POLITECNICO DI TORINO

Master degree course in Artificial Intelligence and Data Analytics

Master Degree Thesis

Combining coarse- and fine-grained DNAS for TinyML

Politecnico di Torino

Supervisors
Prof. Daniele Jahier Pagliari
Dr. Matteo Risso
Dr. Alessio Burrello

Candidate
Pietro Borgaro
matricola: 292501

Academic Year 2022-2023

Summary

Artificial Intelligence (AI), and especially Machine Learning (ML) and Deep Learning (DL) have become increasingly important for Internet of Things (IoT) devices, as they provide the capability to process and analyze vast amounts of data collected directly from the device itself. With AI, IoT devices can make real-time decisions, detect patterns and anomalies, and improve their performance over time. However, the deployment of big Deep Neural Networks (DNNs) for IoT devices presents several challenges due to limited computational resources. In particular, IoT nodes are characterized by tight power, latency, and memory constraints. Neural Architecture Search (NAS) has emerged as an effective approach to optimize the process of searching and selecting DL models. NAS involves using machine learning to automatically search for the best neural network architecture for a given task, reducing the need for manual tuning and improving efficiency. Early NAS techniques were based on Reinforcement Learning or Evolutionary Computing and even if effective, they required to train each architecture explored and thus incurred long training times even with powerful GPUs. To overcome this issue Differentiable NAS (DNAS) techniques were presented. In DNAS, the architecture search process is made differentiable, allowing it to be optimized using gradient-based techniques like backpropagation and enabling the optimization of the network's weights jointly with the architecture. DNAS methods are being used to quickly explore different trade-offs between accuracy and some other metrics such as the number of parameters or the number of floating point operations in order to optimize DL models and enable their deployment on resource-constrained devices. As a result, NAS methods represent a first-class solution to accelerate the development of AI for IoT devices, enabling them to perform increasingly complex tasks in many fields such as Computer Vision (CV), Natural Language Processing (NLP), and Speech Recognition with greater accuracy and speed.

This thesis work focuses on extending the DNAS methods available in the

PLiNIO library by developing a new algorithm called PITSuperNet. PLiNIO stands for **P**lug-and-play **Li**ghtweight **N**eural **I**nference **O**ptimization and aims to aggregate in a single user-friendly library multiple Differentiable NAS methods to allow users to easily compare their effectiveness on their specific tasks and search spaces. The first and seminal algorithm included in PLiNIO is PIT (Pruning In Time) a NAS algorithm that tries to reduce the complexity of a seed model by changing the value of network hyper-parameters such as the number of output channels, receptive field size, and dilation of its layers. The search for these hyper-parameters is performed in a mask-based fashion selectively removing parts of the seed network with fine-grain. PITSuperNet aims to combine a coarse-grained search performed with a supernet-based DNAS method inspired by a state-of-the-art method called DARTS, and the fine-grained search performed by PIT. Even though the supernet approach was already well-established, in this work it has been implemented from scratch to perfectly fit the PLiNIO's library and philosophy. Once the SuperNet method was properly tested on relevant use cases, the PIT algorithm has been integrated to explore the aforementioned coarse- and fine-grained search strategy. The combination of the coarse- and fine-grained search has been tested following two different approaches: the first approach consists of performing both PIT and SuperNet search at the same time in a mixed fashion, and the second approach consists of performing several PIT searches on the architecture found by a SuperNet search thus performing the fine-grained search after the coarse-grained one (PIT after SuperNet). Two loss terms are employed in the training phase: the loss of the task and a regularization loss that represents the size of the model. The strength of this term can be controlled by the user by means of a regularization strength constant λ.

Acknowledgements

In occasione del raggiungimento di questo importante traguardo della mia vita vorrei sentitamente ringraziare le persone che mi hanno aiutato a raggiungerlo.

Innanzitutto ci tengo a ringraziare i miei relatori, Daniele Jahier Pagliari, Matteo Risso e Alessio Burrello per avermi accompagnato e aiutato sempre con grande disponibilità e notevole prontezza durante questo percorso di tesi.

Un ringraziamento speciale va ai miei genitori, Fulvio e Marina, che mi hanno sempre sostenuto nel mio percorso universitario, scolastico e in generale di vita, senza farmi mai mancare nulla.

Grazie alle mie nonne, Antonietta e Bruna, che hanno sempre creduto in me e tanto hanno atteso questo momento per festeggiare.

Grazie a mio fratello Enrico che non è mai venuto meno al suo impegno di rimproverarmi se e quando stavo studiando troppo.

Grazie ai miei parenti tutti, zii e zie, cugini, per i bei giorni passati insieme.

Grazie anche ai miei amici con cui ho condiviso indimenticabili momenti di svago durante questi anni di studi e che hanno piacevolmente alleggerito la pressione dei momenti più impegnativi.

Grazie di cuore.

Contents

List of Tables

List of Figures

Acronyms

AI Artificial Intelligence.

BN Batch Normalization.

CNN Convolutional Neural Network.

CV Computer Vision.

DAG Direct Acyclic Graph.

DL Deep Learning.

DNAS Differentiable Neural Architecture Search.

DNN Deep Neural Network.

EC Evolutionary Computing.

FLOP Floating Point Operation.

GPU Graphic Processing Unit.

IoT Internet of Things.

MAE Mean Absolute Error.

MCU MicroController Unit.

ML Machine Learning.

MLP MultiLayer Perceptron.

MSE Mean Squared Error.

NAS Neural Architecture Search.

NLP Natural Language Processing.

NN Neural Network.

RL Reinforcement Learning.

SGD Stochastic Gradient Descent.

TCN Temporal Convolutional Network.

Chapter 1

Introduction

Artificial intelligence (AI) has become one of the most transformative technologies of our time. It has the potential to revolutionize the way we live and work, bringing new levels of efficiency and convenience to countless industries. This technology has the ability to automate tasks, analyze vast amounts of data, and make predictions based on that data, making it a powerful tool for solving complex problems. From healthcare to finance, retail to transportation, AI is already starting to change the way things are done, and many experts believe that its impact will only continue to grow in the coming years.

One of the most relevant branches of AI is Machine Learning (ML). Machine learning represents a possible implementation of artificial intelligence that allows computer systems to learn from data, identify patterns and make predictions or decisions. It involves algorithms that can iteratively improve themselves through training with large amounts of data. A Machine Learning algorithm differs from a traditional one because it does not require explicit rules to be given by the programmer to distinguish all the cases to make a prediction.

There are two main paradigms of machine learning: Shallow Learning and Deep Learning (DL).

Shallow Learning refers to traditional machine learning techniques that often require manual extraction of useful information from raw data as a pre-processing step. This process is called Feature Engineering and since it requires lots of human knowledge and domain expertise to be performed, it represents a limiting factor for the shallow learning paradigm. Deep Learning instead, aims to tackle this problem by performing this step in an automatic way, with no or very little human knowledge, by increasing model

complexity. DL algorithms, while being certainly more demanding in terms of computation requirements, are enabling incredible results in many fields such as Computer Vision (CV), Natural Language Processing (NLP), and Time Series Analysis. Nevertheless, Deep Neural Networks' complexity can be enormous and requires adequate computing tools, especially for the training phase. The training phase of these models has been enabled by tools such as GPUs that can provide high degrees of parallel computing, dramatically decreasing the time needed to train a NN compared to the time achieved by traditional CPUs. However, when it comes to exploiting these Deep Learning methods on small and constrained devices some challenges arise. Artificial Intelligence applications are expanding day by day and the Internet of Things is certainly a tempting field. In fact, having AI on IoT devices could provide several benefits including, for instance, real-time decision-making, personalized user experience, improved data analysis, and predictive maintenance.

In the IoT field, two main approaches have been explored: cloud computing and edge computing.

Cloud computing tries to solve the problem of the limited amount of resources available on the IoT nodes by performing all the computation (or at least the most part) on the cloud, relying on the availability of large server clusters. This involves the need, for the nodes, of sharing all the data collected through an internet connection and most of the time, to receive feedback with the cloud computation results. This approach presents many drawbacks. The first is connected to the network connectivity which can have unpredictable latency thus making this method not suitable for real-time applications. Furthermore when the amount of data to be transmitted is huge, e.g., for CV applications, bandwidth and energy consumption for the transmissions are not negligible. Another problem is privacy since cloud computing requires data to leave the device.

Instead, edge computing tries to perform the required computations (the inference phase for a DL application) directly on the nodes. Nonetheless, IoT devices usually present some important limitations in terms of computing power, available memory, and sustainable energy consumption. For such small devices, usually equipped with general-purpose MCUs and battery-powered, system lifetime is crucial and the deployment of standard NN models as they are i.e., designed to work on powerful GPUs with large memory footprints and energy-hungry operations, is simply not feasible. To solve this issue it is possible to create special-purpose chips that are engineered to solve specific tasks for specific devices but, as easily understandable, this approach

is costly and not scalable. For this reason, a more sustainable approach consists of optimizing the models to be implemented on these devices, in order to respect the limitations imposed by the existing chips. Edge computing solves all the negative sides of cloud computing since latency is usually lower but above all, it is predictable since it does not depend on connectivity, privacy is improved since data do not have to leave the device, and less energy is consumed for the transmissions. Of course, this approach requires more effort in order to adapt the software.

Numerous techniques are available to accomplish the task of reducing the complexity of NN models. Among others, we can mention precision reduction techniques such as quantization [1] which consists of passing from floating-point to fixed-point data representation. Since floating point usually requires 32 bits by switching to an 8-bit data representation it is possible to achieve a 4x memory saving. Moreover, integer arithmetic is usually faster and more efficient than floating-point arithmetic. It is also worth mentioning cardinality reduction techniques such as pruning [2] which tries to simplify architectures by removing redundant parts that do not bring significant improvement to the model performance.

This thesis work will focus on another incredibly effective tool that enables us to find optimized neural network architectures: Neural Architecture Search (NAS). The goal of NAS is to automate the process of designing deep neural networks by evaluating the performance of different candidate architectures and selecting the one that performs the best. Several recent works [3], [4], [5], [6] have focused their attention on developing NAS algorithms that aim to optimize neural network models for specific constraints, such as computational resources, memory usage, and power consumption, making it ideal for IoT devices. By using NAS, researchers can develop NN models that are optimized for the specific requirements of IoT devices and achieve improved accuracy and efficiency.

This context is precisely where PLiNIO [7] (**P**lug-and-play **Li**ghtweight **N**eural **I**nference **O**ptimization) fits in. PLiNIO (see Section 4.2) is a library that aims to integrate different NAS algorithms in order to be able to easily compare their effectiveness and results over some training datasets.

The first and seminal algorithm included in PLiNIO is PIT (Pruning In Time) [6], a NAS algorithm that tries to reduce the complexity of a seed model by changing the value of some network hyper-parameters such as the number of output channels, receptive field size, and dilation of its layers. PIT algorithm belongs to the DMaskingNAS category, which will be widely described in Chapter 3. PIT starts from a user-defined network, the so-called

seed, and explores sub-architectures included within the seed itself. While being extremely light-weight and able to explore a wide range of alternative hyper-parameters values it presents the limitation of being confined to the seed network topology. In fact, PIT is not able to completely switch the operation performed by a layer such as, for instance, swapping a convolutional layer with an identity layer. For this reason, PIT is considered a fine-grained NAS tool.

