the ensemble test score and its base agents test scores. It is an example from the library of homogeneous DDPG agents on the Pymgrid environment.*

7.4.3 Expansion of ensembles

Once we have found hyperparameters set for homogeneous ensembles, to improve again more the ensemble predictions we may train more base agents with the same hyperparameter set. This simple procedure is not possible with heterogeneous ensembles, if more agents are wanted a new Random Search is required to find a new hyperparameter set.

We scale homogeneous ensembles from 1 to 16 base agents on both environments and both RL frameworks. The blue curve indicates the mean results on 4 different training-

test runtimes and the blue cloud around the standard deviation. They are illustrated in figure 7.11 7.12 7.13 7.14.

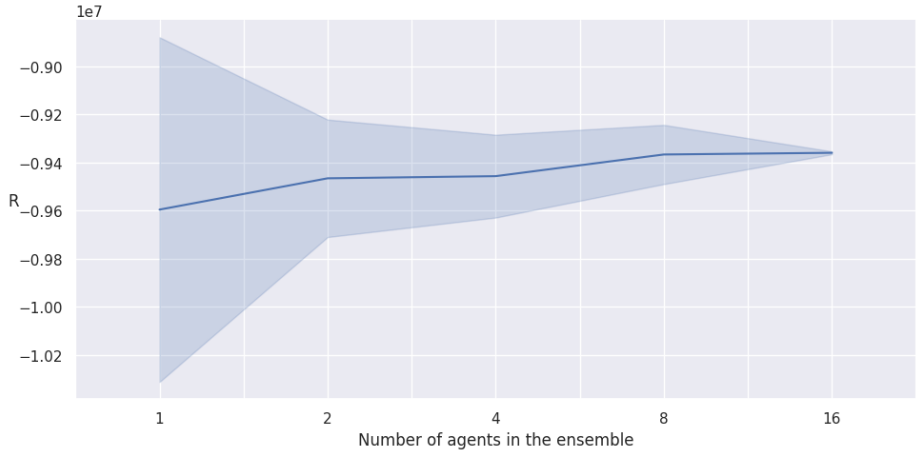

Figure 7.11 – *DQN on Pymgrid. R is the cumulative test rewards*

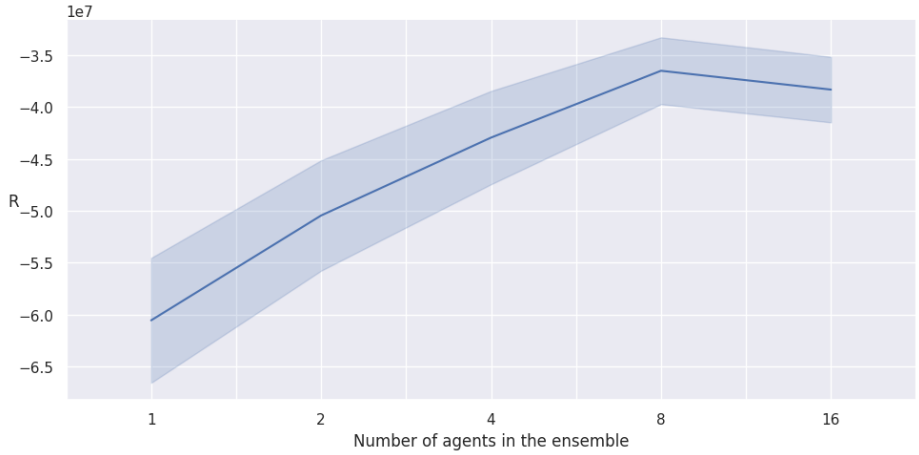

Figure 7.12 – *DDPG on Pymgrid*

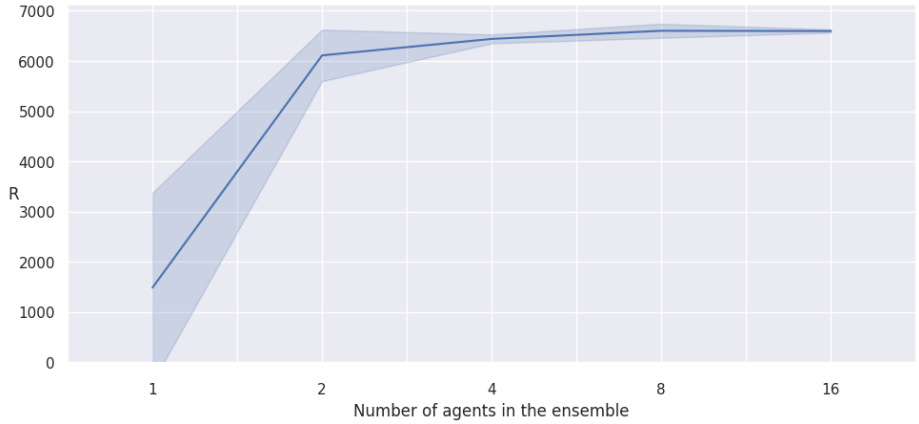

Figure 7.13 – *DQN on GEM*

#	Pymgrid DQN	Pymgrid DDPG	GEM DQN	GEM DDPG
1	-9596 ± 716	-60566 ± 6020	1486 ± 1890	891 ± 290
2	-9466 ± 244	-50460 ± 5321	6112 ± 513	1843 ± 2642
4	-9457 ± 172	-42951 ± 4497	6440 ± 89	3858 ± 2825
8	-9367 ± 123	-36507 ± 3220	6602 ± 140	5474 ± 388
16	-9360 ± 6	-38323 ± 3160	6597 ± 32	6674 ± 70

Table 7.5 – *We scale homogeneous ensembles from 1 to 16 base agents on both environments and both RL frameworks. The trend is more readable in figures 7.11, 7.12, 7.13, and 7.14. Pymgrid units are expressed in thousands.*

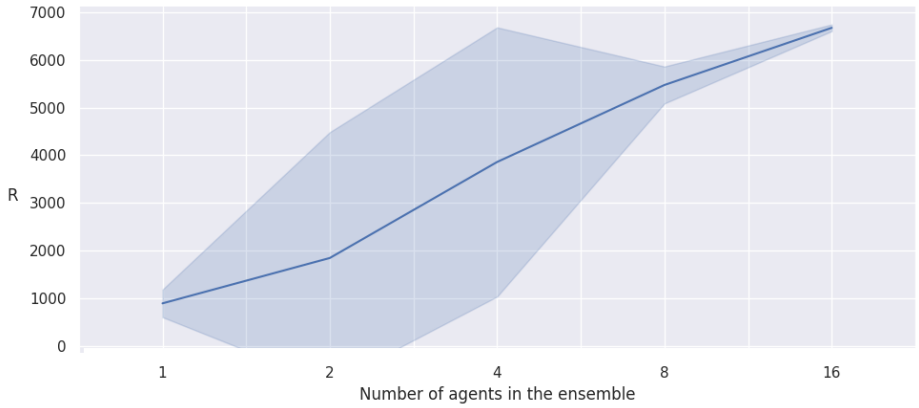

Figure 7.14 – *DDPG on GEM*

We pick the best hyperparameters with $\beta = 0$ of our homogeneous ensembles of size 4. Then we run training and testing 4 times by varying the number of base agents. The mean score and its standard deviation are given in figures 7.11 7.12 7.13 7.14 and the same values are displayed in table 7.5 for better readability of the values.

The global trend is clear, when the number of RL increases both the mean performance and the stability are improved. From 1 to 16 the mean cumulative rewards for those 4 cases are multiplied by {1.03, 1.58, 4.45, 7.49} and the instability measured with the relative standard deviation is divided by {116.4, 1.2, 262.2, 31}. Those experiments show ensembling is a huge benefit for stability and improves the mean performance too.

More agents (e.g., 32, 64, 128...) should exhibit again better results at the cost of more computing power. However, it seems unpredictable how much may be expected before running such experiments. The pattern between the number of agents and those metrics is not so obvious.

7.4.4 Power and time consumption at training time

We compare different hardware allocation of agents (noted 'a') and the environments (or "simulator" noted 's') in terms of computing time and computing power to perform the same work: we train 100 DDPG agents on Pymgrid during 1 episode, hyperparameters are drawn randomly (Random Search).

Allocation settings	Mean power (watt)	Training time (sec.)	Energy consumed (kWh)
1a/2CPU s/CPU	870	928	0.224
1a/CPU s/CPU	875	773	0.188
a&s/CPU	885	1498	0.368
1a/GPU s/CPU	669	1791	0.333
2a/GPU s/CPU	717	1217	0.242
4a/GPU s/CPU	770	759	**0.162**
8a/GPU s/CPU	1005	**598**	0.167
16a/GPU s/CPU	OOM	OOM	OOM

Table 7.6 – *Time and energy consumed by our GPU cluster with different allocation of hardware for training 100 DQN agents 1 episode. In the "allocation settings", "a" corresponds to "agent" and "s" to "simulator". The trend is more easily readble in figure 7.15.*

Our cluster contains 144 virtual CPU cores and 6 GPUs. Our motherboard is equipped with a sensor to measure power consumption. Some evaluated allocations are detailed as follows and results are shown in table 7.6: "a/2CPUs, s/CPU" means each agent is trained using 2 multi-threaded CPU cores while its associated simulator is put on another CPU core. It makes 48 agents handled at the same time. "a/CPU, s/CPU" means each agent is associated with one core while its associated simulator is associated with another core. It makes 72 agents handled at the same time. "a&s/CPU" means each agent and its associated simulator are handled by the same core. It makes 144 agents handled at the same time. "8a/GPU, s/CPU" means 8 agents are co-localized in each GPU, and the corresponding simulators are associated with CPU cores. It makes 48 agents handled at the same time. Additionally, we stopped "4a/CPU, s/CPU" before termination because it takes too much time, and "16a/GPU, s/CPU" leads to memory crashes.

Those results show us that we can train efficiently multiple agents at the same time on a modern GPU cluster. Different allocations provide different computing performances, "8a/GPU, s/CPU" is the fastest allocation while "4a/GPU, s/CPU" is the most power-efficient. We also observe that the CPU-only implementation ("a/CPU, s/CPU") may offer reasonable time performance without requiring GPU investment.

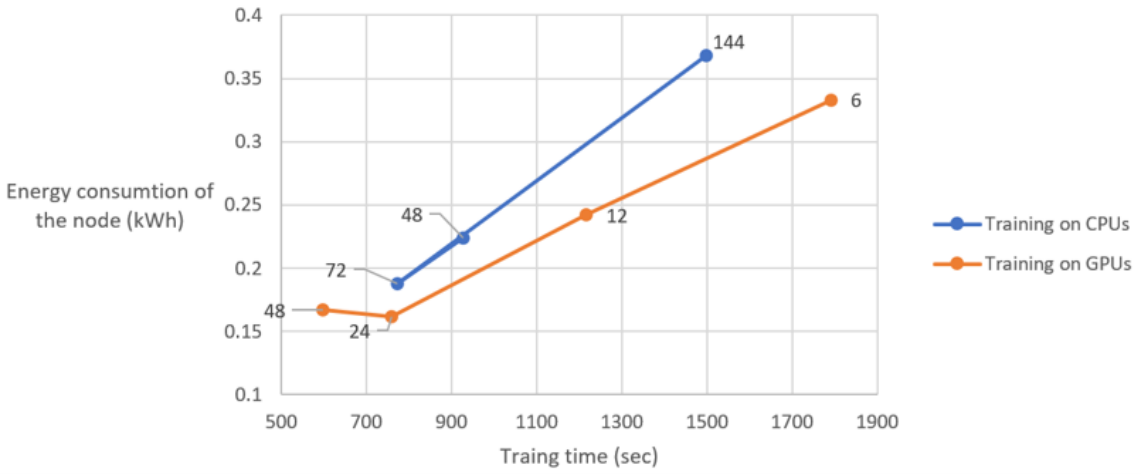

Figure 7.15 – *Time and energy consumed by our GPU cluster with different allocation of hardware for training 100 DQN agents 1 episode. The number represents how many independant training runs at the same in the computing node. The exact values are more easy to read in table 7.6*

7.4.5 Interpretation of hyperparameter values

The large number of trials performed and saved on the disk does not only allow for selecting the best ensemble. It allows also us to extract useful knowledge from the mapping between hyperparameter values and the associated score which has been randomly sampled hundreds of times during the hyperparameter optimization process.

We may identify multiple pieces of information and experiment with them in preliminary experiments: the sensitivity of the hyperparameter, we may formulate a criticism of the hyperparameter range, and the global pattern of the hyperparameter on the score. Note the hyperparameter range used in those preliminary experiments in section 7.2 may differ from those shown here.

First, the **sensitivity** of the hyperparameter on the target score. In the case where the hyperparameter has a strong impact, this may indicate which should tune them with a smaller log base are give more liberty to tune. This is illustrated in the bar plots by the large disparity between bars indicating a strong effect of these hyperparameters. On the opposite, the useless hyperparameters may be removed to simplify the hyperparameter search. For example, we discovered that the batch size has a low impact on the cumulative rewards (figure 7.19) and a strong impact on the training time (figure 7.20). In this case, a small and performant batch size is preferable.

Second, we can also visualize and verify if the maximum and minimum hyperparameter values are suitable. If the hyperparameters range is too short, we may fail to explore promising values. On the opposite, when it is too large, the hyperparameter optimizer loses time to sample poor hyperparameter values in the nonpromising spaces. For example, the learning rate range should be re-centered between 1e-4 and 1e-2 to sample good values (figure 7.19).

Third, the **general pattern** of the hyperparameter may give additional insight for

adapting the hyperparameter space. It may be a bell curve, a threshold function, ascending or descending ...