This work aims at extending the functionalities of the PLiNIO library by implementing a new NAS tool called PITSuperNet. The main objective of this tool is to combine a coarse-grained search, performed using another NAS technique usually referred to as *supernet* that again will be better described in Chapter 3, with the fine-grained search performed by PIT. Even though the supernet approach was already well-established (Section 3.2), in this work it has been implemented from scratch to perfectly fit the PLiNIO's library and philosophy (Section 4.2). After that, the SuperNet tool has been combined with PIT to create PITSuperNet.

A PITSuperNet module contains some user-defined alternative modules for that particular layer of the network. The NAS will establish the best alternative available within the given ones for each involved layer of the model. By doing so the NAS can work within an extended search space that is not necessarily limited to the space of the seed models. In this way, the NAS is able to insert simplified operations in some layers if complexity is not required but unlike PIT alone, it is also able to increase the complexity (w.r.t. the seed model) of certain layers when needed. While SuperNet extends the search space, PIT can still work on the alternative layers proposed in the PITSuperNet module ensuring a fine-grained search.

This method has been tested on some of the tasks available in the MLPerf Tiny Benchmark suite [8] using another library that implemented those tests in the PyTorch framework.

To combine the fine- and coarse-grained search types two approaches have been tried in this work. The first approach consists of performing both PIT and SuperNet searches at the same time with the PITSuperNet algorithm. While this first approach did not provide the hoped results for the reasons described in Chapter 5, with the second approach, which consists of performing a SuperNet search and then performing a PIT search over the SuperNet output model, it was possible to obtain some interesting results able to improve the ones obtained with a single fine- or coarse-grained type of search.

The rest of this work is organized as follows. Chapter 2 describes the necessary background on DL, Chapter 3 describes the related NAS works,

Chapter 4 describes the implementation of PITSuperNet and PLiNIO library organization, and Chapter 5 reports the results obtained with PITSuperNet on ICL, KWS, and VWW three of the available benchmarks in the MLPerf Tiny suite.

Chapter 2

Background

Machine Learning (ML) and Deep Learning (DL) are related but distinct branches of artificial intelligence. ML is a general term that encompasses a range of algorithms and techniques that enable computers to learn from data and make predictions or decisions. On the other hand, DL is a subfield of ML that is specifically concerned with building the so-called Deep Neural Networks (DNNs). These networks are made up of multiple layers of interconnected nodes. Each node includes specific parameters which are tuned in the so-called learning process. In this way, the model is able to directly learn from data becoming capable to solve tasks of increasing complexity.

Deep learning has been responsible for many breakthroughs in fields such as computer vision, natural language processing, and speech recognition, and has enabled systems to perform tasks that were once thought to be beyond the reach of computers. By leveraging vast amounts of data and powerful computing resources, DL models can continue to improve and advance the state of the art in many areas of artificial intelligence.

There are three main approaches used in ML and DL:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

Supervised learning is an approach to machine learning where the algorithms are trained on labeled data, meaning that the desired output is already known for each input. The goal is to learn a mapping from input to output so that given new, unseen data, the algorithm can predict the correct output. Supervised learning is often used for classification and regression tasks,

where the objective is to categorize items into predefined classes or predict a continuous target value, respectively.

Unsupervised learning is a type of machine learning where the algorithms are trained on unlabeled data. The aim is to uncover hidden patterns or relationships in the data, rather than making specific predictions. Unsupervised learning methods can be used for tasks such as clustering, where the goal is to group similar data points together, or dimensionality reduction, where the goal is to represent high-dimensional data in a lower-dimensional space.

Reinforcement learning is an approach to machine learning that involves training an agent to make a sequence of decisions in an environment by receiving feedback in the form of rewards or penalties. The agent's goal is to maximize its cumulative reward over time. Reinforcement learning has been applied to a variety of problems, including game playing, robotics, and control systems. The training process in reinforcement learning is often referred to as trial-and-error, as the agent learns from its own experiences over time to make better decisions.

This work will focus on supervised learning methods and classification tasks. This section presents a brief overview of the main concepts behind Deep Learning.

2.1 Neural Networks

2.1.1 Neuron

The basic unit of a Neural Network is called *neuron*. The artificial neuron, whose architecture is inspired by biological neurons in the human brain, performs a weighted sum of the inputs and passes this sum through a non-linear function called *activation function*. In mathematical formulation:

$$y = h(\sum_{i=1}^{n} w_i x_i + b) \tag{2.1}$$

Where w_i and b are weights and bias respectively, xi are the inputs, and $h()$ is the activation function. After a training phase, where weights and biases are tuned, this basic unit can act as a binary classifier on very simple tasks.

2.1.2 Activation Functions

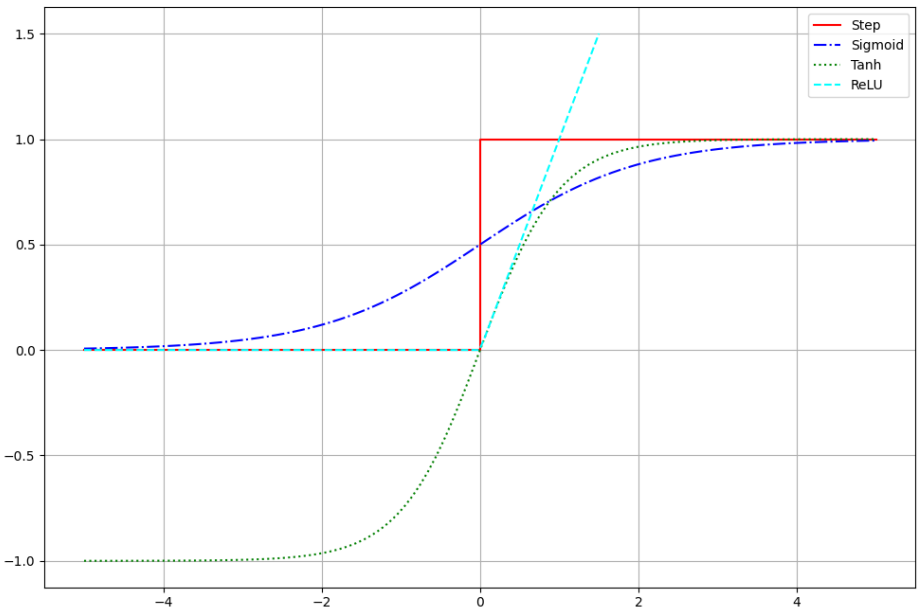


Figure 2.1. Activation functions.

The activation function is a fundamental element for the neuron architecture since it defines the range of its output and it introduces a non-linearity allowing the neural network to model complex relationships between inputs and outputs. Here the most relevant activation functions are listed:

- **Step Function** (red curve Figure 2.1 the simplest activation function is first introduced with the *Perceptron* [9]:

$$h(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases} \tag{2.2}$$

  Its outputs are discrete (only 0 or 1) and this fact, together with the problem of its non-derivability in x=0, strongly limits its usage in modern networks that use the backpropagation technique (Section 2.3.2).

- **Sigmoid Function** (blue curve Figure 2.1): it solves the problem of discrete outputs with a continuous output interval ranging between 0 and 1 and it is fully derivable in its domain:

$$h(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

  Although being fully derivable, sigmoid clearly shows a saturating behavior (see Figure 2.1) that can "kill" the gradients. This is the so-called problem of vanishing gradients in the backpropagation algorithm i.e., when the input is too small or too big the output goes to the constant values 0 or 1, hence saturating and making its gradient vanish. Moreover, sigmoid outputs are not zero-centered and this can create problems with the weight gradient update. In fact, gradients become all positive or all negative, forcing the gradient update direction to follow a zig-zag path whenever the optimal direction is not within the two eligible ones.

- **Hyperbolic Tangent Function (tanh)** (green curve Figure 2.1): similar to the sigmoid function but its outputs are zero-centered (interval [-1, 1]):

$$h(x) = tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.4}$$

  It usually converges faster than the sigmoid function but still presents the problem of vanishing gradients.

- **Rectified Linear Unit (ReLU)** (light-blue curve Figure 2.1): the most used activation function in recent networks together with its variants (Leaky ReLU and ELU):

$$h(x) = max(0, x) \tag{2.5}$$

ReLU is computationally efficient, requiring only a simple threshold operation to compute its output and it can converge much faster than sigmoid or tanh [11]. It solves the problem of vanishing gradients for large values of the input but it is not zero-centered, meaning that it can cause some neurons to always produce the same output, leading to a phenomenon known as the "dead ReLU" problem. Leaky ReLU solves this problem by putting a small slope on the negative side of the ReLU function to avoid zero gradients (Figure 2.2).

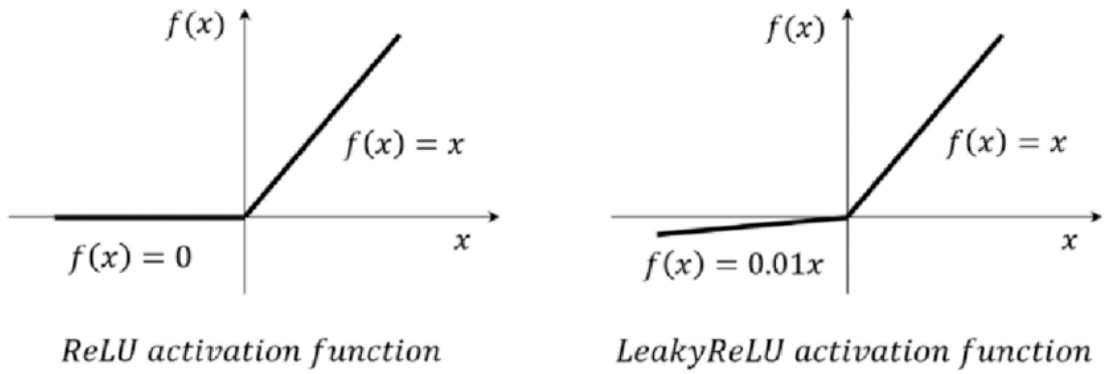


Figure 2.2. ReLU vs LeakyReLU [12].

2.1.3 Layers of Neurons

A single neuron is very limited in terms of tasks that can solve, and the main idea behind the neuron was to connect more of them to mimic the human brain functions performing inference. A layer of neurons can be composed by bringing together several neurons in parallel and an entire network can be created by connecting multiple layers one after the other. This architecture is known as *Multi-Layer Perceptron* (MLP) [13].