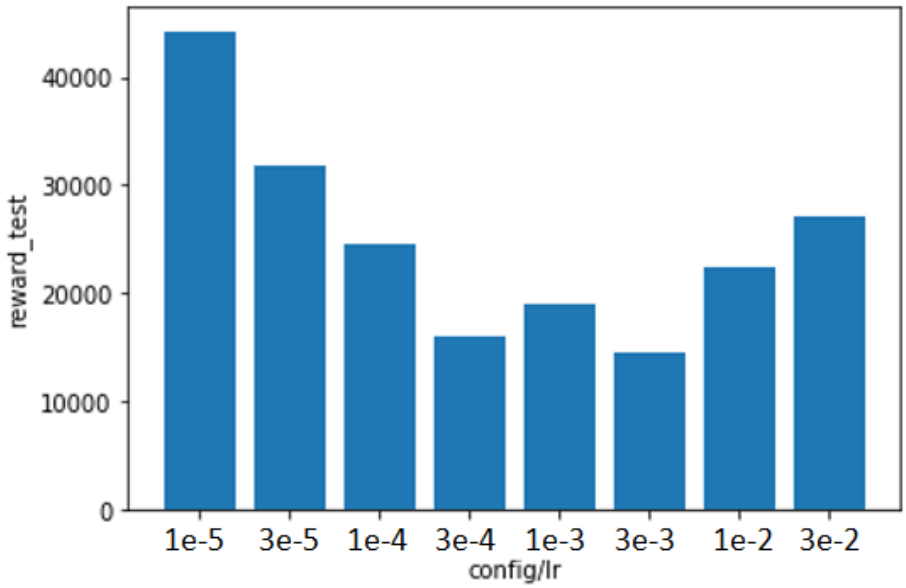

Figure 7.16 – *The distribution of the test cumulative reward by varying the* ***learning rate*** *on Pymgrid. Lower is better.*

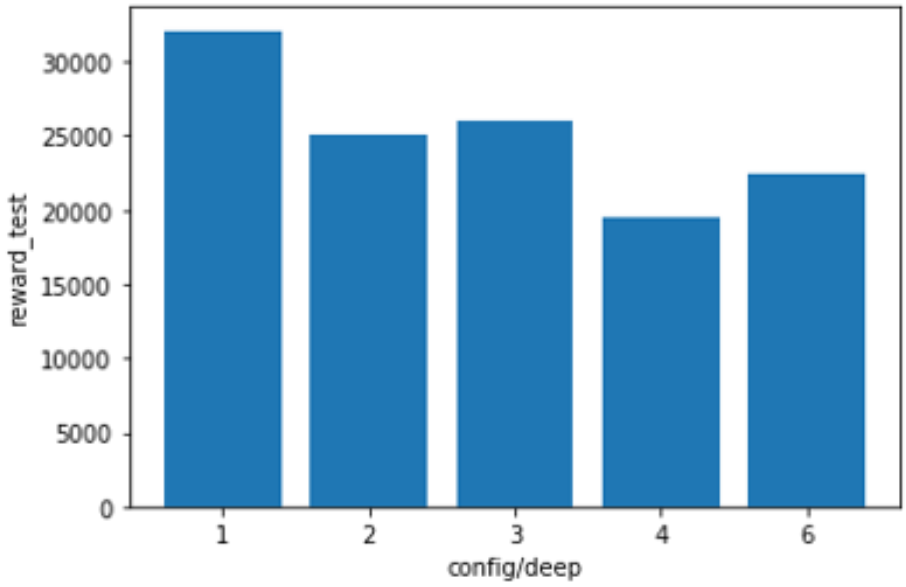

Figure 7.17 – *The distribution of the test cumulative reward by varying the* ***neural network depth*** *on Pymgrid.*

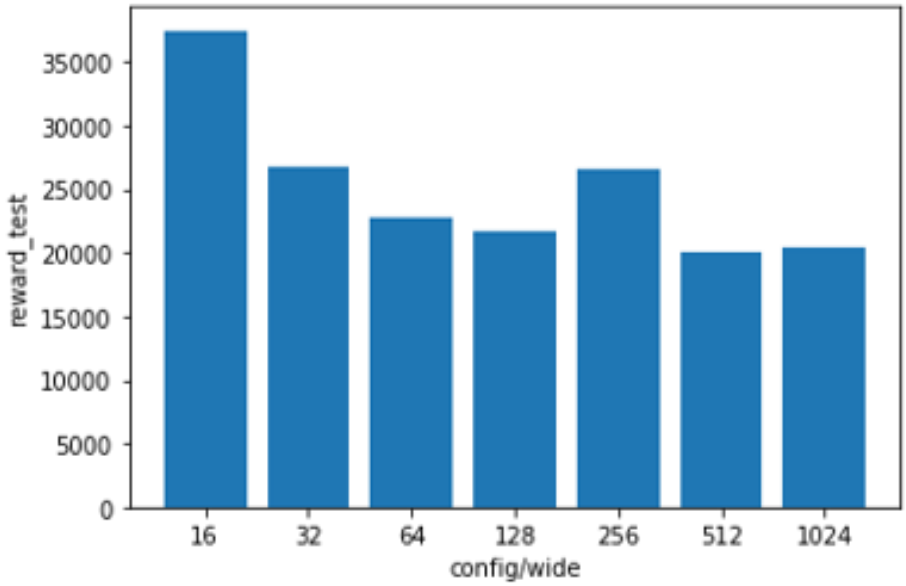

Figure 7.18 – *The distribution of the test cumulative reward by varying the* ***neural network width*** *(or "number of units per layer") on Pymgrid.*

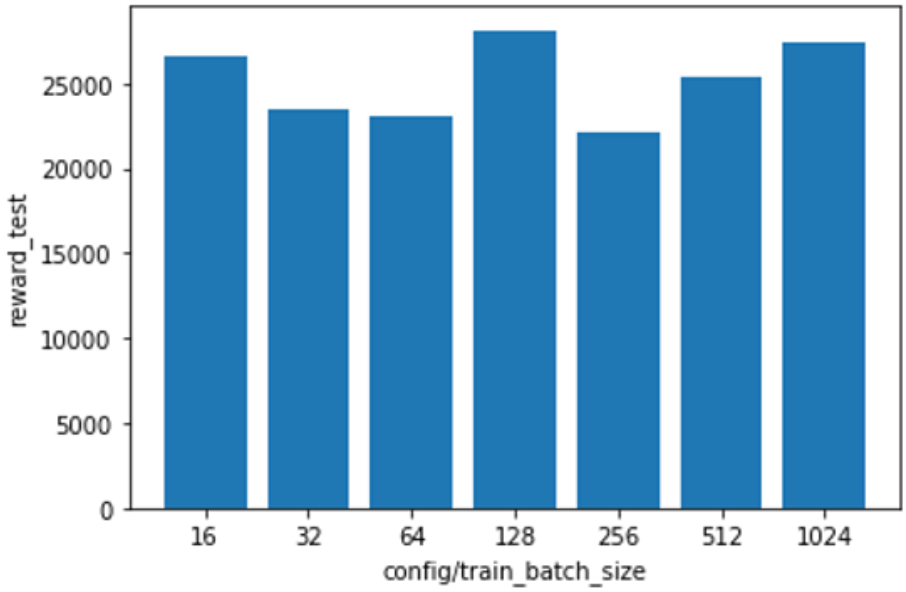

Figure 7.19 – *The distribution of the* ***test cumulative reward*** *by varying the* ***batch size*** *on Pymgrid.*

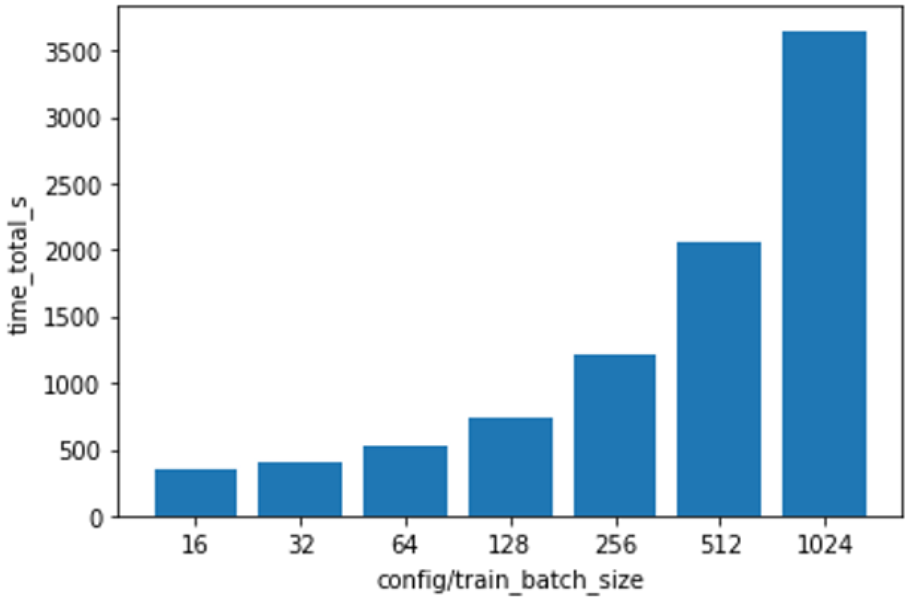

Figure 7.20 – *The distribution of the **training time** by varying the **batch size** on Pymgrid.*

Even if the ensemble is more stable the hyperparameter is still of major importance to get good results. Those analyses show a few examples of the potential of large experiments to give insight and criticisms of the hyperparameter space. More in-depth analysis is possible with stored information on the disk but is beyond the scope of this thesis. For example, the hyperparameter choice is known as not transferable [182] in RL according to the environment. Another emerging idea is that tuning hyperparameter during the training time [88] [182] shows better results than static hyperparameter but strongly decreases the reproducibility of experiments. It may be interesting to see how much dynamic hyperparameter tuning may be mixed with ensembles that stabilize the learning dynamics. We let it for future investigations.

7.4.6 Ensemble at inference time

Training and tuning hundreds of ensembles is a computing-intensive activity where the ensembles train batch by batch, but it is a rather time-bounded activity. The inference is a long-running activity where the ensemble is hosted in memory and waits for one single state to quickly return its associated action. In the inference phase, the agents are sequentially interacting with the environment one state and one action at a time. Therefore, the computing speed is measured with the latency defined by the elapsed time between the ensemble receiving a state from the environment, and the ensemble returning its associated action.

The homogeneous ensembles are trivial to parallelize because each base agent predicts at about the same speed synchronized by the combination rule. Therefore, it limits the case when the cores are idle waiting that the slower agent finishes predicting. Figure 7.21 shows

the study of the scalability on an HGX2 cluster containing 16 Tesla GPUs and 1 dual-socket Intel CPU.

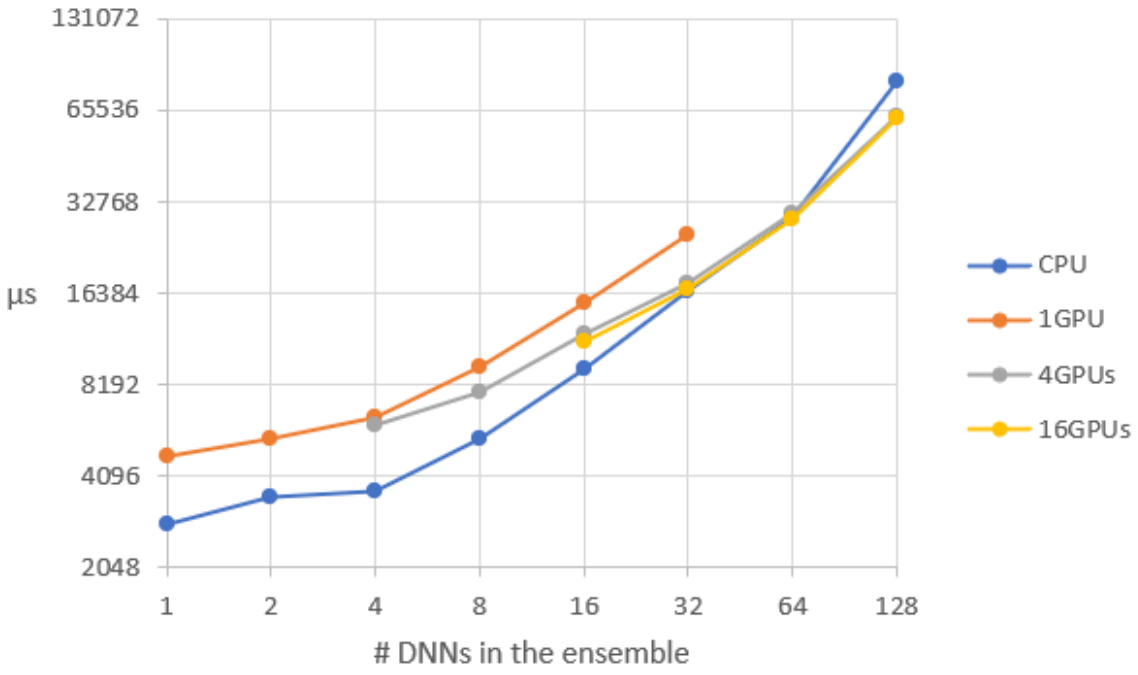

Figure 7.21 – *Median prediction latency (microseconds) of diverse homogeneous ensemble size. The DNNs have 2 layers of 256 units, 11 inputs, and 7 outputs. Note, both axis are log2 scale.*

The first observation is that the CPU seems more suitable for the inference phase. It is about 2 times faster than one GPU to predict the data flow. Moreover, the greater memory of the CPU allows for handling bigger ensembles while 1 GPU cannot handle 64 DNNs. We also observe that increasing the number of GPUs is not very effective to accelerate the prediction flow. Finally, when the ensemble is bigger than 64, multiple GPUs become faster than the CPU to handle the workload.

The second key observation is that increasing the number of DNNs from 1 to 8, multiplies by only 2 the prediction times. This reinforces our claim that modern computers may efficiently run these workloads.

Each point of this figure is the median of hundreds of timesteps in the simulator. Another relevant information for real-time applications is the distribution of the latency which wants the narrower as possible. The 95th percentile of the latency spread is up to 2.5 times slower than the median value, and the 99th percentile is up to 10 times slower than the median. Experiments have been done on a generalist computing platform, real-time operating system (RTOS), and dedicated hardware may increase the time determinism but could decrease the batch workloads performance such as the training phase.

7.5 Future works and distribution shift

Electric simulators may be developed with a high degree of realism but some unexpected and rare events may occur. [105] Pymgrid does not provide events such as dust covering solar panels. GEM does not implement the risk of captor breakdown. Indeed in industrial applications, there are often some unexpected events, simulating and imagining every possible event is impossible.

A key machine learning skill in dealing with such situations is that the model recognizes its ignorance when unexpected states occur, as well as complicated states. In addition to the performance and stability gain, ensembles are known [82] [57] to produce well-calibrated uncertainty estimates in the supervised case. Further analysis should investigate how well uncertainty estimate is reliable in the context of RL. If the research is conclusive, when high uncertainty is detected we should be able either to trigger an emergency procedure.

7.6 Summary

This work answers this missing piece between reinforcement learning and industrial applications aiming at stable, qualitative, and efficient algorithms. Today, these tasks are generally solved with reinforcement learning agents without ensembles, but there is a significant improvement when multiple agents predict together. The ensemble is less sensitive to random effects and may cumulate rewards during the simulation. Then, we observe that modern GPUs and CPUs can efficiently train multiple DNNs independently at the same time. Furthermore, one CPU may deploy them efficiently. Today, we may say that in those electricity control applications homogeneous ensembles is a method which worth the computing cost.

Figure 7.22 shows different technics assessed to control Pymgrid. We observe that HPO+Ensemble allows us to generate better results with extra-linear stability gain (not seen in this figure). And more, while it is commonly admitted that the main drawback of ensembles is the computing cost, we demonstrated that a homogeneous ensemble of size 4 has a limited overhead compared to 1 model due to the internal parallelism of both GPUs and multi-core CPUs.

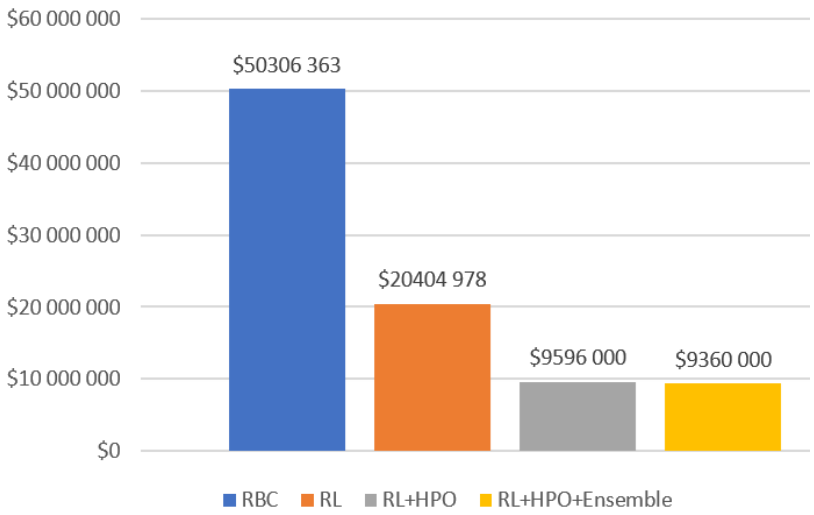

Figure 7.22 – *Cumulative rewards comparison in the Pymgrid test case. Rule-based Control (RBC) developed by the domain specialist, Reinforcement Learning with default hyperparameters set, the best Reinforcement Learning agent among 400 agents, and homogeneous ensemble of 16 agents averaging their predictions (the best hyperparameter in the space of 4 agents among 100 expanded to 16). Re-train the ensemble (RL+HPO+Ensemble) produces about 100 times more stable cumulative rewards compared to the single agents (RL+HPO).*

The previous chapter and this one aim to find efficient and systematic methods to build optimized ensembles. The next one will analyze different scenarios to infer ensembles by exploiting multiple GPUs. We contribute to the field by proposing novel methods and evaluating them at scale.