Each layer of an MLP is composed of different nodes that are single neurons. The first layer of an MLP is called *input layer*, this layer is fed with

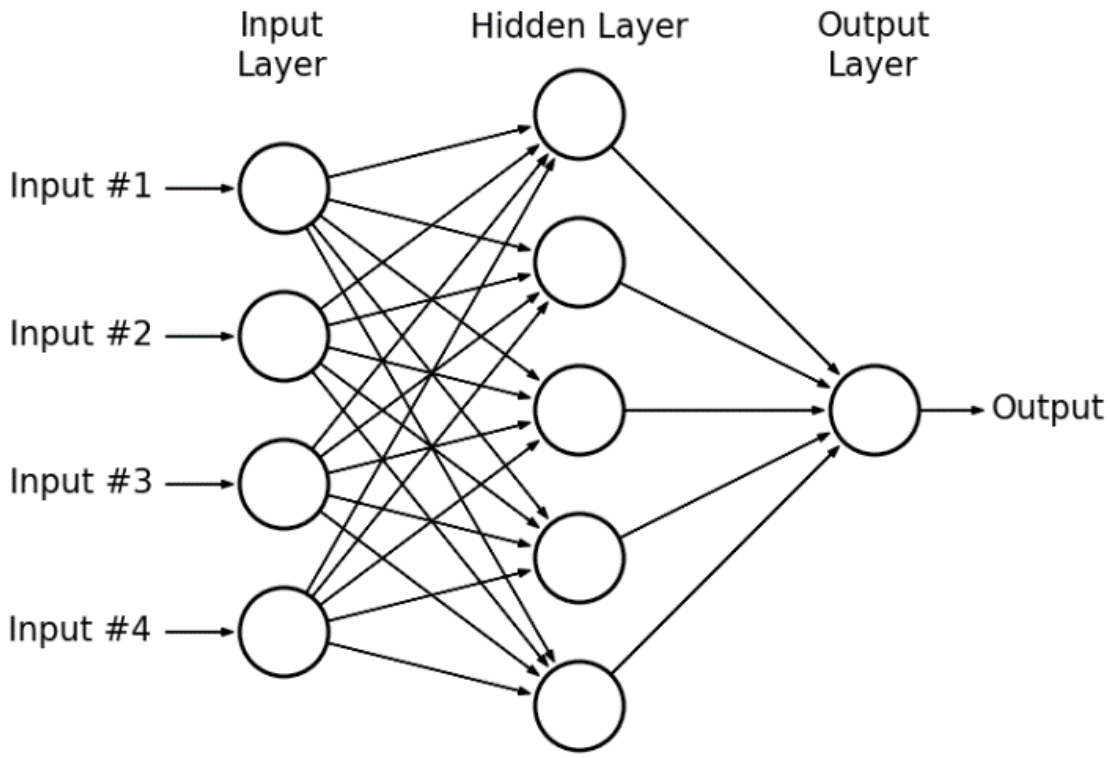


Figure 2.3. MLP sample architecture [10].

the input data in their original representation, and its outputs are propagated to the following layers. The last layer is known as *output layer* since it transforms its inputs into the final desired representation required to solve the task. Each intermediate layer is called *hidden layer* and it transforms the inputs into an intermediate representation that is useful to learn data relations. This part represents the "deep" part of a neural network and enables the model to extract useful information from the data without the need for a manually performed Feature Engineering phase.

The possibility to automatically perform this phase requiring no or very little human knowledge and domain expertise represents one of the main innovations of Deep Learning with respect to Shallow Learning. Not only there is no need to manually extract relevant features from data requiring great knowledge of the task domain but in this way, the models are free to detect and learn to identify new patterns that, in some cases, are not even considered by humans.

In this architecture, each layer is a fully connected one meaning that every

node in the layer is directly connected with all the nodes of the following layer. The number of nodes in the input and output layer depends on the data and the task to be performed, while the dimensions of hidden layers have fewer constraints and are free to be designed.

2.2 Neural Network's Layer Types

One of the most relevant Deep Neural Network architectures available today is Convolutional Neural Network (CNN). CNNs have become the standard approach for Computer Vision (CV) tasks like image classification and object detection. AlexNet [14], VGG [15], GoogLeNet [16] and ResNet [17] are some of the milestone models for the CNN architecture. A CNN consists of multiple layers including convolutional layers, pooling layers, normalization layers, and fully connected layers.

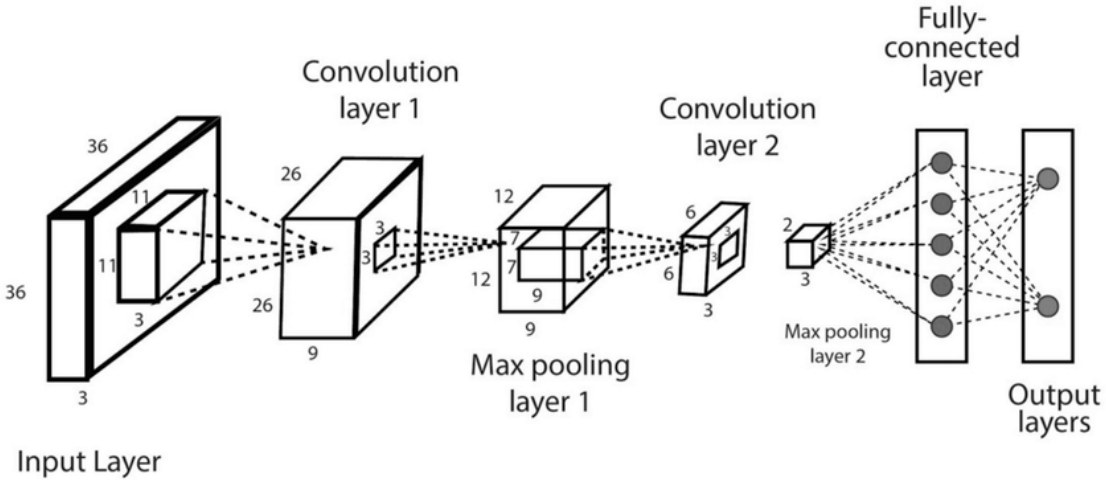


Figure 2.4. CNN example [18].

Convolutional Layer

The main layer of a CNN is the convolutional layer. A convolutional layer operates by "sliding" a small matrix called *filter* over the input data, such as an image, and computing dot products between the weights of the filter and the input at each location. This results in a feature map that represents the presence of certain features in the input data. Multiple filters can be used in parallel to extract different features. By using multiple convolutional layers,

a deep neural network can learn increasingly complex representations of the input data. The filter, whose size is named *kernel*, slides over the image with a step called *stride*. By using a filter that slides on the whole image, the weights defining this filter are shared for all the pixels. This characteristic avoids the use of a different weight for each pixel which would make the task feasible only for really small images.

Pooling Layer

Pooling layers are used to reduce the spatial dimensions of the feature maps produced by the convolutional layers. The goal of pooling is to down-sample the feature maps, which helps to reduce the computational load and prevent overfitting. There are two commonly used types of pooling: *Max Pooling* and *Average Pooling*.

In Max Pooling, the maximum value of a set of adjacent feature map values is taken, whereas, in Average Pooling, the average value of a set of adjacent feature map values is taken. Both types of pooling perform the down-sampling by dividing the feature map into non-overlapping regions and computing a summary statistic for each region. Pooling helps to make the feature maps invariant to small translations and distortions in the input data, as well as reducing their size and computational complexity, allowing the CNN to focus on more abstract and important features in the data.

Normalization Layer

Normalization layers are used to improve the stability of training and avoid overfitting. They usually normalize the output of preceding convolutional layers, typically by subtracting the mean and dividing by the standard deviation. Batch Normalization (BN) [19] is the most widely used type of normalization in CNNs. It normalizes the activations of neurons within a mini-batch of data, helping to prevent the internal covariate shift. In mathematical formulation:

$$y^{(k)} = \gamma^{(k)} \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{Var[x^{(k)}] + \epsilon}} + \beta^{(k)} \tag{2.6}$$

This transformation is applied to each point of each channel k of the input feature map x. $\mathbb{E}[x^{(k)}]$ represents the mean within a batch, $Var[x^{(k)}]$ is the variance within a batch, and γ and β are two trainable parameters used to scale and shift the result. ϵ is just a small number to avoid divisions by zero. By normalizing the activations, these layers can improve the convergence

speed of the training process and make the model more robust to parameter changes. Batch Normalization allows using higher learning rates in the training phase and creates a regularizing effect helping models to better generalize.

Fully-Connected Layer

Fully connected layers, also known as dense layers, are a type of layer commonly used in deep neural networks (DNNs). They are called "fully connected" because each neuron in the layer is connected to every neuron in the previous layer. In a fully connected layer, the neurons receive input from all neurons in the previous layer, process this input using an activation function, and produce an output that is fed as input to the next layer. Fully connected layers are used to learn complex representations of the input data in a DNN, and they are typically found at the end of the network, after a series of convolutional or recurrent layers. The output activations of a fully-connected layer can be interpreted as a probability distribution over the possible classes for the input. In many CNN architectures, the final layer of the network is a fully-connected layer that outputs the class predictions.

2.3 Training a DNN

The objective of a Deep Neural Network (DNN) training phase is to learn the optimal weights of the network that map inputs to outputs such that the error between the network's predictions and the actual target values is minimized. In the supervised learning approach, the training process involves presenting the network with a large dataset of input-output pairs, allowing the network to make predictions, computing the error between its predictions and the actual targets, and updating its weights based on the error. This process is repeated multiple times, or over multiple epochs until the error reaches a satisfactory level, or no further improvement is observed. The end result is a trained network with weights that can generalize well to new unseen data, making accurate predictions for the task it was trained on. The training phase of a Deep Neural Network (DNN) consists of the following steps:

1. Feeding the network with a large amount of labeled training data.
2. Initializing the weights of the network randomly.
3. Feeding the input data through the network and computing the output (forward pass).

4. Calculating the error between the network's output and the actual target value.

5. Updating the weights of the network to reduce the error, using an optimization algorithm.

6. Repeat steps 3 to 5 for multiple epochs (iterations) until the error reaches a satisfactory level or no further improvement is observed.

2.3.1 Loss Functions

The error to be minimized during the training phase is represented by the loss function. The loss function can be seen as the distance between the input data distribution and the target data distribution. Different types of loss functions are available for different types of tasks. Here some of the most commonly used loss functions are reported:

- **Cross-entropy Loss**: most used loss for classification tasks. It measures the dissimilarity between the predicted probability distribution and the true distribution of the target classes:

$$H(p, q) = -\sum_{x \in X} p(x) \log(q(x)) \tag{2.7}$$

- **Mean Squared Error (MSE)**: most used loss for regression tasks. It minimizes the squared difference between the predicted value and the target value:

$$L(y, \bar{y}) = \frac{1}{N} \sum_{i=0}^{N} (y - \bar{y})^2 \tag{2.8}$$

- **Mean Average Error (MAE)**: also used in regression tasks, it minimizes the absolute difference between the predicted value and the target value:

$$L(y, \bar{y}) = \frac{1}{N} \sum_{i=0}^{N} |y - \bar{y}| \tag{2.9}$$

2.3.2 Gradient Descent Learning Method

The most used optimization algorithm in Deep Learning training is called Stochastic Gradient Descent (SGD). SGD is a special case of the more general Gradient-based learning method which is a method used to update the

weights of the network during the training phase. It involves computing the gradient of the error (loss function) with respect to the weights and updating the weights in the direction that minimizes the error. The gradient provides the direction of the steepest descent in the error space, and the step size (or learning rate) determines the magnitude of the update. Setting a good learning rate is crucial. A learning rate that is too low would require a huge number of epochs to reach a good result while a learning rate that is too high would make the algorithm diverge. The formula used in gradient descent to update the network's parameters is:

$$\theta^{new} = \theta^{old} - \eta \nabla_\theta \mathcal{L}(\theta) \tag{2.10}$$

Where θ represents the network's parameters, η is the learning rate and $\nabla_\theta \mathcal{L}(\theta)$ is the gradient of the loss with respect to θ.

It is called "stochastic" since it differs from the standard batch gradient descent because instead of calculating the gradients based on the average of all samples in the training set, SGD uses only one randomly selected sample from the training set to calculate the gradients at each iteration allowing the process to be much faster. However, this random selection introduces more noise in the gradients and makes the optimization more unstable. The loss decreases on average with more iterations but it can temporarily increase and decrease with respect to the preceding iteration. This instability is not entirely a bad thing since it can also help to get out of local minima and reach faster the global minimum. To mitigate this, SGD typically uses a smaller learning rate and implements techniques such as learning rate decay and mini-batch SGD. Some alternative optimizers to the SGD are represented by Adam [20], RMSProp [21], and Adagrad [22].