Inference system for ensembles 8

A machine learning algorithm creates value only when it is deployed in the inference phase. As previously seen, we know that the most qualitative model is often made of several DNNs combined. That is why efficient inference systems are required to build qualitative and high-speed underlying resources.

Our work benefits from already proposed inference servers aiming at messages management and inference systems aiming at individual DNNs performance. The novelty is that we add an intermediate software layer to manage the ensembles of heterogeneous DNNs. This contribution was published in IEEE BigData 2021 [125].

8.1 Potential workflows and challenges

Potential workflow exists according to if the ensembles are homogeneous or heterogeneous. They may also differ in function of the application needs: if the data come one by one or if the ensemble predicts a large number of data samples in batch mode. Table 8.1 illustrates the different ensemble types, data pattern, and their inherent challenges.

		Challenges			
Data pattern	Ensemble type	Hetero. comput.	Mem. Alloc.	Data parallel.	Batch tuning
Batching	Homogeneous	N	Y	Y	Y
Batching	**Heterogeneous**	**Y**	**Y**	**Y**	**Y**
Data flow	Homogeneous	N	Y	N	N
Data flow	Heterogeneous	Y	Y	N	N

Table 8.1 – *The 4 inference system types according ensemble and data pattern. We associate them their computing challenges to tackle for building fast predictions. The fifth challenge not shown in this table is exploiting the heterogeneous devices (e.g., GPUs and CPUs).*

We identify four main challenges for efficient ensemble predictions: heterogeneity of the computing cost, allocation in the limited amount of memory, allocation of the workers for data parallelism, the tuning of the batch size, and heterogeneous computing devices.

First, the heterogeneousity of the neural networks computing resources consumption. In general, we should avoid, when it is possible, the biggest neural network slowdown during the entire computing time. A smart inference algorithm should give more computing resources to the slowest neural networks to avoid delaying ensemble predictions.

Second, the allocation challenge. Inference with ensemble requires allocating n neural networks to the m computing devices. The allocation becomes an NP-complete problem similar to bin packing when the allocation is not trivial (i.e., a discrete optimization problem, in which n objects of different sizes must be packed into a finite number of m bins).

To simplify, we consider objects cannot be splittable in this section. The neural network may be heterogeneous, meaning they have different memory requirements and different computing times. The allocation produces a performance score or an out-of-memory error. The problem we are facing is indeed again more complex than a pure bin-packing problem by the fact we do not only want a possible memory allocation, we also want an allocation that delivers fast predictions too.

If no allocation is found or possible, intensive I/O access is required to load the neural network when data samples come. This may result in very poor performance. We do not explore such methods in this case.

Third, in the batch applications, the data parallelism of the neural networks may allow increasing the ensemble speed but it requires enough computing power and memory to do this. The data parallelism in the inference mode consists in predicting with multiple neural network replicas (or "workers") and combining and sorting the predictions. If enough resources are available, data-parallel to the slowest neural network is beneficial to accelerate the overall ensemble.

Forth, in batch applications, the tuning of batch applications is essential to increase the speed of workers. However, bigger batch sizes generally consume more computing and memory resources. Last but not least, it is possible to give different batch size values for different data-parallel workers of the same neural network.

Fifth, the modern computing infrastructure has evolved in heterogeneous computing node containing CPUs, GPUs, TPUs, each one with different computing power, memory capacity, interconnection links, ... And more, we see the emergence of more and more interconnected infrastructures such as two different HPCs, or hybrid HPC-cloud.

All those 5 challenges are combined when we want to predict efficiently with an ensemble of heterogeneous neural networks on batch applications with heterogeneous devices. To do this is the main challenge we address in this chapter.

All experiments have been developed with Tensorflow on the HGX-2 machine with Python. We consider the latency as the time between data samples known from the CPU side and consider the prediction is finished when their associated predictions are stored on the CPU side.

8.2 Data flow applications

Data flow applications are generally interested in improving their speed performance measured as the latency. In those applications, data samples comes generally one per one and therefore return their associated predictions one per one too. The reinforcement

learning applications are a perfect illustration of data flow applications where an RL agent receives one state and returns its associated action.

Latency improvement cannot benefit from internal data parallelism with batch size or data parallelism. This makes those applications generally simpler to tune. Their performance depends on the efficient parallelism of neural networks and careful data structure for accessing the incoming data samples and synchronizing their predictions.

Reinforcement learning is typically data flow applications where data samples (states) came one by one, and predictions (actions) are returned also one by one.

8.2.1 Allocation system

The algorithm 1 shows our proposed system to predict efficiently with m heterogeneous model in the ensembles on d heterogeneous devices. The amount of possible allocation is d^m showing it cannot be reasonably evaluated by brut-forcing all possible allocations. The problem is more complicated than a bin packing problem because we are not only interested in founding an available solution in terms of memory, but we assess them on hundred of requests to compute a latency score. A typical latency score is a median latency, 95th percentile, or the 99th percentile.

Algorithm 1: Allocate DNNs into a computing node to fit into memory and optimize the time

1: **input:** N the list of neural networks in the ensemble, D the set of devices, $default\_device$ where to perform benchmarks such as $default\_device \in D$, $calib\_data$ contains calibration data samples to perform offline benchmarks
2: **output:** P the key/values pairs which associates all the devices D to their neural networks
3: **start**
4: // First structures are initialized
5: $P_i \leftarrow$, $i = 0...length(D)$
6: D are sorted according their speed in $default\_device$ to predict on $calib\_data$
7: **for** $i \leftarrow 0$ to $length(D)$ **do**
8: $avail\_D \leftarrow$ All
9: **if** $avail\_D =$ **then**
10: raise out of memory error
11: **else**
12: $j \leftarrow$ gets the identifier of the fastest in $avail\_D$ to run $D[i]$
13: $P[j]$.append($D[i]$)
14: **end if**
15: **end for**
16: return P

We propose a greedy algorithm to break down those d^m possibilities in $m * d$ evaluated possibilities. This algorithm is thought to allocate heterogeneous neural networks on heterogeneous devices and may generalize to all other cases: homogeneous neural networks

on heterogeneous devices, heterogeneous neural networks on homogeneous devices, and homogeneous neural networks on homogeneous devices.

8.2.2 Prediction system

After getting an ensemble of DNNs, we need to serve it efficiently on the available computing resources. We design an efficient inference pipeline illustrated in figure 8.1. When DNNs are waked up by the orchestrator, they read the data sample from the shared memory space and return their associated prediction to the orchestrator. All those operations are performed asynchronously.

The orchestrator asynchronously runs in one CPU-core the cumulative averaging of all predictions to avoid slowing down the entire pipeline compared to gathering all predictions and performing the average.

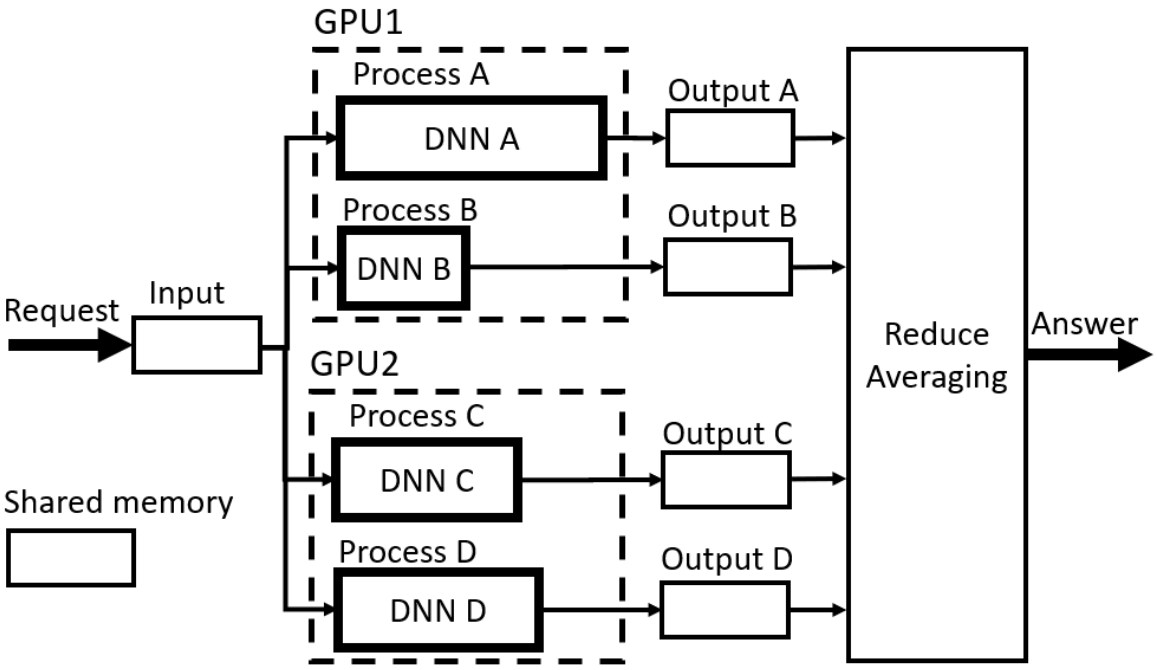

Figure 8.1 – *Toy example of allocation of 4 heterogenous DNNs into 2 GPUs for inference. There are 5 processes, the orchestrator containing and its 4 children A, B, C and D. The orchestrator receives data to predict and return the ensemble prediction. Shared memory are buffers in the RAM.*

8.3 Batch predictions workflow

Once an ensemble has been trained and built, we need to serve it efficiently on the available computing resources. We design an efficient inference pipeline illustrated in figure 8.2.

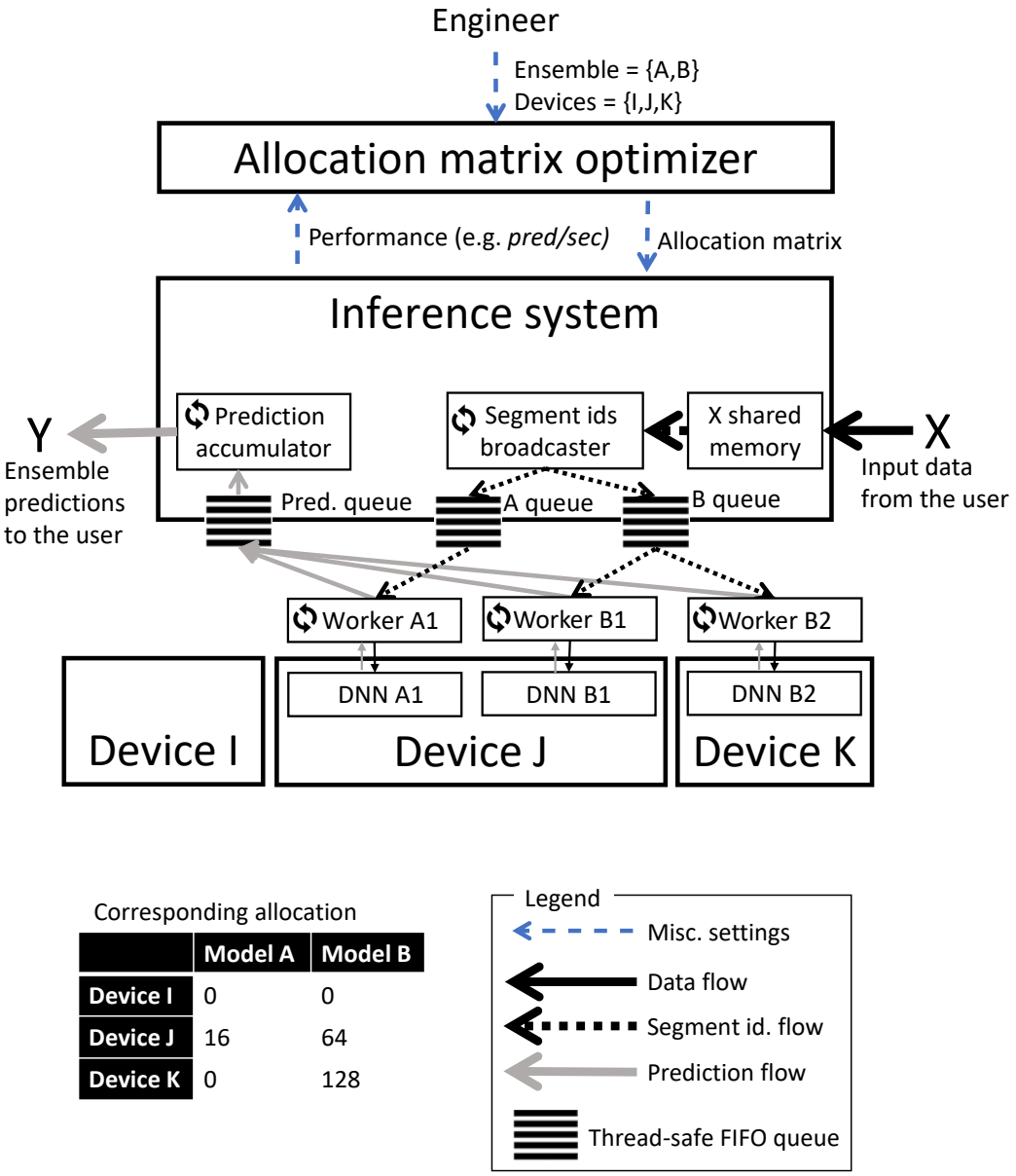

Figure 8.2 – *Illustration of our inference server on a toy example of allocation of 2 DNNs into 3 devices. The DNN model B is run by 2 data-parallel workers on device J and device K. The DNN instances A1 and B1 are co-localized in device J. The corresponding allocation matrix is described in the bottom left corner. Threads inside a worker are not fully described for visibility purposes (see figure 8.3).*

In the following sections, we will describe first how the inference server is used. Then, the allocation matrix which drives performance. Finally, main components of our server are presented one-by-one: the *inference system*, the *worker-pool* and the *allocation optimizer*.

8.3.1 Using our server

The engineer who is responsible for managing the inference server provides to the *matrix allocation optimizer* the ensemble of DNNs, the CPUs, and the GPUs to use. Sometimes,

the engineer does not want to give all available devices into its cluster to deploy an ensemble, so he can keep GPUs for other applications. The matrix allocation optimizer will automatically optimize the allocation of the ensemble for the given devices.

Once the allocation matrix is computed, the inference system is deployed online. It implements the usual inference server features such as an HTTP/HTTPS wrapper and adaptative batching. To be more precise, the term "adaptative batching" can now lead to confusion. The buffer waiting request is now defined by the size of segments and not the batch size of the individual DNNs.

8.3.2 The allocation matrix data structure

The allocation matrix is a data structure describing with high flexibility the design of the *worker-pool* used by the *inference system*. It designs how DNNs are allocated into the devices including co-localization (multiple DNNs instances into the same device), data-parallelism (one DNN multi-threaded on multiple devices), and the batch size.

In this matrix, element o means an absence of a worker process in the given device, while other values represent the batch size. The workers co-localized into the same device are readable as non-zero values in the row of the allocation matrix. The workers which are instances of the same DNN are data-parallel, they are readable as the non-zero values in a given column.

Finally, some devices may not be used, so we can observe rows containing only zero values. But it is illicit to have a column with only zero values. In other words, all DNNs must be represented in the ensemble.

Co-localization

Co-localization allows fitting more DNNs than the number of devices (the pigeon-hole principle). Co-localization allows maximizing the utilization of internal cores in a device too.

It is well known that the batch size is an important setting. However, optimal batch size values in the context of multiple DNNs co-localized into one GPU are challenging to find. In general, the larger batch may increase cores utilization and it consumes more memory. Due to those complex relationships between multiple DNNs co-localized and hardware, only benchmarks allow knowing the performance of co-localized models. And more, only multiple benchmarks allow for finding good settings such as which DNNs are put together and their batch size values.