To enable the computation of all the gradients in an efficient way the backpropagation algorithm [23] has been proposed. In this algorithm two passes over the network are done: the forward pass where the input is fed into the network's layers and the loss is evaluated, and the backward pass in which the network is passed in the opposite direction through a chain-rule technique allowing to compute the gradients of the loss with respect to the weights. The chain rule in the backpropagation algorithm is a mathematical concept that allows the gradient of the loss with respect to the parameters of a neural network to be efficiently calculated. It is based on the chain rule of differentiation and exploits the fact that the loss can be expressed as a composite function of many intermediate computations within the network. The backpropagation algorithm is what makes it possible to train deep neural networks effectively and efficiently, as it allows the error gradients to be

efficiently calculated and the weights updated, even for complex network architectures with many layers.

2.3.3 Regularization Techniques

Regularization loss is a term used in deep learning to refer to additional terms added to the original loss function during training. The purpose of these additional terms is to discourage the model from overfitting the training data and to improve its generalization to new, unseen data. Overfitting occurs when a model becomes too complex and starts to memorize the training data instead of learning its underlying patterns. This results in high training accuracy, but poor accuracy on test data, as the model has not learned to generalize well. Regularization helps to mitigate overfitting by adding a penalty to the loss that discourages the model from having large weights. There are several types of regularization methods:

- **L2 Regularization** (also known as Ridge): adds a penalty proportional to the square of the weights:

$$\mathcal{L}_2 = \lambda ||\theta||_2^2 = \lambda \sum \theta_i^2 \tag{2.11}$$

- **L1 Regularization** (also known as LASSO): adds a penalty proportional to the absolute value of the weights:

$$\mathcal{L}_1 = \lambda ||\theta||_1 = \lambda \sum |\theta_i| \tag{2.12}$$

- **Dropout**: randomly drops out neurons during training, effectively making the model smaller and more robust to overfitting. It works by randomly dropping out (i.e., setting to zero) a certain percentage of neurons during training. This forces the network to learn multiple independent representations of the data, rather than relying too heavily on any single neuron. During test time, all neurons are used, but their activations are scaled by the dropout rate to balance the reduced number of active neurons during training.

- Regularization loss can also be used to encode a prior in the gradient-based learning process as seen in the NAS algorithms. (Chapter 3).

Chapter 3

Related Works

Neural Architecture Search (NAS) is a field of machine learning that focuses on automating the process of designing deep neural networks. In traditional deep learning, the architecture of a network, such as the number of layers, the type of activation functions, and the number of neurons in each layer, is typically hand-designed by a human expert. However, as the complexity and diversity of problems and deployment targets in artificial intelligence continue to increase, hand-designing network architectures can become a time-consuming and challenging task. This is where NAS comes in: it uses a combination of machine learning algorithms and search algorithms to automatically discover the best architecture for a given problem. By automating the process of architecture design, NAS can save time, reduce human error, and enable the creation of more sophisticated and effective models. NAS tools can also jointly optimize the model's accuracy and other metrics like the size (number of parameters), the number of FLOPs, etc. . This result in a model that delivers the best performance in terms of accuracy while respecting some limitations imposed by the designers that can be needed to deploy this model in edge devices.

3.1 Reinforcement Learning NAS

The first attempts to realize a NAS tool were based on Reinforcement Learning (RL) [24]. In this approach, one network named *controller* outputs network architectures. These architectures are trained and evaluated on a validation set. The accuracy reached by this architecture is used as a reward for the controller NN that will use this signal to update its policy gradient in order to produce a better architecture over time. The controller is trained

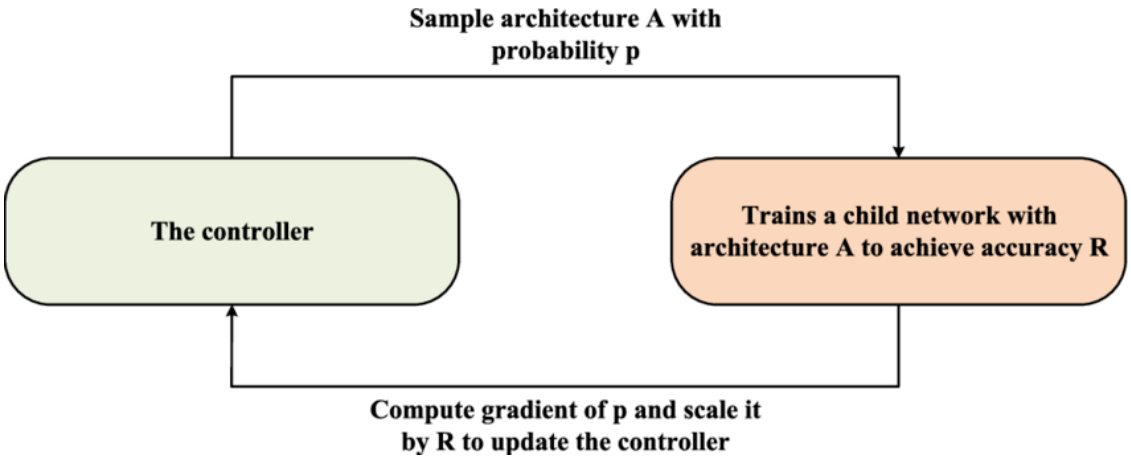


Figure 3.1. RL-based NAS concept [25].

to make decisions about the architecture of its child NN, such as adding or removing layers, or for instance for a convolutional layer changing the kernel size, the number of filters, or the stride. Even though RL-based NAS has the potential to explore a big number of network architectures it requires a large amount of computation to train the controller and evaluate the performance of different architectures. Moreover, training times are usually very long (they can require days if not weeks even with powerful GPUs on simple tasks), as the search process requires the evaluation of many different architectures. This can make it difficult to perform large-scale searches, especially when working with limited computational resources. This method showed the great possibilities offered by the NAS technique but at the same time highlighted the need for a lighter technique to enable the architectural search even with fewer computational resources available.

Evolutionary Computing (EC) based NAS represents an improved approach with respect to RL-based NAS. ENAS [26] tries to avoid training from scratch each architecture by sharing weights coming from previous searching phases allowing to shrink the time needed to perform NAS training. Even if improved, EC-based NAS still requires long training times since it has to train a big number of different model architectures and thus basically suffers from the same problem as RL-based NAS.

3.2 Supernet-based Differentiable NAS

Differentiable Neural Architecture Search (DNAS) is a variant of Neural Architecture Search that has been introduced to overcome the problems that affect Reinforcement Learning NAS. The first method that was proposed with this DNAS approach was DARTS [27]. In DNAS, the architecture search process is made differentiable, allowing it to be optimized using gradient-based optimization techniques like backpropagation and enabling the optimization of the network's weights jointly with the architecture optimization. The architecture of the network can be expressed as a continuous and differentiable function, allowing the optimization algorithm to smoothly and efficiently make changes to the architecture. The starting architecture is called a supernet since it provides more alternative operations for each layer that will eventually be shrunk down to the most performing operation available.

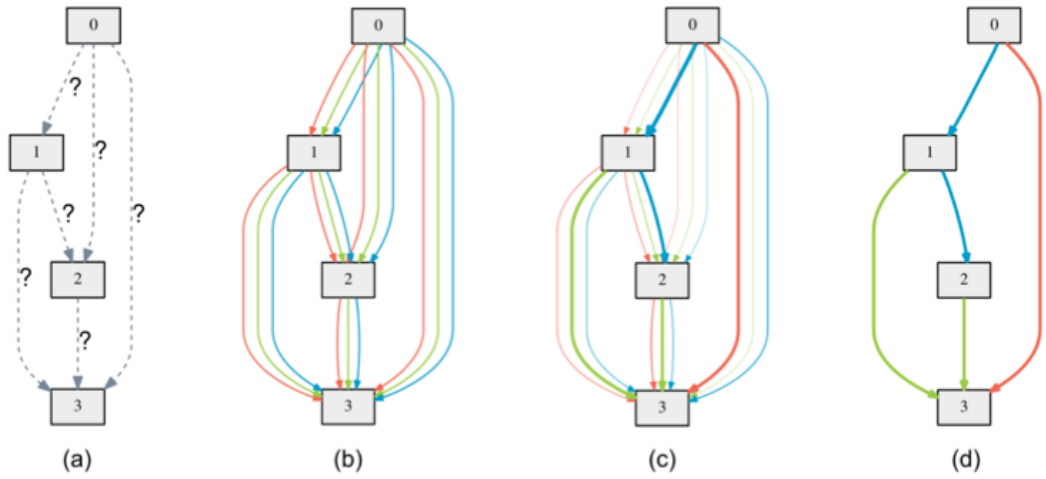


Figure 3.2. DARTS search steps [27].

DARTS works by representing the neural network architecture as a directed acyclic graph (DAG), where each node $x^{(i)}$ is a data representation (feature map) and each edge (i, j) is associated with some operation $o^{(ij)}$ (e.g. convolution, pooling, etc.) that transforms $x^{(i)}$.

Given a set of possible operations $\mathcal{O}$ where each operation $o(\cdot)$ is a function to be applied to $x^{(i)}$, in order to make the search space continuous, a softmax with this form is applied over all the candidate operations:

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp\left(\alpha_o^{(i,j)}\right)}{\sum_{o' \in \mathcal{O}} \exp\left(\alpha_{o'}^{(i,j)}\right)} o(x) \tag{3.1}$$

The result $\bar{o}^{(i,j)}$ is a mixed operation.

The architecture search is performed by optimizing the connections between these nodes. During the training process, the parameters of the operations are learned and updated through gradient-based optimization. At the same time, the connections between the nodes are also treated as variables and their associated weights α_o are optimized using gradient descent. This allows the architecture to change dynamically and continuously during training, based on the feedback from the optimization process. The final architecture is selected by pruning the connections with the lowest weights discarding the operations that contribute the least to the performance of the network. The process is repeated until the desired number of connections is reached. This makes it possible to search for the optimal architecture in a matter of hours, compared to days or weeks for traditional NAS methods based on RL. However, DARTS's main drawback depends on the high memory consumption that grows linearly with the supernet size and that impedes the exploration of bigger search spaces. ProxylessNAS [3] is another DNAS method that tries to overcome this problem by training only a single path of the graph for each batch of training data, thus limiting memory occupation but giving up exploring the whole search space. DARTS uses only the task loss as a target while ProxylessNAS is the first method to employ a multi-objective loss in order to search for the best trade-off between model accuracy and some hardware-specific constraints such as latency.

3.3 Masked-based Differentiable NAS

Another promising DNAS method that has been introduced in the last years is mask-based DNAS or DMaskingNAS. This method uses binary masks, which are made differentiable and trainable, to explore some network's architecture hyperparameters that were not called into question in the preceding methods such as spatial and channel dimensions. The masks are updated at each iteration of the gradient-based training algorithm to allow for the discovery of new and improved architectures. This method allows for a fine-grained search over a seed network performed at the same time as the training of the network weights, as the DNAS technique suggests, thus making the search

really lightweight (with no significant overhead with respect to standard network training). This approach solves the scalability issue that affected most of the previous NAS methods making it possible to apply NAS also on large models and datasets. This of course comes at the cost of a search space limited to sub-architectures contained in the single seed architecture. The two seminal mask-based DNAS works are MorphNet [4] and FBNetV2 [5].