Data-parallelism

Data-parallelism allows to speed up a prediction of one DNN using multiple devices. The workers run the same DNN but in different instances. They take data samples to predict from the same input FIFO queue.

Ideally, n threads should multiply the number of images predicted per second by n. Yet, because we use multiple shared data structures: one segment ids FIFO queue, the shared data memory, and the prediction FIFO queue, perfect scalability is not ensured. Only benchmarks allow to compute the performance and if increasing the number of workers is worth it.

8.3.3 The inference system

The inference system is the core component of the server. It is a function $f(X, A) -> \{Y, S\}$ with X data samples to predict and Y the associated predictions of the ensemble of DNNs. A is the allocation matrix driving the worker-pool construction and S is the performance score.

The inference system can be run with 2 modes. In "Deploy Mode" it is deployed online to serve client requests, A is fixed and P is ignored. In "Benchmark Mode" it measures the performance P provided by the allocation matrix A on the data calibration samples X and Y is ignored.

To accelerate the inference system we design it with multiple processes. The *segment identifiers broadcaster* which splits the incoming workload of requests into segments, the *worker pool* containing DNNs instance and return segments of predictions in the *combination accumulator* FIFO queue, the *combination accumulator* combines segments of predictions and returns to the client the final prediction: the prediction of the ensemble.

The segment ids broadcaster

The *inference system* contains thread-safe FIFO queues allowing to broadcast and gather information with the workers. It contains also the *X shared memory* it is a heavy buffer of data readable by all the workers. All those data structures are stored in the RAM.

To avoid transmitting heavy messages and stressing thread-safe FIFO queues, they are gathered into segments, and only segment identifiers are passing through queues. All segments contain N samples, except the last segment which contains the information of the remaining samples.

After getting a segment identifier s, such as $s >= 0$, a worker knows he is responsible to predict the images from $s * N$ position to the $min((s + 1) * N, nb\_images)$ position with nb_images the number of images in the *X shared memory*. The *segment ids broadcaster* can also put special values like $s = -1$ to ask workers to shut down.

For example, in the figure 8.2 if the user requests the prediction for 300 images with $N = 128$, they are represented internally as 3 segments, two are size 128 and one is size 44. Then, the *metadata broadcaster* put 6 messages: 0, 1, 2 integers into A queue and into B queue.

The prediction accumulator

This process combines efficiently predictions from workers and when it is finished it returns them to the client. After predicting a segment of data, a worker puts a message in the prediction queue. Each of these messages is a triplet $\{s, m, P\}$ with s the segment identifier, m the model identifier and P the prediction matrix of dimension $(end(s) - start(s)) \times C$ and C the prediction length for each image e.g., the number of classes. To combine predictions the *prediction accumulator* first allocates a buffer Y of dimension $nb\_data \times nb\_classes$ zeroed. Then, it updates the cumulative prediction each time it receives a message triplet $\{s, m, P\}$. The Python code using Numpy arrays of the averaging accumulation is simply:

```
Y[start(s):end(s)]+=P/M
```

Other combination rules can be easily implemented such as majority voting or weighted averaging. Furthermore, other applications can require specific combination rules such as those applied in object detection [150]. Any combination rule code must be developed by keeping in mind that predictions come into messages to be asynchronous with the neural network predictions.

To go into further details, the workers can send special messages to the *prediction accumulator*. The special message {-1, None, None} allows notifying that a device has not enough memory to load or initialize a DNN. This triggers the shutdown of the *inference system* and every process into it. The special message {-2, None, None} allows notifying that one worker is ready to serve after its initialization. We know the *inference system* is ready to receive the client requests when all workers send {-2, None, None} to the *prediction accumulator*.

8.3.4 The worker pool

The worker pool gets segments from FIFO queues, predicts them, and returns the predictions to the *prediction accumlator*.

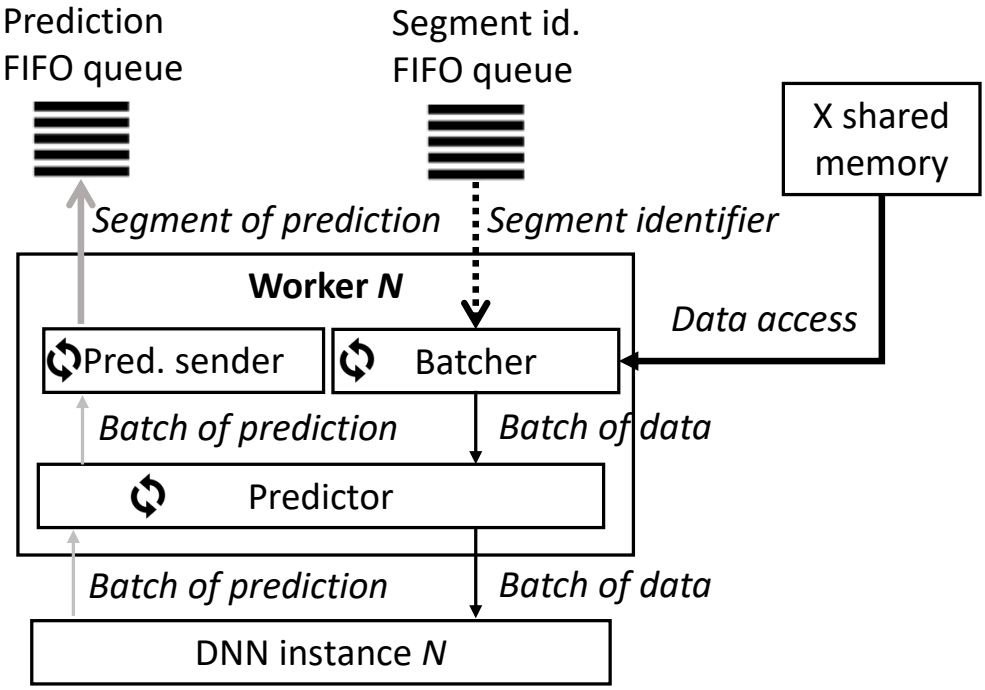

Figure 8.3 – *Anatomy of a worker. The meaning of arrows is the same than figure 8.2*

All workers are designed the same, they are illustrated in figure 8.3. To be performant it contains 3 asynchronous threads: the *batcher*, the *predictor* and the *prediction sender*.

- The *batcher* waits for incoming segment identifiers from the input FIFO queue. When it receives one, it splits the segment into a batch of data and gives them to the *predictor*. Each worker has its batch size described in the allocation matrix.
- The *predictor* persists the DNN into the device memory. When it receives a segment identifier, first it loads the segment of data. Then it runs the prediction on each batch and finally, it gives the batch of predictions to *the prediction sender*.
- The *prediction sender* gathers predictions batch by batch to build segments of prediction. When a segment is completed, this process puts the segment of prediction into the prediction FIFO queue.

8.3.5 The allocation optimizer

The *allocation matrix optimizer* goal is first to fit DNNs into the memory and then optimize their performance. That is why it runs first a worst-fit-decreasing algorithm to solve the bin-packing problem to fit DNNs into the memory of the device. Second, a greedy algorithm assesses thousands of matrices to find the faster one based on offline benchmarks. Finally, the best matrix is cached to avoid recomputing it again when the server will be restarted. These two algorithms are detailed now.

Algorithm 1 - Worst-Fit-Decreasing with priority to GPUs

It solves a bin packing problem to put objects (DNNs) into a finite number of bins (devices). All DNNs are set with the minimum batch size value (8 in our experiments). It is more exactly a bin packing with an offline heuristics problem because the DNNs are known before execution and can be sorted in decreasing order. We attempt to find a feasible solution with the already known Worst-Fit Decreasing algorithm.

This algorithm has the already known properties to optimize memory allocation. At each step of the worst-fit algorithm, it chooses to allocate a DNN to the device with the largest available memory. Therefore it may remain a large memory space in the GPU after a DNN allocation. This remaining memory space can be big enough so that other smaller DNNs can also be placed in that remaining memory and therefore maximize the memory filling. Second, ordering the input list by decreasing DNNs size has been proved [106] to maximize the memory filling compared to a non-ordered version. Finally, Worst-Fit prioritizes an equitable workload between homogeneous devices based on the memory criterion. On the opposite, First-Fit, Best-Fit, and Next-Fit, attempt to fill the first devices and keep the last devices empty.

To improve significantly the speed up of the allocation, we hard code a rule to allocate in priority the GPUs rather than CPUs. It is common knowledge that GPUs can run DNNs an order of magnitude faster than CPUs. Indeed, the CPUs start to be used by algorithm 2 only when no more space is available on the GPUs.

`more_remaining_memory` function returns the device with the most remaining memory. `fit_mem` returns if the allocation matrix is feasible in terms of memory availability.

Algorithm 2 - Bounded greedy optimization

Algorithm 2 goal is to refine the starting allocation matrix and return a faster allocation matrix. Before explaining this algorithm, we will first provide a detailed analysis of the complexity of its decision space which drives our choices to design it.

The number of matrices is given by equation 8.1 with D devices, B batch size values and M DNN models in the ensemble.

$$total\_matrices = ((B+1)^D - 1)^M \tag{8.1}$$

The number of possible element values is $(B+1)$ all possible batch size values plus the 0 elements for no worker. The number of possible values in a single column is described by $(B+1)^D - 1$. The term $(B+1)^D$ allows to brute-force all possible values in a column minus the zero columns which are forbidden.

Note the two power terms which show the explosion number of combinations. For example, if we have 8 DNNs, 4 GPUs, and 1 CPU $total\_matrices \approx 1.3E31$ much more than the number of stars in the universe. To put the problem into perspective, one allocation matrix takes an average of 40 seconds to be assessed. It includes the construction of the

inference system time plus the offline benchmark time. That is why only a few thousand matrices can be reasonably assessed and smart optimization must be used.

A greedy algorithm provides an often effective complexity reduction to many combinatorial optimization problems. When applied to our case, it starts with the matrix given by algorithm 2 and at each iteration, it assesses all neighborhood matrices and replaces the current matrix with the best-assessed matrix. We consider that two matrices are neighborhoods if they are both valid (again, no 0 columns) and if there is only one different element between them. Any greedy algorithm is stopped when no neighborhood improves the current matrix.

The greedy algorithm breaks down the overall complexity described in equation 8.1 into a succession of combinations (or neighbors) measured by equation 8.2 to assess. With F the number of forbidden matrices we cannot explore such $0 \leq F \leq D$.

$$total\_neighs = (B + 1) * (D * M) - F \tag{8.2}$$

Let's take our previous example, 8 DNNs, 4 GPUs, and 1 CPU. The total number of combinations is $total\_matrices \approx 1.3E31$ but the greedy algorithm needs to assess only between 232 and 240 neighbors at each iteration.

Greedy algorithms are well-known approximation algorithms. However, the number of neighbors can still be very computing-intensive to evaluate in some cases, and the number of required iterations can be large before to found an optimum. To limit the computing cost of the greedy we propose two bounds. First, each iteration evaluates at most *max_neighs* randomly drawn neighbors (line 9). We also limit the number of iterations to *max_iter* (line 6).

Even though our algorithm is an approximation of a greedy algorithm, we have the guarantee that in the worst-case scenario a solution as good as the starting one is returned. This characteristic is inherited from the greedy algorithm. Line 18 can be read "if we do not improve strictly the performance, the algorithm leaves the loop".

The rate of visited neighbours is measured with $\frac{max\_neighs}{total\_neighs}$. If it is close to zero it means the final solution performance may be very volatile. At the opposite, $\frac{max\_neighs}{total\_neighs} \geq 1$ means all neighbours are visited and does not introduce volatility. Another source of volatility is the benchmark function (lines 4 and 11) but in practice, when the amount of calibration data samples is large enough, the measurement is stable.

The pseudo-code is presented in bloc code 3. `bench` function instantiates the pipeline with the given allocation settings (first argument) on the calibration data samples (second

argument) and returns the performance to maximize or 0 if a DNN instance does not fit in memory.

Algorithm 3: Bounded greedy algorithm

input: *max_iter* maximum number of greedy iteration, *max_neighs* maximum number of neighboors to evaluate at each iteration, A is the zeroed allocation matrix, *calib_data* contains calibration data to perform offline benchmarks
output: The optimized allocation matrix A
start
$A\_speed \leftarrow$ bench(A,*calib_data*)
$iter \leftarrow 0$
while $iter < max\_iter$ **do**
  $neighs \leftarrow$ neighborhood(A)
  if length($neighs$) $> max\_neighs$ **then**
    $neighs \leftarrow$ draw randomly max_neighs samples from $neighs\_A$
  end if
  $best\_A$, $best\_speed \leftarrow$ for all n in $neighs$ return the best one and its score based on the bench(n,*calib_data*) criterion
  if $best\_speed > A\_speed$ **then**
    $A \leftarrow best\_A$
    $A\_speed \leftarrow best\_speed$
    $iter \leftarrow iter + 1$
  else
    // local maxima (or plateau) detected
    $iter \leftarrow max\_iter\_greedy$
  end if
end while
return A

8.3.6 Offline benchmark settings: ensembles, hardware

This section provides a detailed description of our experimental settings.

The performance metric. The transmission of online messages with the client and their characteristics is application dependant and thus we rather perform offline benchmarks to evaluate the core prediction performance. This allows us to stay focused on the specificity to deploy ensembles under a heavy workload of requests. Therefore, our metric will be the throughput measured by the number of data samples predicted per second.

Code and framework. All the asynchronous objects are implemented with the "Multiprocessing" built-in Python 3.6 package. Our FIFO queues, shared memory, and processes are respectively implemented with Queue class, Manager class, and Process class.

Additionally, we use two well-known external packages: the Numpy numerical library and Tensorflow 1.14 to deploy models widely supported by GPUs and CPUs.

The hardware. All results reported are performed into an HGX-2 cluster containing 16 Tesla-V100 GPUs. The flexibility and efficiency of our server are analyzed by varying the number of GPUs to use from 1 to 16.

We also performed a few benchmarks on a computing node identical to the Oak Ridge Summit node. Both clusters have the same GPUs but different CPUs and different operating systems. The measured results are very similar to HGX-2 so we decided to not report the performances.

The ensembles of DNNs. To assess efficiency and flexibility we benchmark 5 heterogeneous ensembles named according to the number of models and the name of the database they learned from.

First, we build 3 ensembles of famous DNNs for reproducibility purposes:

- **IMN1** contains only ResNet152 and it allows us to show our workflow is general enough to optimize one single DNN.
- **IMN4** contains 4 DNNs {ResNet50, ResNet101, DenseNet121 and VGG19}.
- **IMN12** contains all DNNs from IMN1 and IMN2 plus {ResNet18, ResNet34, ResneXt50, InceptionV3, Xception, VGG16, MobilNetV2}.

Then, we generate 2 others ensembles named **FOS14** and **CIF36**. FOS14 contains 14 DNNs and were build with the workflow described in chapter 6.

Calibration data samples. The meaning of the data has no impact on any performance measured on the classification task. However, applying this system to other DNN methods may need a few realistic data samples to measure a relevant performance time. For example, in the Faster-RCNN [131] architectures the RPN module iterates on some region of interest for each input image.

The possible batch size values. We fixed {8, 16, 32, 64, 128} as possible batch size values. More values increase significantly the number of possible allocation matrices and make the exploration of combinations more difficult. Fewer values reduce the degree of freedom to control internal parallelism, memory consumption, and data exchange.

The segment size. We evaluate multiple segment sizes and we observe that smaller values reduce the granularity of the workload and improve its distribution between processes. In all this work the segment size is fixed to 128. It should generally be equal to or greater than the maximum batch size.

Algorithm 2 - Greedy allocation. We choose $max\_neighs = 100$ and $max\_iter = 10$. This is a total of at most 1000 combinations to assess. On average one matrix evaluation takes 40 seconds therefore the computing is limited to 12 hours.