MorphNet work provides a tool to search for neural architectures that can target some specific constraints in terms of model size or, for instance, the number of FLOPs per inference to assist the deployment of these models on resource-limited devices such as IoT edge devices. MorphNet focuses on the search for the optimal value for output channels of convolutional layers while network topology, filter dimensions, and other possible design choices are treated as fixed. The optimization problem assumes the form:

$$O^*_{1:M} = \underset{\mathcal{F}(O_{1:M} \leq \zeta)}{\arg\min} \min_{\theta} \mathcal{L}(\theta) \tag{3.2}$$

Where θ are the network's parameters, $\mathcal{L}$ is the loss function, $O_{1:M}$ are the output channels of every layer and $\mathcal{F}(O_{1:M} \leq \zeta)$ represents the costraint to be respected (e.g., number of FLOPs). To solve this optimization problem MorphNet combines two alternative solutions by means of a 3-step algorithm. The first is based on a sparsifying regularizer that put a greater cost on the neurons that contribute more to the $\mathcal{F}$. The choice of the regularizer depends on the constraint required. This is applied in steps 1 and 2 which are usually shrinking steps. These two steps usually result in a network that requires fewer resources but sacrifice performance. The second is based on a simple width multiplier that can shrink or expand the output channels of each layer. This is used in step 3 to perform an expansion of the network and regain performance. This procedure is repeated iteratively and leads to an overall improved architecture, with respect to the seed one, able to respect the constraints imposed while not sacrificing performance.

FBNetV2 work proposes an efficient way to use the masking mechanism for feature map reuse in order to search over spatial and channel dimensions. As depicted in Figure 3.3 instead of summing the output of 3 different padded convolutions (step B) or using three different padded masks (step C) that would still make the computational cost grow with the number of options available, FBNetV2 approximates the different convolutions by using weight sharing and summing the padded masks before multiplying by the output (before summing the masks are multiplied by Gumbel-Softmax weights [28]). In this way, they can provide a constant computational cost

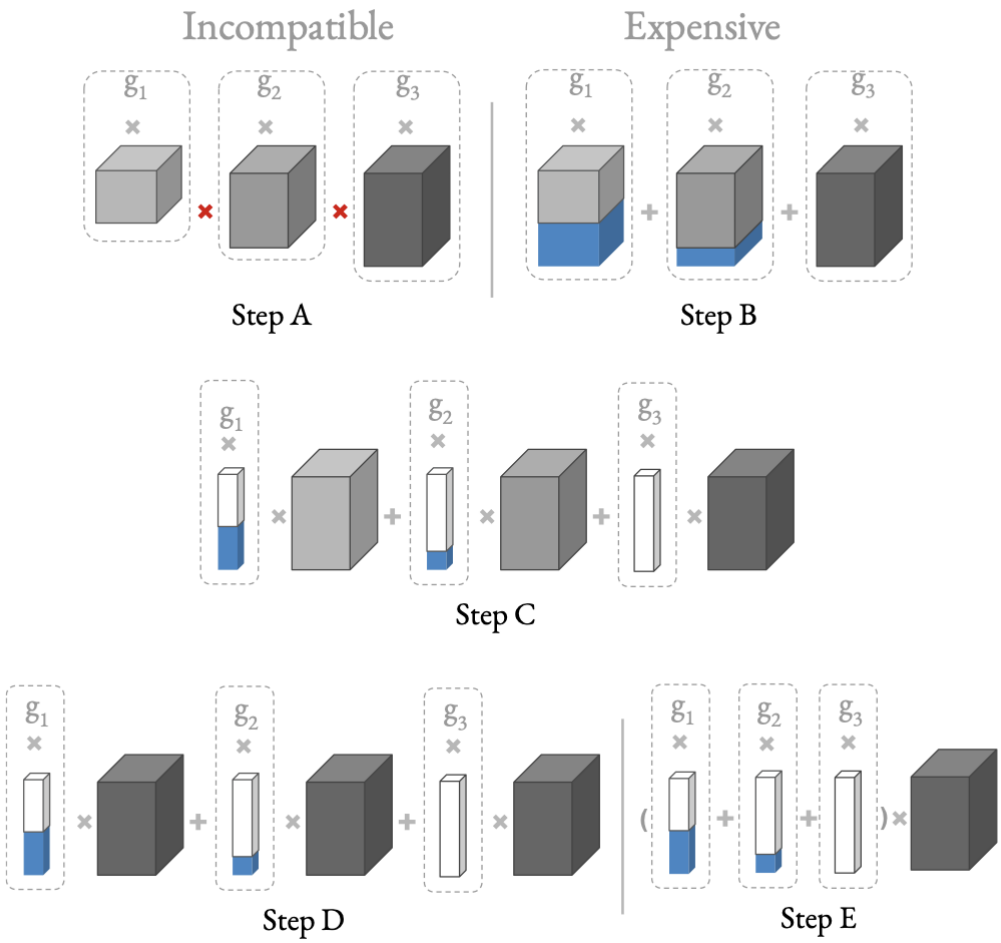


Figure 3.3. FBNetV2 Channel Search [5].

even with increasingly larger search options.

Another relevant mask-based DNAS method for this thesis work is PIT (Pruning In Time) [6]. PIT, as MorphNet, starts from a seed network and tries to reduce model complexity while preserving accuracy by employing trainable masking parameters to perform structured weight pruning. PIT tries to reduce the complexity of a seed model by changing the value of network hyper-parameters such as the number of output channels, receptive field size, and dilation of its layers. PIT's main focus and novelty is the optimization of Temporal Convolutional Networks (TCNs) that are commonly used for time-series processing tasks while the rest of the NAS techniques

focused only on 2D-CNN models. Moreover, while other works optimize the number of channels PIT is able to extend this optimization also to receptive field and dilation. To optimize the output channels PIT uses a vector of α parameters of the same length as the original number of output channels and applies a Heaviside binarization function to the parameters α returning 0 if the value is below the threshold and 1 if it is higher. The weights of each output channel are then multiplied for the masks and if the mask value is zero the corresponding channel is pruned.

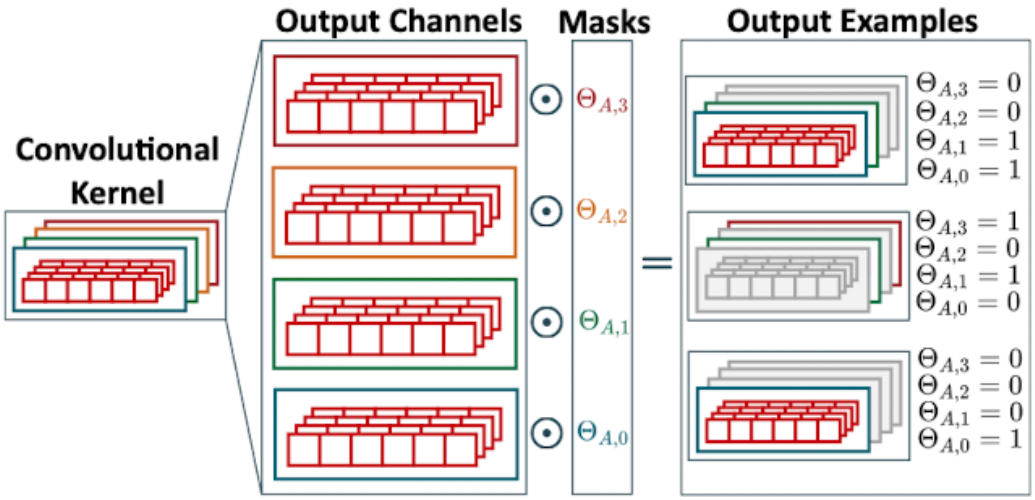


Figure 3.4. PIT Channel Search [6].

Chapter 4

Hybrid Coarse- and Fine-Grained DNAS

4.1 Objective

In recent years, the increased availability and impressive performance of AI applications have risen the interest in the possibility of deploying DNN models directly at the edge. On the other side, the Internet of Things (IoT) has revolutionized the way we interact with our surroundings, enabling us to connect and communicate with an expanding range of devices. The deployment of AI on such devices can enable performing inference in real-time and take decisions directly at the edge. However, deploying DNN models on edge devices introduce several challenges. Some of the major challenges include:

- Limited computing resources: Edge devices have limited processing power, memory, and storage, which can make it hard for them to run complex algorithms. This requires developing lightweight algorithms to ensure they can be executed efficiently on edge devices.
- Power consumption: DNN models deployed on IoT edge devices may consume a significant amount of power, which can limit the battery life of the devices. IoT devices are typically battery-powered, and frequent replacement of them may not be easy (e.g., sensors that are located in hard or dangerous places to reach such as machinery, bridges, etc.) as well as representing a significant cost term over time. This requires special consideration when developing models for these devices.
- Heterogeneous hardware and software: Edge devices can be built using

different hardware and software architectures, making it difficult to develop applications that can run on all edge devices. This requires the development of standardized interfaces and protocols to ensure interoperability across devices.

- Scalability: The number of edge devices in an IoT deployment can grow rapidly, making it challenging to manage and scale the infrastructure.

To tackle some of these challenges, as extensively described in Chapter 3, NAS methods are becoming increasingly popular, making it possible to find architectures optimized to work on edge devices while respecting the constraints imposed by them and without or with reasonable loss of accuracy. Aside from the Reinforcement Learning-based methods that, as seen in Section 3.1, require too many computational resources and very long training times, the supernet-based and mask-based DNAS methods have yielded the best results while being much more efficient from the training time perspective.

The mask-based DNAS methods especially are the ones that were found to be the lightest and consequently allow more options to be explored in the search given that they are also scalable to larger networks and datasets. However, these methods while being able to explore a large number of alternatives for sub-architectures contained in the seed network, remain unable to explore completely different layer operations alternatives. On the other hand, supernet-based methods allow exploring different network topologies providing the NAS the ability to choose within some alternative operations to be performed at each involved layer.

The first goal of this work is to create a method that is able to combine the coarse-grained search performed by supernet-based methods and the fine-grained one performed by PIT while preserving mask-based methods' lightness as much as possible. This is achieved by inserting into the original model some user-defined supernet modules containing some layer alternative operations, and by applying the PIT algorithm on all the available layers. By doing this, the search space is still wide and full of possible fine-grained choices like the precise number of output channels of each NAS-able layer but at the same time, the NAS is allowed by the supernet implementation to explore some alternative operations for some of the network layers that are outside the seed network topology. It is important to underline that the layers contained in the supernet modules are constantly evaluated during the training in the optimized form chosen by the PIT algorithm allowing a complete mixture of the two methods.

In another approach explored in this thesis, however, PIT-based fine-grained search is performed later and separately from supernet-based coarse-grained search. In particular, in this approach, the PIT search is performed on the model exported by the supernet-only search. A comparison of the two approaches (the mixed coarse- and fine-grained search and the fine-grained search performed after the coarse-grained search) is provided in Chapter 5.

4.2 PLiNIO Library

This section describes the characteristics and organization of the PLiNIO library [7].

PLiNIO stands for **P**lug-and-play **Li**ghtweight **N**eural **I**nference **O**ptimization and aims to aggregate in a single user-friendly library multiple Differentiable NAS methods in order to allow the users to easily compare their effectiveness on their models. It is a Python package built on top of the PyTorch ecosystem and, as highlighted by its name, it targets the optimization of Neural Network models with an eye on the inference phase in order to facilitate the deployment of these models on edge devices. It exploits DNAS algorithms that, as described in Chapter 3, have proven their lightness and effectiveness in finding optimized architectures to be deployed on edge devices.