When $D - M > max\_iter$ with M DNNs and D devices. The max_iter is replaced with $D - M$. It allows getting a chance of using all devices when the number of devices is large. In all our benchmarks, it is used only three times with IMN1 ensemble on 12 GPUs and 16 GPUs and IMN4 on 16 GPUs.

8.4 Offline benchmarks

This section describes 3 performance analyses of our server. The first one estimates the overhead introduced by the inference system. Then we analyze the performance varying the ensembles and the number of GPUs. Finally, we will compare our allocation matrix optimizer to a simple baseline named "Best Batch Size".

8.4.1 Overhead of the inference system

In all our benchmarks, the inference system overhead is measured at most 2% of the total inference time. To estimate this overhead we temporarily replace all the DNNs calls with a fake prediction containing only zero values and the combination still gathers predictions but returns zero values.

We perform multiple offline benchmarks of those fake inference systems and we report it takes at most 0.035 seconds in the case of IMN12 ensemble on 16 GPUs the *allocation matrix optimizer* producing 22 workers. In comparison, the true inference system (without faking predictions), takes 2.528 seconds to predict 1024 images (i.e., throughput=405 in the table 8.2).

8.4.2 Varying GPUs and ensembles

Table 8.2 shows the performance in terms of prediction performance of the 5 ensembles. The first major observation is that to serve efficiently an ensemble of DNNs we do not need systematically as much as GPUs as DNNs, but more GPUs generally improve performance.

	IMN1		IMN4		IMN12		FOS14		CIF36	
#G	A1	A2	A1	A2	A1	A2	A1	A2	A1	A2
1	106	136	-	-	-	-	-	-	-	-
2	106	270	13	101	-	-	213	233	-	-
3	106	394	158	199	-	-	308	339	-	-
4	106	539	160	251	15	24	380	410	-	-
5	106	617	160	294	65	106	388	461	15	15
6	106	722	160	351	103	194	397	470	35	37
8	106	974	160	472	103	226	483	518	239	243
12	106	1436	160	686	103	317	511	545	428	481
16	106	1897	160	877	103	405	511	559	563	633

Table 8.2 – *We benchmark the throughput of 5 ensembles on different numbers of GPUs (+1 CPU). The mention '-' means memory error is returned. Because A2 is a stochastic algorithm, each run time was performed 3 times and the median value is reported. The trend is more easily readable in figure 8.2.*

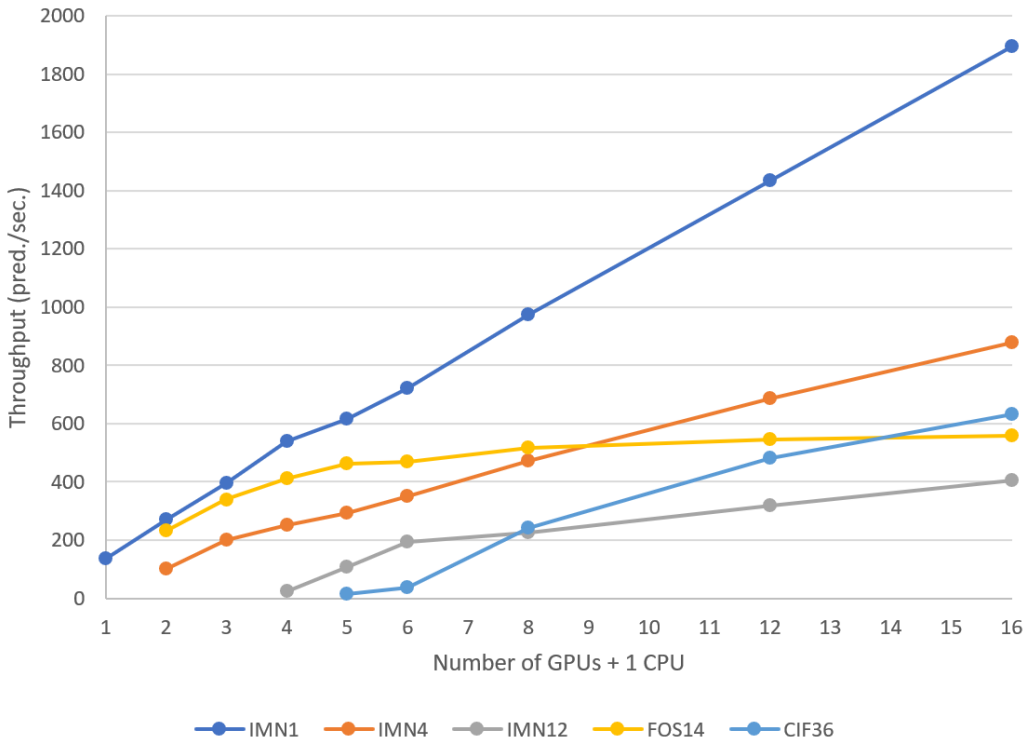

Figure 8.4 – *Weak scalability of diverse ensemble. They have been distributed and set according our smart allocator and deployed with our fast scheme. The exact numbers are shown in table 8.2.*

The *allocation optimizer* successes in automatically constructing allocation matrices. It found a feasible solution to store 12 DNNs into only 4 GPUs. On the opposite, it also shows success to leverage multiple GPUs, for example, the Resnet152 model alone gets a Weak Scaling Efficiency of 87% with 16 GPUs. Finally, when we compare algorithm 1 (Worst-Fit Decreasing) and algorithm 2 (Bounded Greedy) throughput, we observe also that the second algorithm produces generally a significant speed-up confirming the usefulness of this second algorithm.

We observe the different matrices produced. To illustrate the decision making of the *matrix allocation optimizer*, we show an example of a matrix allocation in figure 8.3.

	ResNet50	ResNet101	DenseNet121	VGG19
CPU	0	0	0	0
GPU1	8	8	0	0
GPU2	0	128	0	0
GPU3	0	0	8	0
GPU4	0	0	0	8

Table 8.3 – *Allocation matrix of IMN4 on 4 GPUs returned by our allocation matrix optimizer*

In general, the slower DNNs responsible for the performance bottleneck of the ensemble are multi-threaded in priority and their batch size is optimized. Furthermore, we observe that when some models are co-localized their batch size is often chosen smaller. Finally,

when we increase the number of GPUs, algorithm 2 quickly stops using the CPU. Indeed, the CPU is only used by Algorithm 1 and Algorithm 2 when the GPU memory is full. To accelerate a DNN, some nontrivial decisions may be also automatically taken such as:

- Launching a new thread
- Optimizing the batch size of an existing thread
- Alleviating a device by removing a thread of another DNN

In addition to measuring the throughput, we also measure the good stability of this throughput for any allocation matrix. This is a desirable property for some industrial applications where the server must guarantee a certain quality of service. More formally, the function bench(A,$fake\_data$) (algorithm 2) returned values varies to a relative standard deviation (RSD) below 2% for any A. We also measure that when the rate $max\_neighs/total\_neighs$ is low (e.g., inferior to 0.2) the bounded greedy algorithm can return diverse matrices performance at different run-time until RSD=16%.

8.4.3 Baseline comparison

We compare in table 8.4 our *allocation matrix optimizer* with the commonly used strategy we named BBS as baseline. The BBS uses n GPUs for n models and for each model it searches for the optimum batch size. It requires the same amount of GPUs as DNNs, this is a major limitation that requires small ensembles or it requires large hardware investment.

	BBS baseline		Our server	
	img/sec	# combinations	img/sec	# combinations
IMN1 / 1GPU	136	5	**136**	69
IMN4 / 4GPUs	211	20	**251**	200
IMN12 / 12GPUs	136	60	**338**	1000
”	”	”	**376**	2000

Table 8.4 – *Comparison between two allocation strategies: The simple preferred batch size strategy and our proposed allocation matrix optimizer with different ensembles (from IMN1 to IMN12) and different GPUs (+1 CPU each time). Those two strategies produce an allocation matrix for the inference system and benefit from the highly asynchronous inference system design and low bottleneck. The last line we set* $max\_iter = 20$.

8.5 Summary

This work answers this missing piece of pipelines between DNNs built by machine learning practitioners and serves them efficiently with any number of GPUs and CPUs. We enumerate multiple scenarios for inference in machine learning applications with ensembles and propose a solution for each of them. For the first time, we propose a solution to the complex allocation problem of multiple heterogenous DNNs in multiple

heterogeneous GPUs in the context of batch applications. The flexibility obtained relies on the formalism of the allocation matrix format allowing data-parallelism, co-localization, and batch size optimization.

To target efficiency, thousands of allocation matrices are assessed before selecting the best one and instantiating it. The complexity of the allocation optimization is bounded for practical utilization and the matrices may be cached to avoid recomputing the allocation optimization.

And more, the inference system is built with many asynchronous processes to avoid overhead and accelerate the combinations. In our benchmarks, we observe the smart decision of our allocation optimizer. When the number of GPUs is superior to the number of DNNs, the heavier DNNs are automatically data-parallel to avoid bottleneck performance. On the opposite, when the number of GPUs is lower, we observe automatically co-localization and smaller batch size to fit all DNNs into the memory.

Finally, the proposed inference optimization of ensembles may be widely applied to applications requiring high accuracy. Each part of the proposed pipeline is well identified to guarantee easy code adaptation to facilitate introduction in current inference servers. For example, Object Detection and classification require different combination rules. To benefit from hardware-specific optimization, changing the inference framework requires localized updates into the *predictor* process.

Algorithm 2: Worst-fit-decreasing with a priority to GPUs

input: M the list of DNN models in the ensemble, D the device set, $default\_batch\_size$

output: A the allocation matrix containing all models placed
start
$A_{m,d} \leftarrow$ 0, $m = 1...card(M)$ and $d = 1...card(D)$
M sorted in desc. order of memory size
for m in M **do**

    // AG is the matrix where m is put on the GPU-side
    $g \leftarrow$ more_remaining_memory(A,$default\_batch\_size$,'GPU')
    $AG \leftarrow$ copy(A)
    $AG_{m,g} \leftarrow default\_batch\_size$

    // AC is the matrix where m is put on the CPU-side
    $c \leftarrow$ more_remaining_memory(A,$default\_batch\_size$,'CPU')
    $AC \leftarrow$ copy(A)
    $AC_{m,c} \leftarrow default\_batch\_size$

    // Put the model m
    if fit_mem(AG) **then**
        $A \leftarrow AG$ // Priority to the GPU-side
    else if fit_mem(AC) **then**
        $A \leftarrow AC$
    else
        Error no device have enough memory
    end if
end for
return A

Conclusion and perspectives 9

In the two first chapters 1 and 2, we introduced the importance of deep learning for energy applications, and how those applications may benefit from the multi-GPU computing nodes. Then, we discussed the challenges to run those workloads on multi-GPU computing nodes.

9.1 Summary of contributions

In chapter 3, we introduce four novel deep learning applications at TotalEnergies to produce lower carbon energies developed during this thesis: microfossils image recognition for better geology understanding; biodiversity image recognition to better preserve animals located close to any industrial facility; biology image recognition for monitor the health of algae responsible of carbon capture; microgrid control automation to optimally control the power supply by reducing electricity supply cost and CO2 emission.

Afterward, we summarize some relevant common goals and challenges associated with deep learning projects. The first common point is the need to run multiple experiments to calibrate the hyperparameters such as the gradient descent optimizer and the neural architecture. The second common goal is that real applications generally require optimizing the prediction quality (e.g., accuracy, statistical robustness, uncertainty estimates, reproducibility) and computational quality (e.g., predictions per second, memory fitting, power consumption). Finally, we highlight that ensembles have the potential to overcome the prediction quality, but the computational quality should be kept under control.

In chapter 4 we detail, benchmark, and analysis the characteristics of a deep learning workload. We present some usual approaches to attempt to handle efficiently deep learning: transfer learning, parallel patterns, hyperparameter optimization, and ensemble learning. We also underline common limitations.

Due to the lack of comparative study, we enumerate some complete workflows [160] [42] [18] to build ensembles of (non-deep) machine learning algorithms. We identify those methods construct ensembles by running a sequence of abstract steps: tuning, training, and ensembling but with different underlying methods. Those works inspired ours.

Tuning and training neural networks require a hyperparameter space and an optimizer to sample this space. We show that the optimizer is always set to be a trade-off [169]

between global exploration and local exploitation. We then discuss that the more explorative optimizers have better characteristics in the context of high computing power. The more explorative ones (e.g. random search [6]) have low chance to be stuck in a local optimum, higher robustness of the random initialization, and better GPUs utilization.

One single deep neural network may take several hours to train on a modern GPU, therefore previous researchers have not performed a large-scale study on how systematically build ensembles of deep neural networks yet. Indeed, it is required to have a multi-GPUs node to handle such ensemble workflow in a reasonable amount of time. As a consequence, to compare multiple workflows, multi-nodes multi-GPUs fit this computation need.

Due to this gap, we enumerate, evaluate and compare multiple ensemble technics and assess them. We discover that simple ensemble technics provide outperform one single base neural network from a large margin. Those results confirm the trend that more complex machine learning maths models, (ensembles of medium-size models or one big model), produce better predictions but require also more computing resources. This is why optimizing the hardware utilization is strategic for those new applications.

We dedicated the chapter 5 to the deep learning parallel patterns. We first introduce them, and present the goal they are pursuing: time reduction, push back the memory limit, or accuracy gain. We measure and compare their performance under different settings, and by identifying the different phenomenons interacting together.

We conclude that parallel one big deep neural network across multiple GPUs (either model-parallelism or data-parallelism) generally fails to efficiently leverage the underlying hardware. This is due to the nature of deep neural networks to be a sequence of computing layers with heavy communications between them (vertical communication) and tensor operators in each layer (horizontal communication). And more, there is generally no possibility to efficiently interleave computation and communication due to the fact they are a sequence of layers.

We then discuss parallelism of hyperparameter optimization, training ensembles of deep neural networks, and inference with ensembles. They are three workloads made of independent DNN computing with low communications between GPUs. Therefore, it is a natural design to distribute them across multiple GPUs.

The preliminary ensemble of deep neural network results (in chapter 4), their potential computing efficiency (chapter 5), and the development of computing nodes toward highly parallel computing units motivated us to search for a systematic procedure to build an ensemble of deep neural networks. This is why, in chapter 6 and 7, we contribute by proposing novel workflows to build ensembles in both supervised deep learning and reinforcement learning for energy applications. The abstract workflow is presented in figure 9.1.

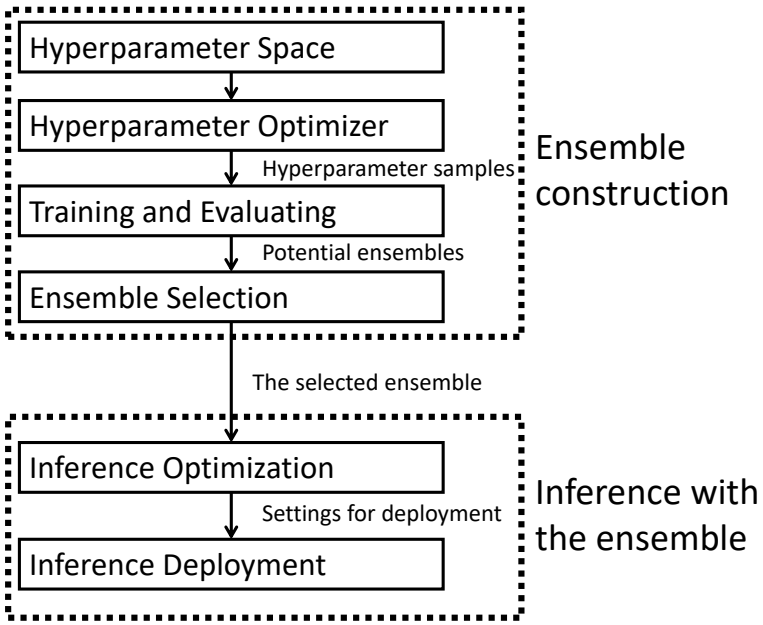

Figure 9.1 – *Abstract workflow to build and deploy ensembles. Note the combination rule function is called in "Inference Deployment" and either in"inference may be integrated with the "Ensemble Selection" (in the case of post-hoc ensembling) or the "Training and Evaluating" step (ad-hoc ensembling).*

In chapter 6 we apply it to the common deep learning case of image classification: the microfossils dataset and the CIFAR100 dataset [77]. It is the first workflow integrating hyperparameter optimization and ensemble construction in the case of deep neural networks.