Ease of use has been one of the main concerns during the development of this tool leading to the creation of a user-friendly library (Plug-and-play). To optimize the architecture of a model using one of the available DNAS methods the user has to add only three simple lines of code to the original training loop (Figure 4.1):

- A line to convert the original model into a NAS-able model. Calling the desired DNAS method constructor, with the original model as an argument, is what it takes.
- A regularization term needs to be added to the loss in order to balance accuracy and the other chosen metric (e.g., model size) and effectively train the architectural parameters together with the model parameters. The regularization term is easily retrieved by calling the function `get_regularization_loss()` on the NAS model.
- A line to perform the conversion of the final NAS discovered model into a standard PyTorch model. The function `arch_export()` called on the NAS model returns the optimized model into a standard PyTorch model.

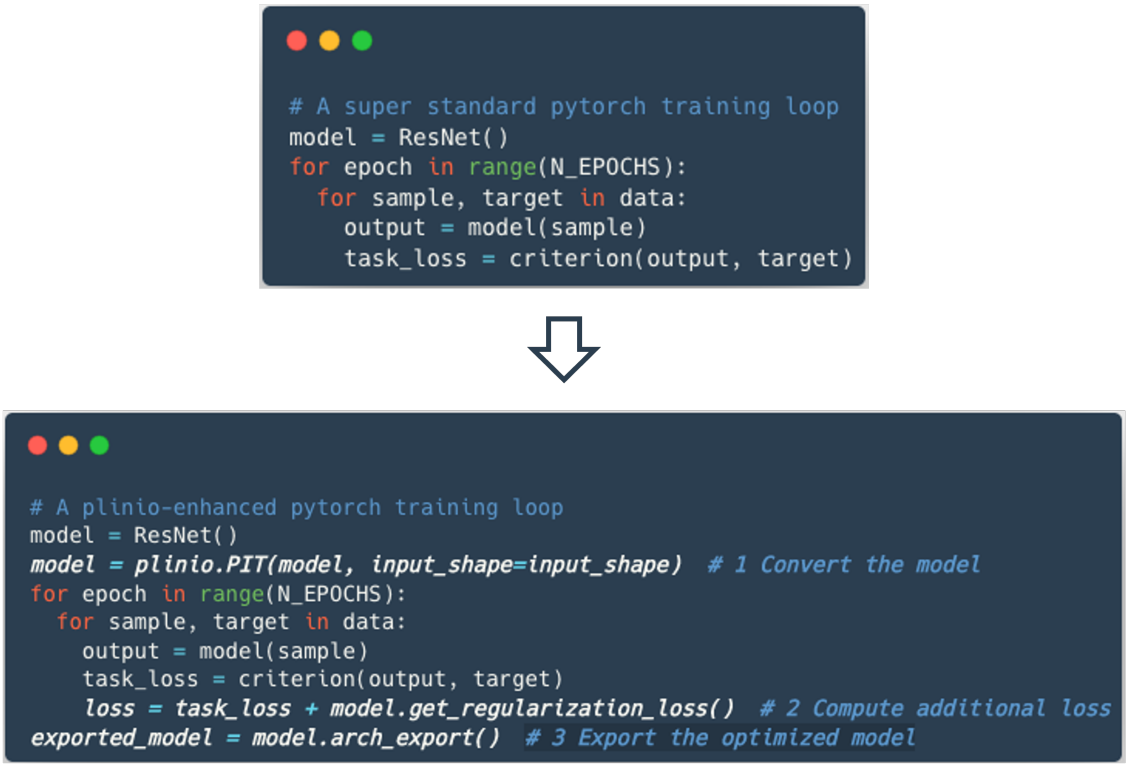


Figure 4.1. Standard training loop vs PLiNIO training loop [7].

Moreover, the library has been organized to ease the integration of new methods and algorithms.

As depicted in Figure 4.2 the code in the library is organized into two main directories `plinio` and `unit_test`.

The plinio directory contains the main library code while the second contains a set of unit tests. The graph folder contains a set of files that perform various general operations on the model graph that are useful for all the NAS methods. Some of these operations will be better described in Section 4.3. The methods folder instead contains all the available DNAS methods in the library. Currently, these methods are PIT, PITSuperNet, and MixPrec (under development):

- `dnas_base`: it contains an abstract class (DNAS) with common code for the DNAS algorithms. Its constructor takes as input a string describing the regularizer to be used in the architecture search ('size' or 'macs' are currently available), a list of names of layers that should be excluded from the search, and a list of types to be excluded.

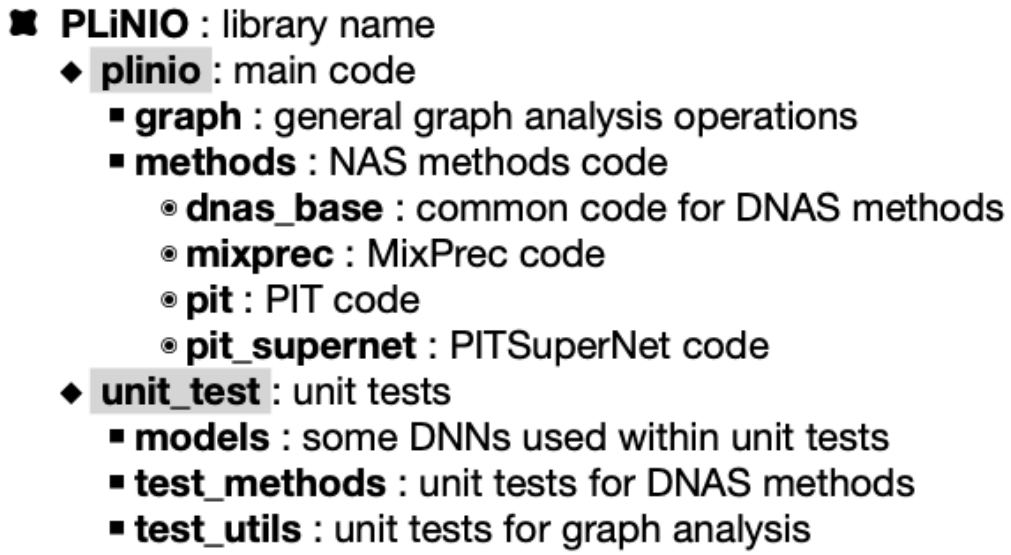


Figure 4.2. PLiNIO Library code organization.

- `mixprec`: this part is currently under development and will not be discussed in this thesis.
- `pit`: it contains the implementation of the PIT DNAS method [6]. The `PIT` class takes a PyTorch model as input and converts all the convolutional and fully-connected layers that can be found into a `PITModule` which is an abstract class that provides an interface to all the possible specific PIT modules. The `PITModule` class and all its specific inheriting classes (`PITLinear`, `PITConv1d`, `PITConv2d`, `PITBatchNorm1d`, `PITBatchNorm2d`) are contained in the `nn` folder together with `PITFeaturesMasker`, `PITTimestepMasker`, and `PITDilationMasker` that are used to train respectively the number of output channels, the receptive field, and the dilation. The `graph` file contains all the PIT-specific functions that operate on the model's graph. These operations are mainly related to the model's conversion.
- `pit_supernet`: this module has been introduced in this thesis work. It contains the implementation of the PITSuperNet DNAS method. It presents a similar structure to the `pit` folder. The `PITSuperNet` class takes as input a PyTorch model with some user-defined

`PITSuperNetModule`s. This model is converted by applying PIT conversion with some additional steps for the SuperNet part. The `nn` folder contains the `PITSuperNetModule` definition and the definition of the `PITSuperNetCombiner` which contains the logic behind the `PITSuperNetModule` that acts just like a wrapper. The graph file has the same function as the PIT one, of course for PITSuperNet.

The `unit_test` folder is organized as follows:

- The `models` folder contains some simple test models used within the unit tests.
- The `test_methods` folder contains the tests developed for all the DNAS methods available. Each method has its own folder with some tests developed to ensure an easier implementation of new features. Some PITSuperNet tests include the model conversion in import and export mode, the individuation of the right number of supernet target modules, and the `get_size` and `get_macs` value test.
- The `test_utils` folder contains some additional tests utils as a file that tests some operations performed on the graph.

4.3 PITSuperNet

This section will describe the characteristics of the PITSuperNet method explaining how it works, how it can be used and the challenges encountered in its implementation.

4.3.1 PITSuperNetModule

The first and most evident element of the PITSuperNet method is the `PITSuperNetModule`. The `PITSuperNetModule` is what defines the so-called *supernet* i.e., a neural network that contains an arbitrarily large number of possible sub-networks or candidate architectures that can be obtained by applying different combinations of operations or transformations (such as convolutional layers, pooling, skip connections, etc.) to the supernet's architecture. `PITSuperNetModule` is the API that PLiNIO exposes to allow the user to define the different choices to be explored. These PITSuperNetModules can be used as drop-in replacements for some of the seed layers.

The basic idea is to enlarge the NAS search space by providing some valuable alternative options to be explored and the NAS will eventually select one of these options available replacing the `PITSuperNetModule` with the selected option. Inside each `PITSuperNetModule`, there is an instance of the `PITSuperNetCombiner` class that combines all the output of the candidate modules into a single one (more in Section 4.3.2). In Figure 4.3, a simple example of the idea of a supernet module is provided. A common pattern

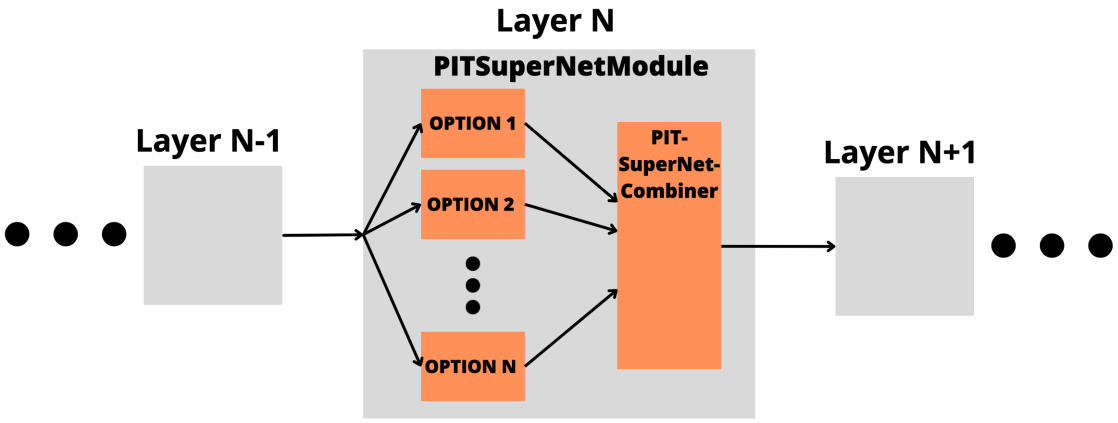


Figure 4.3. PITSuperNetModule.

that was frequently used in the experiments of this work consists of replacing a convolutional layer of the seed model with a `PITSuperNetModule` (Figure 4.3). This `PITSuperNetModule` could be provided with a list of possible operations such as:

- The original layer's convolution.
- A convolution with an increased kernel size with respect to the original one.
- A depthwise-separable convolution.
- An identity operation to enable the NAS to completely skip the layer if not needed.

In the `PITSuperNetModule` example reported in Figure 4.4, each alternative module has been paired with an independent BatchNorm module because in the tests conducted during the implementation of PITSuperNet, a shared

```
# self.conv1 = nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1)  # original layer
self.conv1 = PITSuperNetModule([
    # conv3x3 (original)
    nn.Sequential(
        nn.Conv2d(16, 16, 3, padding='same'),
        nn.BatchNorm2d(16),
    ),
    # conv5x5
    nn.Sequential(
        nn.Conv2d(16, 16, 5, padding='same'),
        nn.BatchNorm2d(16),
    ),
    # depthwise separable conv
    nn.Sequential(
        nn.Conv2d(16, 16, 3, groups=16, padding='same'),
        nn.BatchNorm2d(16),
        nn.ReLU(),
        nn.Conv2d(16, 16, 1),
        nn.BatchNorm2d(16),
    ),
    # identity to skip the layer if needed
    nn.Identity()
])
```

Figure 4.4. PITSuperNetModule usage example.

BatchNorm operation for all the alternative modules was found to be less effective (see Section 5.1.1).

This additional step required to be performed by the user is needed to ensure the flexibility of the tool making it useful in different kinds of tasks and applicable to various models. In this way, the user can directly choose the kind of operations he wants the NAS to consider for any specific layer. In this preparation phase, it is up to the user to ensure that all the alternative layers are compatible with their predecessors and successors e.g., an identity layer cannot be listed in a `PITSuperNetModule` that is positioned in a place where was originally intended an operation with the number of output channels different from the number of input channels. For this reason, all alternative modules in the supernet should share the same output tensor dimension.

`PITSuperNetModule` also takes as input two additional parameters `gumbel_softmax` and `hard_softmax`. They are two flags used to determine which type of softmax function to apply on the architectural weights α. Their usage will be better described in Section 4.3.3.

4.3.2 Model's Conversion

The `PITSuperNet` constructor accepts a PyTorch model with a variable number of `PITSuperNetModule`s inside and deals with the conversion of the model into a NAS-able one. This is a crucial step that converts also all the eligible layers into PIT ones. The PITSuperNet tool can also work just with the SuperNet part avoiding using the PIT optimization, if this is the case needed the user can set the `autoconvert_layers` flag to `False` while calling the constructor. In this way, the convert function will not try to transform all convolutional and fully connected layers into `PITModule`s and the NAS will just optimize the choice of the more suitable module in the supernet layers.

In the following, the most relevant operations performed inside the `convert` function (Figure 4.5) to build a NAS-able model are reported.

The first step is symbolic tracing and the creation of the graph of the model using the `torch.fx` tool. Symbolic tracing consists of a "symbolic execution" of the model code. During this execution, fake input values are fed through the model and are recorded (traced) in order to build a graph of the model composed of operations as nodes. An instance of the class `torch.fx.Tracer` inspects each module determining whether it is a leaf module or not. If it is a leaf module a special node is inserted in the model graph otherwise the tracing continues by entering the forward function of the current module and recursively inspecting all possible child modules. The tracer class can be overridden to implement various behaviors of this tracing process.

Some important labeling operations are then performed on the nodes of the graph in order to enable the correct propagation of the PIT feature masks.

The first operation is the `add_node_properties` which performs a visit on the graph and adds to a dictionary contained in each node some properties. These properties mostly define how the node propagates the features to its successors i.e., whether it defines a new number of output features different from what it receives (features defining operation) or simply propagates the features received (features propagating operation). These properties will become useful in the next steps.

The second operation is the `set_combiner_properties` and it is a PITSuperNet-specific operation that set the node properties of the combiners' nodes. An instance of the `PITSuperNetCombiner` class is contained in each `PITSuperNetModule` and contains the module logic. In fact, the `PITSuperNetModule` is just a wrapper for the combiner needed to allow the `fx.Tracer` to get inside each alternative module in order to apply PIT conversion. Using a special implementation of the `fx.Tracer` called

```python
def convert(model: nn.Module, input_shape: Tuple[int, ...], conversion_type: str,
            exclude_names: Iterable[str] = (),
            exclude_types: Iterable[Type[nn.Module]] = ()
            ) -> Tuple[nn.Module, List, List]:

    if conversion_type not in ('import', 'autoimport', 'export'):
        raise ValueError("Unsupported conversion type {}".format(conversion_type))

    tracer = PITSuperNetTracer()
    graph = tracer.trace(model.eval())
    name = model.__class__.__name__
    mod = fx.GraphModule(tracer.root, graph, name)
    batch_example = torch.stack([torch.rand(input_shape)] * 1, 0)
    device = next(model.parameters()).device
    ShapeProp(mod).propagate(batch_example.to(device))
    add_node_properties(mod)
    set_combiner_properties(mod, add=['shared_input_features', 'features_propagating'])
    # first convert Conv/FC layers to/from PIT versions first
    pit_graph.convert_layers(mod, conversion_type, exclude_names, exclude_types)
    if conversion_type in ('autoimport', 'import'):
        # then, for import, perform additional graph passes needed mostly for PIT
        pit_graph.fuse_conv_bn(mod)
        add_features_calculator(mod, [pit_graph.pit_features_calc, combiner_features_calc])
        set_combiner_properties(mod, add=['features_defining'], remove=['features_propagating'])
        associate_input_features(mod)
        pit_graph.register_input_features(mod)
        # lastly, import SuperNet selection layers
        sn_combiners = import_sn_combiners(mod)
        pit_modules = find_other_pit_modules(mod)
    else:
        export_graph(mod)
        sn_combiners = []
        pit_modules = []

    mod.graph.lint()
    mod.recompile()
    return mod, sn_combiners, pit_modules
```

Figure 4.5. convert function.

`PITSuperNetTracer` the `PITSuperNetCombiner` instead is treated as a leaf module and remains untouched by the tracing operation and it is able to provide the needed logic to train a SuperNet module.

The following step of the convert function is to call a PIT-specific function used to convert all the eligible layers into `PITModules`. Then another PIT-specific function is used to fuse together PIT convolution and PIT batch norm operations coming one after the other.

Then the `add_features_calculator` operation is called and with another visit on the graph it adds to each node dictionary an object called

`FeatureCalculator` in one of its variants depending on the properties that have been added to the node in the `add_node_properties` step.

The `FeatureCalculator` is an abstract class that defines the main interface to be exposed to all of its children that are used to compute the number of input features for a layer during the NAS optimization phase. This is needed because during the search phase PIT could mask parts of the previous layer and since different operations propagate features in different ways there is a need for different kinds of `FeatureCalculator`s.

The following operation is the `associate_input_features` that again for each node sets in the dictionary a field to tell what node within the preceding ones has set the input features of the current node.

The `register_input_features` function set for all the `PITModule`s an input `FeatureCalculator` taking the `FeatureCalculator` of the node within the preceding ones that set the input features.

Lastly, the `import_sn_combiners` function individuates all the `PITSuperNetCombiner`s and adds them to a list to be easily accessible in future operations and the `find_other_pit_modules` function does the same for all the `PITModule`s that are outside of a `PITSuperNetModule` (this is needed to make sure that they are counted in the `get_size` or `get_macs` operations that are described in Section 4.3.3).

4.3.3 Search Phase

During the search phase, each `PITSuperNetCombiner` forwards a weighted sum of the outputs of all the alternative modules. The weights applied to each input of the weighted sum are the architectural parameters that are all initialized to a constant value and then updated during each step of the training loop. The higher the weight of a certain module within the alternative ones, the higher its contribution to the output of that particular supernet layer. The output of a PITSuperNet layer y_{sn} has this form:

$$y_{sn} = \sum_{i=0}^{N} \theta_{\alpha_i} y_i \tag{4.1}$$

$$\theta_{\alpha_i} = softmax(\alpha_i) \tag{4.2}$$

Where y_i are the outputs of each candidate module contained in the PITSuperNetModule and θ_{α_i} are the architectural parameters of that `PITSuperNetModule` α_i that have been passed through a softmax function.

Supernet candidate modules could have also been converted to `PITModules` and in this case, their output dimensions depend on the masks that are currently applied to them. However, since all `PITSuperNetModule` candidates should share the same output dimension, PIT is forced to choose a shared mask for the output of all alternative layers. This constraint could indeed represent a limiting factor for the PITSuperNet method since it can become harder for PIT to find the perfect fine-grained optimization for a layer having to share that choice with all the other candidate layers.

In PITSuperNet it is possible to specify the type of softmax function to be applied to the alpha parameters within two types: standard softmax and Gumbel softmax [28] [29]. Standard softmax has the form:

$$softmax(\alpha_i) = \frac{\exp(\frac{\alpha_i}{\tau})}{\sum_j \exp(\frac{\alpha_j}{\tau})} \tag{4.3}$$

Conversely, Gumbel softmax has the following expression:

$$GumbelSoftmax(\alpha_i) = \frac{\exp((\log(\alpha_i) + g_i)/\tau)}{\sum_{j=1}^{k} \exp((\log(\alpha_j) + g_j)/\tau)} \tag{4.4}$$

Where g are samples drawn from the Gumbel(0,1) distribution and τ is the temperature value. The temperature value τ is used both in standard softmax and Gumbel softmax. This value can be used to change the confidence in the choices made. If the temperature value is below 1 the softmax will output more confident results i.e., more polarized values, while a higher value (more than 1) will decrease the confidence of the choices and provide as output less polarized values.

Optionally when using the Gumbel softmax, it is possible to set the flag `hard_softmax=True`. In this case, after the sampling from the distribution, an argmax is performed rather than a softmax and consequently, a one-hot vector is obtained as output. Since a discretization would not be differentiable, the backward function is rewritten with a straight-through estimator (i.e., the backward of the operation in question is rewritten as if it were the backward of an identity function) to ensure that the operation remains differentiable.

Gumbel softmax can be applied to face the problem of unpolarized alpha values that was discovered while testing the PITSuperNet method. The alpha values passed through the standard softmax function append to be very similar to each other causing the supernet choices not to be really polarized towards a particular candidate module. During the search phase, little differences in the alpha values of a supernet module can lead to an output that

is the result of the contribution of all (or most of) the candidate modules in equal measure creating a sort of mixture layer that, as soon as the model is exported (Section 4.3.4), disappears replaced by the module with the slightly higher alpha value (see Section 5.1.2). This selected module does not well represent what the NAS has been used to "see" during the search phase eventually leading to a model that is not well optimized and a consequent accuracy drop.

To make sure that the NAS searches for the best tradeoff between accuracy and model size (or number of MACs) a regularization term can be added to the loss computation into the training loop. The loss takes the form:

$$\mathcal{L} = \mathcal{L}_{task} + \lambda\mathcal{R} \tag{4.5}$$

Where $\mathcal{L}_{task}$ is the loss on the task, λ is the regularizer strength and $\mathcal{R}$ is the regularization term. The regularization term is obtained by calling the function `get_regularization_loss`. Depending on the regularizer chosen, this function calls the `get_size` or `get_macs` function of PITSuperNet. This function retrieves the size (or MACs) of all the layers involved in the NAS optimization both supernet and pit layers. For the pit modules, the size is computed based on the current mask applied to the module in the search phase. For the supernet layers, the size (or MACs) is computed again as a weighted sum of the sizes (MACs) of all the candidate modules in the same way as the forward function computes the output of the `PITSuperNetModule`. The current size of a `PITModule` inside a supernet layer is computed and passed through the weighted sum as any other module contained in the `PITSuperNetModule`.