We then study the expected characteristics of a good ensemble: accurate base models, complementary models, and larger ensemble size. We also show that complementary models require diversity between their predictions, but the diversity is not a sufficient condition and the interaction between models is still not known a priori. This is why ensemble selection algorithms are often formulated as combinatorial optimizers to assess many combinations and return the most promising one. We propose a new Ensemble Selection algorithm to optimize ensemble accuracy under a computing cost. We use the inference speed as a constraint but it may be easily applied to other constraints such as the power consumption, FLOPS or memory.

We also demonstrate that in the supervised case similar hyperparameter produces similar predictions using distance matrices between predictions and hierarchical clustering. This claim intuitively shows that to increase diversity exploration of diverse hyperparameters and ensembling should be integrated to build diverse models and have a chance to find complementary ones. However, in the case of very unstable base algorithms such as reinforcement (chapter 7), homogeneous ensembles produce already very diverse models and have the potential to create complementary and individually accurate models.

Multiple GPUs allow to run each step in embarrassingly parallel: hyperparameter optimization, optimize the ensemble inference, host the prediction of the ensemble. The

proposed workflow shows that it exploits and benefits from high computing power by converging high and keeping the GPU occupied. We reach state-of-the art accuracy by sampling and ensembling ResNet architectures [58] [173]. This is due to some relevant design choices: asynchronous Hyperband [91] explores the hyperparameter by keeping the GPUs occupied to train the most relevant neural networks. It allows generating a diverse, accurate and large number of base neural networks. Then, this large number of base neural networks are used to create ensembles with a controlled computing cost. Integrating the last EfficientNetV2 [159] architecture published a few months ago should outperform our previous results

In chapter 7, we confirm that the workflow is applicable in reinforcement learning. More surprisingly, we discovered that ensembles are extra-linearly more stable compared to one single RL agent. This is an important step forward in this field making experiments much more reproducible and comparable. We also notice that we deal with DDPG [174] and DQN [56] RL algorithms but those experiments may be applicable beyond.

Re-building ensembles and re-training them are computing-intensive algorithms taking GPU hours or days. However, those workloads need to be launched only sometimes when the application conditions change e.g. every week or month. Whereas, the inference phase is a long-running process delivering the prediction to the associated business client application whenever it is requested. Therefore, optimizing the inference speed is sometimes considered more strategic. [33]

This is why in chapter 8, we introduce and enumerate the four inference system types and their associated challenges. They vary according to if data come per batch or per-flow, and if the ensemble is made of heterogeneous or homogeneous deep neural networks.

We propose an algorithm to parallel and accelerate each of them. We then underline that the most complex system is "the batch inference system of heterogeneous ensembles leveraging heterogeneous devices". We then contribute by implementing and evaluating this inference system. Our approach is based on two key components: 1) We propose an optimizer to explore the space of the possible allocation matrices 2) an efficient communication scheme between processes to deploy the allocation described by the matrix. The proposed allocation matrix fits well the decision space: co-localization, inference data parallelism, and batch size settings.

We observe that the exploitation of the matrix format as decision space is very flexible and the optimizer found up to 2.8 more effective allocations compared to the baseline. The baseline consists in spreading each DNN on each GPU and maximizing the batch size of the DNNs.

9.2 Methodological perspectives

The proposed workflow is a succession of well-identified tasks it makes the workflow flexible enough to integrate new algorithms, each step is easily re-utilizable, and can be

adapted to new applications. We enumerate some open directions that we identify as a pertinent continuation of the work presented in this thesis.

9.2.1 Keep following the deep learning progress

There has been a considerable amount of research work in the direction of deep learning, such as designing novel operations, novel architectures [159], or automatic architecture exploration methodologies [89] and technics [101]. Any novelty in this field may be plugged into the proposed workflow by replacing the first 2 steps of the workflow. One must be aware that the best architecture optimization method to optimize and return one deep neural network may not be the best deep neural network aiming ensembling.

Deep learning frameworks and arithmetic libraries (such as CudNN, MKL), are quickly evolving with novelty every year. They make the deep learning applications optimize the code representation, core parallelism, and memory consumption. Maintaining this underlying software stack frequently up-to-date is essential to ensure optimal performance. However, only hardware investment allows for handling significantly larger neural networks and bigger ensembles.

9.2.2 Keep following the IA hardware progress

This thesis experiments with different workflows on multi-GPU computing nodes. However, there is recently an emergence of IA dedicated hardware for handling different IA workloads and different specific running contexts. They include but are not limited to:

- Embedded systems such as NVIDIA Jetson for cost-effectiveness and proximity with the targeted applications. IoT (Internet of Things) are embedded systems that are also connected to a network. They have a broad range of applications including facility control, microscopy images, robotics, appliance, automotive and more.
- IA dedicated computing nodes such as Google TPU pods, Cerebras CS, NVIDIA HGX. They widely differ between the number of chips in one node, the number of cores per chip, and the instruction set.

Commercialized NVIDIA GPU progress in terms of the number of cores, and available memory with reasonable, by keeping the energy consumption between 225 and 300 watts until the Hopper generation. The figure 9.2 shows the non-exhaustive list of NVIDIA GPUs for each year. The trend is favorable to the development of large deep neural networks and handling multiple of them at the same time.

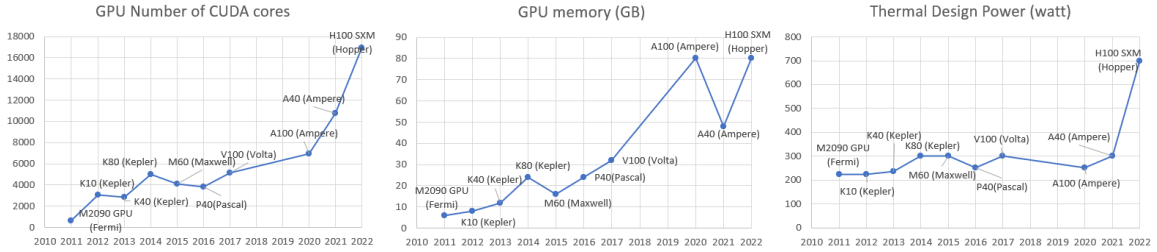

Figure 9.2 – *NVIDIA GPU evolution. For each year, we display the one with the most CUDA cores. Other characteristics such as the number of Tensor Cores, memory speed, FLOPS, confirm also the going fast trend.*

We also observe dedicated components for IA workload to attempt to handle more efficiently tensor operators of the GPUs such as the Tensor Core, NVLINK, TensorFloat. The MIG [26] (Multi-Instance GPU) may allow to better tune the co-localization of neural networks and would be a natural direction for accelerating independent tasks.

More efficient deep neural networks, more computing power, and better proximity between computing power and the applications open the door to better tuned neural networks and bigger ensembles, and new applications.

9.2.3 Faster inference of ensembles

Smarter matrix allocator with reinforcement learning Our greedy-based algorithm find up to X2.7 faster inference settings than baselines to optimize the placement of neural networks on GPUs. Reinforcement learning may take better decisions than our greedy-based approach (i.e., automatic discover mid-term strategy and long-term strategy). Therefore it is reasonable to believe it may improve again the placement of neural networks on available GPUs. Multiple times, reinforcement learning has been successful to solve this kind of (NP-Hard) optimization problems consisting in optimizing the placement of objects on a limited amount of resources: Allocating tensor computings on GPUs [107], resource provisioning in telecommunication networks [4], HPC jobs on multi-node resources [183]...

Leverage from diverse technologies Ensembles may offer the flexibility to be composed of neural networks implemented with different technologies (i.e., inference compiler, optimizer, runtime engine...). The allocation matrix optimizer (chapter 6) only optimizes the batch size and the multi-GPU allocation with only one fixed inference framework. We may generalize it beyond by optimizing the inference technology choice. For example, we now know that TensorRT is faster than ONNX-RT with larger batch size values but this is the opposite with smaller batch size values. We may reasonably hypothesize that the right inference technology properly set may have the potential to improve again the inference performance. In this case, each element of the allocation does not only contains the batch size value but a fixed-vector length of settings (i.e., batch size, inference engine framework, ...). However, more degree of freedom during the exploration of the inference settings (i.e., allocation matrix) should increase the complexity to automatically found good settings.

On the other hand, RL agents require multiple interactions with the target software to optimize it. It may take too much time to converge and obtain reasonable performance.

Only benchmarks may allow us to conclude the benefit of RL compared to the simple greedy approach (if any).

Due to the originality of our proposed inference ststem, there are no designed RL methods on the shelf to optimize the ensemble of neural networks on GPUs. This means we must invent it ourselves inspired by previous works [107] [4] [183]. We started to design one RL method with the last inference frameworks. We lack time to compare it with our proposed greedy-based approach. It will be the subject of a future paper.

9.2.4 AutoML for business logic

The invented procedure in chapter 6 and 7 allows to generate ensembles optimized for a given task. Running the workflow allows maintaining the ensemble up to date when the application context varies. For example, when the microfossils training dataset is updated or when a microgrid component is replaced.

As a consequence, the question of building a model may shift from "According to the application X how build/tune a model F to optimize the predictions Y ?" to "How tune the application X to improve the predictions Y produced by the automatically build model F ?".

Following this logic, we may for example ask ourselves how best to construct microgrids (select solar panels among different brands and characteristics, number of batteries...). We may create multiple RL environments reflecting different scenarios to compare, run the proposed workflow on each one, and observe the cumulative rewards. This new approach allows fair and automatic comparison by focusing the next investigations on the business logic rather than the machine learning models.

9.3 Application perspectives

The different ensemble constructions and deployment discussed in this thesis may be generalized to a wide range of applications. First, we enumerate some deep learning applications that may benefit from this method. Then, we show that an ensemble of estimators is useful in a lot of other scientific applications and our proposed methodologies may benefit partially from a wider range of software.

9.3.1 Generalize to other deep learning applications

Uncertainty detection. We do not perform detailed study uncertainty quality estimates in this thesis. Yet, ensembles of classifiers are known [82] [57] to produce better uncertainty estimates in comparison to one single classifier. That is to say, detect that the ensemble performs well to detect when it is unsure of its own prediction due to unobserved input data as well as complicated input data. This is an important property that allows requesting human intervention when uncertainty is detected.

In the case of neural networks for regression problems (e.g., time series forecasting), individual neural networks do not return uncertainty at all. Ensemble procedures have the potential to measure an uncertainty value based on their disagreement.

Bigger neural networks. We may integrate model parallelism while searching hyper-parameters. It may be useful for application predicting on large dataset such as 2D image segmentation [134], 3D image segmentation [24]... Therefore, it requires also adapting the proposed inference system to handle model-parallel neural networks.

Ensemble of object detection. Object detection [131] differs compared to the classification tasks by the fact that outputs multiple vectors: object localization with box coordinates, the binary probability that the box contains an object of interest, classification of the box... First combination rules [150] recently emerged in those applications based on geometrical computing to interpolate object detection outputs. We lacked time to apply it to the biodiversity project.

9.3.2 Generalize to other applications

We present a few other applications where ensemble methods have been shown useful to combine the outputs of different software. We may notice that different communities between machine learning scientists, meteorologists, and bio-computer scientists, ... often use a different vocabulary to deal with similar concepts. We use our previous vocabulary for simplicity.

For example, base applications settings and ensembling may be combined with a similar approach to improve the complementarity of base software predictions. We classify two approaches and discuss advantages and disadvantages to know if combinations should be post-hoc (i.e., store predictions on disk and found combinations) or ad-hoc (the optimization of base hyperparameter generates directly the ensemble). The combination rule was discussed in terms of advantage, disadvantage, and applicability.

Ensemble of bioinformatics software.

Some meta-methods have been proposed in bio-informatics use a combination of existing approaches to produce better outputs than one single method. This is the case of MARIO [121], a heterogeneous ensemble of 4 different genomics methods that predict together. The predictions are combined with a domain-specific combination rule. Base components utilize the default (hyper-)parameters which may not be optimal. This thesis shows that coupled hyperparameter optimization and ensemble on calibration data samples allow to optimize outcomes.

Ensemble of simulators in meteorology.

Simulators are of great importance to the energy industry. They apply computational methods aiming to improve energy production while minimizing risks and environmental impacts. They have a wide range of applications to study phenomenons, verify the hypothesis, make reliable predictions, evaluate concepts, compare, and refine them.

The literature is rich in applications of the ensemble of simulators. It especially true

in the climatology and meteorology [104] [164]. Authors propose different workflows to build an ensemble of forecasters in varying settings such as the combination rules and the number of base models.

Previously proven ensemble methods include but are not limited to:

- Homogeneous ensemble [104]. That is to say the same forecast model but different initial conditions of the simulation. Initial conditions are input from noisy observations. Like us, they have demonstrated that a larger number of base simulators (e.g. often between 30 and 200) improve the accuracy and uncertainty estimates, therefore the ensembles require smart distribution of the workload on a High-Performance Computer.

- Heterogeneous ensembles [164]. different initial conditions, different forecast models.

- Heterogeneous ensembles with weighted averaging of the predictions [164]. Note those methods require calibration data samples from past observations to compute the ensemble's weights.

Some methods proposed in this thesis may be beneficial to this context. Automatic tuning of the base models, ensemble selection with controlled computing cost, and the systematic optimization of the ensemble's allocation on multiple processors.

Bibliography

[1] Martıén Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283. ISBN: 9781931971331.

[2] Deepak Agarwal et al. "LASER: A scalable response prediction platform for online advertising". In: Feb. 2014, pp. 173–182. DOI: 10.1145/2556195.2556252.

[3] Ashvin Agrawal et al. "Cloudy with high chance of DBMS: a 10-year prediction for Enterprise-Grade ML". In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. 2020.

[4] Jose Jurandir Alves Esteves et al. "DRL-based Slice Placement Under Non-Stationary Conditions". In: *CNSM 2021 - 17th International Conference on Network and Service Management*. Izmir, Turkey, Oct. 2021. URL: https://hal.inria.fr/hal-03332502.

[5] James S. Bergstra et al. "Algorithms for Hyper-Parameter Optimization". In: *Advances in Neural Information Processing Systems 24*. Ed. by J. Shawe-Taylor et al. Curran Associates, Inc., 2011, pp. 2546–2554. URL: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[6] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization." In: *J. Mach. Learn. Res.* 13 (2012), pp. 281–305. URL: http://dblp.uni-trier.de/db/journals/jmlr/jmlr13.html#BergstraB12.

[7] Andrew Bock and Ione Fine. "Anatomical and Functional Plasticity in Early Blind Individuals and the Mixture of Experts Architecture". In: *Frontiers in human neuroscience* 8 (Dec. 2014), p. 971. DOI: 10.3389/fnhum.2014.00971.

[8] Tossapon Boongoen and Natthakan Iam-On. "Cluster ensembles: A survey of approaches with recent extensions and applications". In: *Computer Science Review* 28 (2018), pp. 1–25. DOI: 10.1016/j.cosrev.2018.01.003. URL: https://app.dimensions.ai/details/publication/pub.1100959500.

[9] Leo Breiman. "Bagging Predictors". In: *Machine Learning* 24.2 (Aug. 1996), pp. 123–140. ISSN: 1573-0565. DOI: 10.1023/A:1018054314350. URL: https://doi.org/10.1023/A:1018054314350.

[10] Leo Breiman. "Random Forests". In: *Mach. Learn.* 45.1 (2001), pp. 5–32. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

[11] Gavin Brown et al. "Diversity Creation Methods: A Survey And Categorisation". In: *Information Fusion* 6 (Mar. 2005), pp. 5–20. DOI: 10.1016/j.inffus.2004.04.004.

[12] Lucian Busoniu, Robert Babuska, and Bart De Schutter. "Multi-agent Reinforcement Learning: An Overview". In: vol. 310. July 2010, pp. 183–221. ISBN: 978-3-642-14434-9. DOI: 10.1007/978-3-642-14435-6_7.

[13] Rich Caruana, Art Munson, and Alexandru Niculescu-Mizil. "Getting the Most Out of Ensemble Selection". In: Dec. 2006, pp. 828–833. DOI: 10.1109/ICDM.2006.76.

[14] Rich Caruana et al. "Ensemble Selection from Libraries of Models". In: *Proceedings of the Twenty-First International Conference on Machine Learning*. ICML '04. Banff, Alberta, Canada: Association for Computing Machinery, 2004, p. 18. ISBN: 1581138385. DOI: 10.1145/1015330.1015432. URL: https://doi.org/10.1145/1015330.1015432.

[15] Gerardo Ceballos et al. "Accelerated Modern Human-Induced Species Losses: Entering the Sixth Mass Extinction." In: *Science Advances* 1 (June 2015), e1400253. DOI: 10.1126/sciadv.1400253.

[16] Zachary Charles and Jakub Konečný. "Convergence and Accuracy Trade-Offs in Federated Learning and Meta-Learning". In: *The 24th International Conference on Artificial Intelligence and Statistics, AISTATS 2021, April 13-15, 2021, Virtual Event*. 2021, pp. 2575–2583. URL: http://proceedings.mlr.press/v130/charles21a.html.

[17] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*. 2016, 262–263.

[18] Boyuan Chen et al. "Autostacker: A Compositional Evolutionary Learning System". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '18. Kyoto, Japan: Association for Computing Machinery, 2018, pp. 402–409. ISBN: 9781450356183. DOI: 10.1145/3205455.3205586. URL: https://doi.org/10.1145/3205455.3205586.

[19] Jianmin Chen et al. "Revisiting Distributed Synchronous SGD". In: *International Conference on Learning Representations Workshop Track*. 2016. URL: https://arxiv.org/abs/1604.00981.

[20] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785.

[21] Ting Chen et al. "Big Self-Supervised Models are Strong Semi-Supervised Learners". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 22243–22255. URL: https://proceedings.neurips.cc/paper/2020/file/fcbc95ccdd551da181207c0c1400c655-Paper.pdf.

[22] Zhi Chen et al. "Class-Imbalanced Deep Learning via a Class-Balanced Ensemble". In: *IEEE Transactions on Neural Networks and Learning Systems* (2021), pp. 1–15. DOI: 10.1109/TNNLS.2021.3071122.

[23] François Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 1800–1807.

[24] Özgün Çiçek et al. "3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation". In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2016*. Springer International Publishing, 2016, pp. 424–432. DOI: 10.1007/978-3-319-46723-8_49. URL: https://doi.org/10.1007%5C%2F978-3-319-46723-8_49.

[25] Marius Cordts et al. "The Cityscapes Dataset for Semantic Urban Scene Understanding". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[26] NVIDIA Corporation. *Featured Demo: Running Multiple Workloads on a Single A100 GPU*. ISC High Performance 2020. Frankfurt, Germany, June 2020.

[27] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "BinaryConnect: Training Deep Neural Networks with binary weights during propagations". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/3e15cc11f979ed25912dff5b0669f2cd-Paper.pdf.

[28] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 613–627. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

[29] Mike Davies et al. "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning". In: *IEEE Micro* PP (Jan. 2018), pp. 1–1. DOI: 10.1109/MM.2018.112130359.

[30] Pooya Davoodi et al. ""TensorRT inference With TensorFlow"". In: GPU Technology Conference. 2019.

[31] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[32] J. Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *NAACL*. 2019.

[33] Dominic Divakaruni, Peter Jones, Sudipta Sengupta, Liviu Calin. "Amazon Elastic Inference: Reduce Learning Inference Cost". In: 2018.

[34] Xuanyi Dong and Yi Yang. "One-Shot Neural Architecture Search via Self-Evaluated Template Network". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2019.

[35] Xuanyi Dong and Yi Yang. "Searching for a Robust Neural Architecture in Four GPU Hours". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.

[36] Daniel L. Elliott and Charles Anderson. "The Wisdom of the Crowd: Reliable Deep Reinforcement Learning Through Ensembles of Q-Functions". In: *IEEE Transactions on Neural Networks and Learning Systems* (2021), pp. 1–9. DOI: 10.1109/TNNLS.2021.3089425.

[37] Logan Engstrom et al. "Implementation Matters in Deep RL: A Case Study on PPO and TRPO". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=r1etN1rtPB.

[38] Dumitru Erhan et al. "The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training". In: *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*. Ed. by David van Dyk and Max Welling. Vol. 5. Proceedings of Machine Learning Research. Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA: PMLR, Apr. 2009, pp. 153–160. URL: https://proceedings.mlr.press/v5/erhan09a.html.

[39] Lasse Espeholt et al. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1406–1415. URL: http://proceedings.mlr.press/v80/espeholt18a.html.

[40] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231.

[41] M. Everingham et al. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[42] Matthias Feurer et al. "Efficient and Robust Automated Machine Learning". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 2962–2970. URL: http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf.

[43] Max Finlayson et al. "Millennium Ecosystem Assessment: Ecosystems and human well-being: wetlands and water synthesis". In: (Jan. 2005).

[44] Mike Folk, Albert Cheng, and Kim Yates. "HDF5: A file format and I/O library for high performance computing applications". In: *Proceedings of Supercomputing*. Vol. 99. 1999, pp. 5–33.

[45] Kunihiko Fukushima. "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position". In: *Biological Cybernetics* 36 (1980), pp. 193–202.

[46] Michael Gashler, Christophe Giraud-Carrier, and Tony Martinez. "Decision Tree Ensemble: Small Heterogeneous Is Better Than Large Homogeneous". In: *2008 Seventh International Conference on Machine Learning and Applications* (Jan. 2008), pp. 900–905. DOI: 10.1109/ICMLA.2008.154.

[47] Yi Ge and Monique Jones. "Inference With Intel". In: AI DevCon 2018. 2018.

[48] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: (2010).

[49] Hui Guan et al. "FLEET: Flexible Efficient Ensemble Training for Heterogeneous Deep Neural Networks". In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 247–261. URL: https://proceedings.mlsys.org/paper/2020/file/ed3d2c21991e3bef5e069713af9fa6ca-Paper.pdf.

[50] Joris Guérin et al. "CNN Features are also Great at Unsupervised Classification". In: Feb. 2018, pp. 83–95. DOI: 10.5121/csit.2018.80308.

[51] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[52] Suyog Gupta et al. "Deep Learning with Limited Numerical Precision". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 1737–1746.

[53] Isabelle Guyon et al. "Analysis of the AutoML Challenge Series 2015–2018". In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Cham: Springer International Publishing, 2019, pp. 177–219. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_10. URL: https://doi.org/10.1007/978-3-030-05318-5_10.

[54] Song Han et al. "Learning Both Weights and Connections for Efficient Neural Networks". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 1135–1143.

[55] Sherif Hashem. "Optimal Linear Combinations of Neural Networks". In: *Neural Netw.* 10.4 (June 1997), pp. 599–614. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(96)00098-6. URL: https://doi.org/10.1016/S0893-6080(96)00098-6.

[56] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, 2016, pp. 2094–2100.

[57] Marton Havasi et al. "Training independent subnetworks for robust prediction". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=OGg9XnKxFAH.

[58] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[59] Peter Henderson et al. "Deep Reinforcement Learning that Matters". In: *AAAI*. 2018.

[60] Gonzague Henri et al. "pymgrid: An Open-Source Python Microgrid Simulator for Applied Artificial Intelligence Research". In: *NeurIPS 2020 Workshop on Tackling Climate Change with Machine Learning*. 2020. URL: https://www.climatechange.ai/papers/neurips2020/3.html.

[61] Robin Henry and Damien Ernst. "Gym-ANM: Open-source software to leverage reinforcement learning for power system management in research and education". In: *Software Impacts* 9 (2021), p. 100092. ISSN: 2665-9638. DOI: https://doi.org/10.1016/j.simpa.2021.100092. URL: https://www.sciencedirect.com/science/article/pii/S2665963821000348.

[62] Torsten Hoefler. *Twelve ways to fool the masses when reporting performance of deep learning workloads*. Workshop III: HPC for Computationally and Data-Intensive Problems. Los Angeles, CA, Nov. 2018.

[63] Matthew Hoffman, Eric Brochu, and Nando de Freitas. "Portfolio Allocation for Bayesian Optimization". In: *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*. UAI'11. Barcelona, Spain: AUAI Press, 2011, pp. 327–336. ISBN: 9780974903972.

[64] Dan Horgan et al. "Distributed Prioritized Experience Replay". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=H1Dy---0Z.

[65] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90009-T. URL: https://www.sciencedirect.com/science/article/pii/089360809190009T.

[66] Gao Huang et al. "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 2261–2269. DOI: 10.1109/CVPR.2017.243.

[67] Gao Huang et al. "Snapshot Ensembles: Train 1, Get M for Free". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=BJYwwY9ll.

[68] Yanping Huang et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach et al. 2019, pp. 103–112. URL: https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html.

[69] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "Sequential model-based optimization for general algorithm configuration". In: *International conference on learning and intelligent optimization*. Springer. 2011, pp. 507–523.

[70] Kevin Jamieson and Ameet Talwalkar. "Non-stochastic Best Arm Identification and Hyperparameter Optimization". In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. Ed. by Arthur Gretton and Christian C. Robert. Vol. 51. Proceedings of Machine Learning Research. Cadiz, Spain: PMLR, Sept. 2016, pp. 240–248. URL: http://proceedings.mlr.press/v51/jamieson16.html.

[71] Feibo Jiang et al. "Distributed Resource Scheduling for Large-Scale MEC Systems: A Multi-Agent Ensemble Deep Reinforcement Learning with Imitation Acceleration". In: *IEEE Internet of Things Journal* (2021), pp. 1–1. DOI: 10.1109/JIOT.2021.3113872.

[72] Haifeng Jin, Qingquan Song, and Xia Hu. "Auto-Keras: An Efficient Neural Architecture Search System". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 1946–1956. ISBN: 9781450362016. DOI: 10.1145/3292500.3330648. URL: https://doi.org/10.1145/3292500.3330648.

[73] Travis Johnston et al. "Optimizing Convolutional Neural Networks for Cloud Detection". In: *Proceedings of the Machine Learning on HPC Environments*. MLHPC'17. Denver, CO, USA: Association for Computing Machinery, 2017. ISBN: 9781450351379. DOI:

10.1145/3146347.3146352. URL: https://doi.org/10.1145/3146347.3146352.

[74] Kate E. Jones et al. "Global trends in emerging infectious diseases". In: *Nature* 451.7181 (Feb. 2008), pp. 990–993. ISSN: 1476-4687. DOI: 10.1038/nature06536. URL: https://doi.org/10.1038/nature06536.

[75] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (Aug. 2021), pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2. URL: https://doi.org/10.1038/s41586-021-03819-2.

[76] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (Dec. 2014).

[77] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[78] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[79] Ludmila I. Kuncheva and Christopher J. Whitaker. "Measures of Diversity in Classifier Ensembles and Their Relationship with the Ensemble Accuracy." In: *Mach. Learn.* 51.2 (2003), pp. 181–207. URL: http://dblp.uni-trier.de/db/journals/ml/ml51.html#KunchevaW03.

[80] Ludmila Kuncheva and Chris Whitaker. "Measures of Diversity in Classifier Ensembles and Their Relationship with the Ensemble Accuracy". In: *Machine Learning* 51 (May 2003), pp. 181–207. DOI: 10.1023/A:1022859003006.

[81] Thanard Kurutach et al. "Model-Ensemble Trust-Region Policy Optimization". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=SJJinbWRZ.

[82] Balaji Lakshminarayanan, A. Pritzel, and C. Blundell. "Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles". In: *NIPS*. 2017.

[83] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '21. Virtual Event, Republic of Korea: IEEE Press, 2021, pp. 2–14. ISBN: 9781728186139. DOI: 10.1109/CGO51591.2021.9370308. URL: https://doi.org/10.1109/CGO51591.2021.9370308.

[84] Tung D. Le et al. "Automatic GPU Memory Management for Large Neural Models in TensorFlow". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1–13. ISBN: 9781450367226. DOI: 10.1145/3315573.3329984. URL: https://doi.org/10.1145/3315573.3329984.

[85] Chris Leary and Todd Wang. "XLA: Tensorflow, Compiled!" In: *Tensorflow developer summit*. 2017.

[86] Yann Lecun et al. "Gradient-based learning applied to document recognition". In: (1998), pp. 2278–2324.

[87] Erin LeDell and Sebastien Poirier. "H2O AutoML: Scalable Automatic Machine Learning". In: *7th ICML Workshop on Automated Machine Learning (AutoML)* (July 2020). URL: https://www.automl.org/wp-content/uploads/2020/07/AutoML_2020_paper_61.pdf.

[88] Ang Li et al. "A Generalized Framework for Population Based Training". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 1791–1799. ISBN: 9781450362016. DOI: 10.1145/3292500.3330649. URL: https://doi.org/10.1145/3292500.3330649.

[89] Chaojian Li et al. "{HW}-{NAS}-Bench: Hardware-Aware Neural Architecture Search Benchmark". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=_0kaDkv3dVf.

[90] Liam Li and Ameet Talwalkar. "Random Search and Reproducibility for Neural Architecture Search". In: *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*. Ed. by Ryan P. Adams and Vibhav Gogate. Vol. 115. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 367–377. URL: http://proceedings.mlr.press/v115/li20c.html.

[91] Liam Li et al. "A System for Massively Parallel Hyperparameter Tuning". In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 230–246. URL: https://proceedings.mlsys.org/paper/2020/file/f4b9ec30ad9f68f89b29639786cb62ef-Paper.pdf.

[92] Liam Li et al. "Geometry-Aware Gradient Algorithms for Neural Architecture Search". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=MuSYkd1hxRP.

[93] Lisha Li et al. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization". In: 18.1 (Jan. 2017), pp. 6765–6816. ISSN: 1532-4435.

[94] Mingzhen Li et al. "The Deep Learning Compiler: A Comprehensive Survey". In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2021), pp. 708–727. DOI: 10.1109/TPDS.2020.3030548.

[95] Mu Li et al. "Scaling Distributed Machine Learning with the Parameter Server". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 583–598. ISBN: 9781931971164.

[96] Eric Liang. *Scalable Reinforcement Learning Systems and their Applications*. Tech. rep. EECS, university of California, Berkley, 2021.

[97] Yuansong Liao and John Moody. "Constructing Heterogeneous Committees Using Input Feature Grouping: Application to Economic Forecasting". In: NIPS'99 (1999), pp. 921–927.

[98] Richard Liaw. "A Guide to Modern Hyperparameters Turning Algorithms". In: *PyData Los Angeles*. 2019.

[99] Tsung-Yi Lin et al. "Feature Pyramid Networks for Object Detection". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 936–944. DOI: 10.1109/CVPR.2017.106.

[100] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 740–755. ISBN: 978-3-319-10602-1.

[101] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "DARTS: Differentiable Architecture Search". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=S1eYHoC5FX.

[102] Y. Liu and X. Yao. "Ensemble learning via negative correlation". In: *Neural Networks* 12.10 (1999), pp. 1399–1404. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(99)00073-8. URL: http://www.sciencedirect.com/science/article/pii/S0893608099000738.

[103] Scott M Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: (2017). Ed. by I. Guyon et al., pp. 4765–4774. URL: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

[104] N. Maher, S. Milinski, and R. Ludwig. "Large ensemble climate model simulations: introduction, overview, and future prospects for utilising multiple types of large ensemble". In: *Earth System Dynamics* 12.2 (2021), pp. 401–418. DOI: 10.5194/esd-12-401-2021. URL: https://esd.copernicus.org/articles/12/401/2021/.

[105] Rowan McAllister et al. "Concrete Problems for Autonomous Vehicle Safety: Advantages of Bayesian Deep Learning". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 2017, pp. 4745–4753. DOI: 10.24963/ijcai.2017/661. URL: https://doi.org/10.24963/ijcai.2017/661.

[106] Michael J. Fischer. "Thesis (Ph. D.)–Massachusetts Institute of Technology, Dept. of Mathematics". In: (1973).

[107] Azalia Mirhoseini et al. "Device Placement Optimization with Reinforcement Learning". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 2430–2439.

[108] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1928–1937. URL: https://proceedings.mlr.press/v48/mniha16.html.

[109] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[110] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 561–577. ISBN: 9781931971478.

[111] Anusha Nagabandi et al. "Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning". In: May 2018, pp. 7559–7566. DOI: 10.1109/ICRA.2018.8463189.

[112] Arun Nair et al. "Massively Parallel Methods for Deep Reinforcement Learning". In: (2015).

[113] Arun Nair et al. *Massively Parallel Methods for Deep Reinforcement Learning*. ICML deep learning workshop, Lille, France, 2015. 2015. URL: http://arxiv.org/abs/1507.04296.

[114] Deepak Narayanan et al. "PipeDream: Generalized Pipeline Parallelism for DNN Training". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 1–15. ISBN: 9781450368735. DOI: 10.1145/3341301.3359646. URL: https://doi.org/10.1145/3341301.3359646.

[115] *NVIDIA DGX-1 With Tesla V100 System Architecture*. Tech. rep. NVIDIA Corporation, 2017.

[116] Junhyuk Oh et al. "Action-Conditional Video Prediction Using Deep Networks in Atari Games". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 2863–2871.

[117] Deniz Oktay et al. "Scalable Model Compression by Entropy Penalized Reparameterization". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=HkgxW0EYDS.

[118] Randal S. Olson et al. "Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO '16. Denver, Colorado, USA: ACM, 2016, pp. 485–492. ISBN: 978-1-4503-4206-3. DOI: 10.1145/2908812.2908918. URL: http://doi.acm.org/10.1145/2908812.2908918.

[119] Christopher Olston et al. "TensorFlow-Serving: Flexible, High-Performance ML Serving". In: Workshop on ML Systems at NIPS 2017. 2017.

[120] Robert M. Patton et al. "Exascale Deep Learning to Accelerate Cancer Research". In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 1488–1496. DOI: 10.1109/BigData47090.2019.9006467.

[121] Cécile Pereira, Alain Denise, and Olivier Lespinet. "A meta-approach for improving the prediction and the functional annotation of ortholog groups". In: *BMC Genomics* 15.6 (Oct. 2014), S16. ISSN: 1471-2164. DOI: 10.1186/1471-2164-15-S6-S16. URL: https://doi.org/10.1186/1471-2164-15-S6-S16.

[122] A.T.D. Perera and Parameswaran Kamalaruban. "Applications of reinforcement learning in energy systems". In: *Renewable and Sustainable Energy Reviews* 137 (2021), p. 110618. ISSN: 1364-0321. DOI: https://doi.org/10.1016/j.rser.2020.110618. URL: https://www.sciencedirect.com/science/article/pii/S1364032120309023.

[123] Hieu Pham et al. "Efficient Neural Architecture Search via Parameters Sharing". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 4095–4104. URL: https://proceedings.mlr.press/v80/pham18a.html.

[124] Pierrick Pochelu, Serge G. Petiton, and Bruno Conche. "A Deep Neural Networks Ensemble Workflow from Hyperparameter Search to Inference Leveraging GPU Clusters". In: *International Conference on High Performance Computing in Asia-Pacific Region*. HPCAsia2022. Virtual Event, Japan: Association for Computing Machinery, 2022, pp. 61–71. ISBN: 9781450384988. DOI: 10.1145/3492805.3492819. URL: https://doi.org/10.1145/3492805.3492819.

[125] Pierrick Pochelu, Serge Petiton, and Bruno Conche. "An efficient and flexible inference system for serving heterogeneous ensembles of deep neural networks". In: *IEEE BigData 2021*. Virtual, United States, Dec. 2021. URL: https://hal.archives-ouvertes.fr/hal-03494431.

[126] Pierrick Pochelu et al. "Weakly Supervised Faster-RCNN+FPN to classify animals in camera trap images". In: Virtual, Singapore, Mar. 2022. ISBN: 978-1-4503-8741-5/22/03. DOI: 10.1145/3531232.3531235.

[127] Lutz Prechelt. "Early Stopping-But When?" In: *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. Berlin, Heidelberg: Springer-Verlag, 1998, pp. 55–69. ISBN: 3540653112.

[128] Ning Qian. "On the momentum term in gradient descent learning algorithms." In: *Neural Networks* 12.1 (1999), pp. 145–151. URL: http://dblp.uni-trier.de/db/journals/nn/nn12.html#Qian99.

[129] Esteban Real et al. "Large-Scale Evolution of Image Classifiers". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 2902–2911.

[130] Vijay Janapa Reddi et al. "MLPerf Inference Benchmark". In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ISCA '20. Virtual Event: IEEE Press, 2020, pp. 446–459. ISBN: 9781728146614. DOI: 10.1109/ISCA45697.2020.00045. URL: https://doi.org/10.1109/ISCA45697.2020.00045.

[131] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39 (2015), pp. 1137–1149.

[132] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1135–1144. ISBN: 9781450342322. DOI: 10.1145/2939672.2939778. URL: https://doi.org/10.1145/2939672.2939778.

[133] Matthew Rocklin. "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: (2015). Ed. by Kathryn Huff and James Bergstra, pp. 126–132. DOI: 10.25080/Majora-7b98e3ed-013.

[134] O. Ronneberger, P.Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Vol. 9351. LNCS. (available on arXiv:1505.04597 [cs.CV]). Springer, 2015, pp. 234–241. URL: http://lmb.informatik.uni-freiburg.de/Publications/2015/RFB15a.

[135] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[136] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[137] Mark Sandler et al. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[138] Dmytro Dzhulgakov Sarah Bird. "ONNX". In: *Workshop NIPS2017*. 2017.

[139] J. Schreiber et al. "Coopetitive Soft Gating Ensemble". In: *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2018, pp. 190–197. DOI: 10.1109/FAS-W.2018.00046. URL: https://doi.ieeecomputersociety.org/10.1109/FAS-W.2018.00046.

[140] Holger Schwenk and Y. Bengio. "Boosting Neural Networks". In: *Neural computation* 12 (Sept. 2000), pp. 1869–87. DOI: 10.1162/089976600300015178.

[141] Cook Shane. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780124159334.

[142] Rahul Sharma. "Amazon Elastic Inference: Reduce Deep Learning Inference Cost". In: GPU Technology Conference. 2019.

[143] Connor Shorten and Taghi M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6.1 (July 2019), p. 60. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0197-0. URL: https://doi.org/10.1186/s40537-019-0197-0.

[144] Li Shuangfeng. "TensorFlow Lite: On-Device Machine Learning Framework". In: *Journal of Computer Research and Development* 57.9, 1839 (2020), p. 1839. DOI: 10.7544/issn1000-1239.2020.20200291. URL: https://crad.ict.ac.cn/EN/abstract/article_4251.shtml.

[145] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[146] K. K. Singh and Y. J. Lee. "You Reap What You Sow: Using Videos to Generate High Precision Object Proposals for Weakly-Supervised Object Detection". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 9406–9414. DOI: 10.1109/CVPR.2019.00964.

[147] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012, pp. 2951–2959. URL: https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf.

[148] Marina Sokolova and Guy Lapalme. "A systematic analysis of performance measures for classification tasks". In: *Information Processing and Management* 45 (July 2009), pp. 427–437. DOI: 10.1016/j.ipm.2009.03.002.

[149] Peter Sollich and Anders Krogh. "Learning with ensembles: How overfitting can be useful". In: *NIPS*. 1995.

[150] R. Solovyev, Weimin Wang, and Tatiana Gabruseva. "Weighted boxes fusion: Ensembling boxes from different object detection models". In: *Image Vis. Comput.* 107 (2021), p. 104117.

[151] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435.

[152] Christoph Stöckl and Wolfgang Maass. "Optimized spiking neurons can classify images with high accuracy through temporal coding with two spikes". In: *Nature Machine Intelligence* 3 (Mar. 2021). DOI: 10.1038/s42256-021-00311-4.

[153] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Proceedings of the 12th International Conference on Neural Information Processing Systems*. NIPS'99. Denver, CO: MIT Press, 1999, pp. 1057–1063.

[154] C. Szegedy et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594. URL: https://ieeexplore.ieee.org/document/7298594.

[155] C. Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2016, pp. 2818–2826. DOI: 10.1109/CVPR.2016.308. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.308.

[156] Christian Szegedy et al. "Inception-v4, inception-ResNet and the Impact of Residual Connections on Learning". In: AAAI'17 (2017), pp. 4278–4284. URL: http://dl.acm.org/citation.cfm?id=3298023.3298188.

[157] Chuanqi Tan et al. "A Survey on Deep Transfer Learning: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4–7, 2018, Proceedings, Part III". In: Oct. 2018, pp. 270–279. ISBN: 978-3-030-01423-0. DOI: 10.1007/978-3-030-01424-7_27.

[158] Mingxing Tan and Quoc Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 6105–6114. URL: https://proceedings.mlr.press/v97/tan19a.html.

[159] Mingxing Tan and Quoc Le. "EfficientNetV2: Smaller Models and Faster Training". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 10096–10106. URL: https://proceedings.mlr.press/v139/tan21a.html.

[160] Chris Thornton et al. "Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms". In: *KDD* (Aug. 2012). DOI: 10.1145/2487575.2487629.

[161] A. Traue et al. "Toward a Reinforcement Learning Environment Toolbox for Intelligent Electric Motor Control". In: *IEEE Transactions on Neural Networks and Learning Systems* (2020), pp. 1–10. DOI: 10.1109/TNNLS.2020.3029573.

[162] Grigorios Tsoumakas, Ioannis Partalas, and I. Vlahavas. "A Taxonomy and Short Review of Ensemble Selection". In: *ECAI 2008, Workshop on Supervised and Unsupervised Ensemble Methods and Their Applications* (Jan. 2008).

[163] Ujval Kapasi and Elif Albuz and Philippe Vandermersch and Nathan Whitehead and Frank Jargstorff. "NVIDIA CUDA Libraries". In: GPU Technology Conference. San Jose Convention Center, Sept. 2010.

[164] L. Vandenbulcke et al. "Super-ensemble techniques: Application to surface drift prediction". In: *Progress in Oceanography* 82.3 (2009), pp. 149–167. ISSN: 0079-6611. DOI: https://doi.org/10.1016/j.pocean.2009.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0079661109000500.

[165] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. "Improving the speed of neural networks on CPUs". In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*. 2011.

[166] E. M. Voorhees. "Implementing Agglomerative Hierarchical Clustering Algorithms for Use in Document Retrieval". In: *Information Processing and Management* 22 (1986), pp. 465–476.

[167] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2016, pp. 1995–2003.

[168] Y. Wei et al. "Object Region Mining with Adversarial Erasing: A Simple Classification to Semantic Segmentation Approach". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 6488–6496. DOI: 10.1109/CVPR.2017.687.

[169] D. H. Wolpert and W. G. Macready. "No Free Lunch Theorems for Optimization". In: *Trans. Evol. Comp* 1.1 (Apr. 1997), pp. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893. URL: https://doi.org/10.1109/4235.585893.

[170] David H. Wolpert. "Stacked Generalization". In: *Neural Networks* 5 (1992), pp. 241–259.

[171] Xingfu Wu et al. "Performance, Power, and Scalability Analysis of the Horovod Implementation of the CANDLE NT3 Benchmark on the Cray XC40 Theta." In: In Proceedings of the 8th Workshop on Python for High-Performance and Scientific Computing. 2018.

[172] Fanyi Xiao and Yong Jae Lee. "Track and Segment: An Iterative Unsupervised Approach for Video Object Proposals". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 933–942. DOI: 10.1109/CVPR.2016.107.

[173] Saining Xie et al. "Aggregated Residual Transformations for Deep Neural Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 5987–5995. DOI: 10.1109/CVPR.2017.634.

[174] Tianbing Xu et al. "Learning to Explore via Meta-Policy Gradient". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 5463–5472. URL: https://proceedings.mlr.press/v80/xu18d.html.

[175] Tianhao Xu. ""Deep into Triton Inference Server: BERT Practical Deployment on NVIDIA GPU"". In: GPU Technology Conference. 2020.

[176] Sun Yanan et al. "Automatically Designing CNN Architectures Using the Genetic Algorithm for Image Classification". In: *IEEE Transactions on Cybernetics* 50.9 (2020), pp. 3840–3854. DOI: 10.1109/TCYB.2020.2983860.

[177] Yang You et al. "ImageNet Training in Minutes". In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018. Eugene, OR, USA: Association for Computing Machinery, 2018. ISBN: 9781450365109. DOI: 10.1145/3225058.3225069. URL: https://doi.org/10.1145/3225058.3225069.

[178] Cliff Young. *Codesign in Google TPUs: Inference, Training, Performance, and Scalability*. Keynote speech, Processor Conference. Oct. 2018.

[179] Fisher Yu and Vladlen Koltun. "Multi-Scale Context Aggregation by Dilated Convolutions". In: *International Conference on Learning Representations (ICLR)*. May 2016.

[180] Z. Yu et al. "Joint Learning of Saliency Detection and Weakly Supervised Semantic Segmentation". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019, pp. 7222–7232. DOI: 10.1109/ICCV.2019.00732.

[181] Sergey Zagoruyko and Nikos Komodakis. "Wide Residual Networks". In: *Proceedings of the British Machine Vision Conference 2016 (BMVC 2016)*. Ed. by Richard C. Wilson, Edwin R. Hancock, and William A. P. Smith. BMVA Press, 2016. URL: http://www.bmva.org/bmvc/2016/papers/paper087/index.html.

[182] Baohe Zhang et al. "On the Importance of Hyperparameter Optimization for Model-based Reinforcement Learning". In: *AISTATS* (2021).

[183] Di Zhang et al. "RLScheduler: An Automated HPC Batch Job Scheduler Using Reinforcement Learning". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[184] Xiangyu Zhang et al. "Accelerating Very Deep Convolutional Networks for Classification and Detection". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 38.10 (Oct. 2016), pp. 1943–1955. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2015.2502579. URL: https://doi.org/10.1109/TPAMI.2015.2502579.

[185] Zhi Zhang et al. "Animal detection from highly cluttered natural scenes using spatiotemporal object region proposals and patch verification". In: *IEEE Transactions on Multimedia* 18.10 (2016), pp. 2079–2092.

[186] Ruizhe Zhao et al. "Reducing Underflow in Mixed Precision Training by Gradient Scaling". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. Ed. by Christian Bessiere. Main track. International Joint Conferences on Artificial Intelligence Organization, July 2020, pp. 2922–2928. DOI: 10.24963/ijcai.2020/404. URL: https://doi.org/10.24963/ijcai.2020/404.

[187] Barret Zoph et al. "Learning Transferable Architectures for Scalable Image Recognition". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), pp. 8697–8710.