The parameter λ is used as a multiplying factor to tune the effect of the regularization term on the loss. Usually between 0 and 1, the higher it is (closer to 1), the higher the effect of the regularization term on the loss, and consequently the NAS will output more constrained models in terms of size or MACs. On the other hand, with a low λ value the regularization term has less impact on the loss thus allowing the NAS to output bigger models.

Another loss term that can be added to the loss computation is loss icv. With this term the loss assumes this form:

$$\mathcal{L} = \mathcal{L}_{task} + \lambda\mathcal{R} + \lambda_{icv}\mathcal{R}_{icv} \tag{4.6}$$

Where λ_{icv} is again the strength of this regularizer and $\mathcal{R}_{icv}$ is the icv regularization term. This particular term is used as a regularizer to encourage the NAS to take more polarized decisions during the search phase by maximizing the variance between the alpha values. The loss icv term consists of

the reciprocal of the variance of the architectural parameters values to allow maximizing the variance while still minimizing the loss:

$$\mathcal{R}_{icv} = \sum_{i=0}^{L} \frac{\mu_{\theta_\alpha}^{(i)}}{var_{\theta_\alpha}^{(i)} + \epsilon} \tag{4.7}$$

Where L is the number of `PITSuperNetModule`s in the model, $\mu_{\theta_\alpha}^{(i)}$ the mean of the architectural parameters of a particular `PITSuperNetModule`, $var_{\theta_\alpha}^{(i)}$ the variance of the same parameters and ϵ a small number to avoid divisions by zero. Loss icv represents an alternative to the Gumbel softmax to tackle the problem of the alpha values similarity (see Section 5.1.2 for the comparision).

4.3.4 Final Architecture Export

When the search phase is over the architecture discovered must be converted into a normal PyTorch model. To accomplish this task PLiNIO exposes the `arch_export` function. In the `arch_export`, the `convert` function, already described in Section 4.3.2, gets called in export mode. In this mode, the `convert` function calls the PIT-specific function to convert all `PITModule`s into standard modules with the optimized number of channels. In this function, all `PITModule`s both inside a `PITSuperNetModule` or in other layers are replaced with their standard PyTorch operation e.g., `PITConv2d` gets replaced by `Conv2d`, `PITLinear` by `Linear`, etc. All these replacing modules are built with the optimized number of channels defined by PIT. At this point, all `PITModule`s inside the model have been replaced while the supernet layers still contain all their alternatives. Then a PITSuperNet-specific function gets called to export all the supernet layers. In this step for each supernet layer, the candidate module that is paired with the highest architectural parameter α gets chosen and replaces the supernet layer. Since all the `PITModule`s have already been exported, the chosen module will be also PIT optimized.

The resulting model containing optimized supernet choices and modules optimized by PIT is a standard PyTorch model that can be fine-tuned on the task at hand.

Suppose the user requires to perform only the supernet search and sets `autoconvert_layers=False`. In that case, only the second part of the export will be performed, the one related to the choice of the candidate layer in each `PITSuperNetModule` while there will be no PIT optimization and consequently no need to export `PITModule`s.

Chapter 5

Results

This chapter presents the results of the experimental validation conducted on PITSuperNet with the datasets available in the MLPerf Tiny benchmarks suite. In particular, the tests have been conducted on three of the four benchmarks available which are Image Classification (ICL), Keyword Spotting (KWS), and Visual Wake Words (VWW). The fourth benchmark available in the MLPerf Tiny suite is Anomaly Detection and it has not been considered in this work since its seed network is a dense autoencoder, therefore composed of only fully-connected layers on which the application of SuperNet would not be meaningful.

As said in Section 4.1, the thesis' objective is combining coarse- and fine-grained DNAS. To do so, two approaches have been explored: a complete mixture of the two methods with the new PITSuperNet method, and the execution of a PIT search over some of the models obtained with a SuperNet-only search. In order to provide a full significative comparison of all the methods each benchmark has been used to test PIT, SuperNet, PITSuperNet, and finally PIT after SuperNet. Please note, that thanks to the modularity of the developed code, a SuperNet search can be performed with the PITSuperNet method specifying `autoconvert_layers=False` (Section 4.3.2).

For each benchmark, the reference seed model has been modified to contain some PITSuperNet modules. In the ICL benchmark all convolutional operations have been replaced by a PITSuperNetModule except for the ones contained in the residual branches of the network. For KWS and VWW all depthwise separable convolutional operations have been substituted by a PITSuperNetModule leaving untoched all other operations. An example of the first layers of the modified model for PITSuperNet is provided in Figure 5.2 compared to the original seed model in Figure 5.1.

```
class DSCnn(torch.nn.Module):
    def __init__(self):
        super().__init__()

        # Model layers

        # Input pure conv2d
        self.inputlayer = nn.Conv2d(
            in_channels=1, out_channels=64, kernel_size=(10, 4), stride=(2, 2), padding=(5, 1))
        self.bn = nn.BatchNorm2d(64, momentum=0.99)
        self.relu = nn.ReLU()
        self.dropout1 = nn.Dropout(p=0.2)

        # First layer of separable depthwise conv2d
        # Separable consists of depthwise conv2d followed by conv2d with 1x1 kernels
        self.depthwise1 = nn.Conv2d(
            in_channels=64, out_channels=64, kernel_size=3, padding=1, groups=64)
        self.pointwise1 = nn.Conv2d(
            in_channels=64, out_channels=64, kernel_size=1, stride=1, padding=0)
        self.bn11 = nn.BatchNorm2d(64, momentum=0.99)
        self.relu11 = nn.ReLU()
        self.conv1 = nn.Conv2d(
            in_channels=64, out_channels=64, kernel_size=1, stride=1, padding=0)
        self.bn12 = nn.BatchNorm2d(64, momentum=0.99)
        self.relu12 = nn.ReLU()

        # Second layer of separable depthwise conv2d
        self.depthwise2 = nn.Conv2d(
            in_channels=64, out_channels=64, kernel_size=3, padding=1, groups=64)
        self.pointwise2 = nn.Conv2d(
            in_channels=64, out_channels=64, kernel_size=1, stride=1, padding=0)
        self.bn21 = nn.BatchNorm2d(64, momentum=0.99)
        self.relu21 = nn.ReLU()
        self.conv2 = nn.Conv2d(
            in_channels=64, out_channels=64, kernel_size=1, stride=1, padding=0)
        self.bn22 = nn.BatchNorm2d(64, momentum=0.99)
        self.relu22 = nn.ReLU()

        # Third layer of separable depthwise conv2d
        self.depthwise3 = nn.Conv2d(
            in_channels=64, out_channels=64, kernel_size=3, padding=1, groups=64)
        self.pointwise3 = nn.Conv2d(
```

Figure 5.1. First layers of DSCNN model (KWS).

```
class DSCnnPITSN(torch.nn.Module):
    def __init__(self):
        super().__init__()

        # Model layers

        # Input pure conv2d
        self.inputlayer = nn.Conv2d(
            in_channels=1, out_channels=64, kernel_size=(10, 4), stride=(2, 2), padding=(5, 1))
        self.bn = nn.BatchNorm2d(64, momentum=0.99)
        self.relu = nn.ReLU()
        self.dropout1 = nn.Dropout(p=0.2)

        # First layer of separable depthwise conv2d
        # Separable consists of depthwise conv2d followed by conv2d with 1x1 kernels
        self.depthpoint1 = PITSuperNetModule([
            nn.Sequential(
                nn.Conv2d(64, 64, 3, padding='same'),
                nn.BatchNorm2d(64, momentum=0.99),
                nn.ReLU()
            ),
            nn.Sequential(
                nn.Conv2d(64, 64, 5, padding='same'),
                nn.BatchNorm2d(64, momentum=0.99),
                nn.ReLU()
            ),
            nn.Sequential(
                nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1, groups=64),
                nn.BatchNorm2d(64, momentum=0.99),
                nn.ReLU(),
                nn.Conv2d(in_channels=64, out_channels=64, kernel_size=1, stride=1, padding=0),
                nn.BatchNorm2d(64, momentum=0.99),
                nn.ReLU()
            ),
            nn.Identity()
        ], gumbel_softmax=True, hard_softmax=True)
        self.conv1 = nn.Conv2d(
            in_channels=64, out_channels=64, kernel_size=1, stride=1, padding=0)
        self.bn12 = nn.BatchNorm2d(64, momentum=0.99)
        self.relu12 = nn.ReLU()
```

Figure 5.2. First layers of DSCNN model (KWS) modified for PITSuperNet.

Each PITSuperNetModule has been equipped with 3 or 4 candidate modules containing:

- a 3x3 convolution
- a 5x5 convolution
- a depthwise separable convolution
- an identity operation (possibly)

Each of these candidate modules has its own BatchNorm and ReLU operation (see Section 5.1.1)

In the following, all the results reported have been obtained, unless otherwise specified, using the Gumbel softmax described in Section 4.3.3. The tables reported contain the Pareto points corresponding to architectures found with the specified method, each point is described by the λ value i.e., regularizer strength, used during the search phase, the number of parameters contained in the resulting architecture, and the accuracy result obtained on the task. Each accuracy result obtained is an average of 6 fine-tuning results obtained with different seed values.

As described in Section 4.3.3, PITSuperNet supports both size and macs as regularizers. With size, the regularization term is the number of parameters of the layers involved in the search phase, while with macs the number of multiply and accumulate operations of the same involved layers is used. For time reasons, the tests conducted in this thesis work have only used the size regularizer. The usage of the macs regularizer was tested in the early stages of this work and gave similar results to the one obtained with size but a full exhaustive comparison is still to be done.

5.1 Preliminary Experiments

5.1.1 Shared BatchNorm vs Independent BatchNorm

Each one of the PITSuperNetModule candidate modules has been equipped with its own BatchNorm and ReLU operation because sharing these operations with all the candidate modules sometimes caused some troubles in the fine-tuning phase of the training. For instance, in the KWS benchmark, for some lambda values the model found using a shared BatchNorm operation got stuck in the fine-tuning phase to an accuracy value of 8.34%. This value is clearly really low compared to the 91.88% of the reference model but most importantly is almost random for a 12-class task. The same problem occurred in the VWW benchmark where some architectures could not improve from a 52% accuracy value during the fine-tuning, again an almost random accuracy value since VWW is a binary task. By using an independent BatchNorm operation for each candidate module in the PITSuperNetModule this issue was fixed.

5.1.2 Softmax, Gumbel Softmax, Loss ICV

In this section, the results of the techniques employed to face the problem of non-polarized architectural parameters α, described in Section 4.3.3, will be reported.

alpha branch 0	alpha branch 1	alpha branch 2	alpha branch 3
0.22	0.14	0.32	0.31

Table 5.1. Standard softmax alpha values.

In Table 5.1 a sample of the architectural parameters α values for a PITSuperNetModule in the final step of a search phase are reported. Since the 4 values are similar to each other all 4 candidate layers will play a significant part in determining the output of the weighted sum computed in the forward function of the combiner. When the export function gets called, it will select branch number 2 since it has a slightly higher value suddenly changing what the model had been used to see and optimize during the search phase.

In Table 5.2 the same α values obtained in a search phase using loss icv are reported. This time the values are definitely polarized towards a single choice i.e., branch 2, thus solving the problem highlighted before with the

standard softmax. Another option consists of using the Gumbel softmax

alpha branch 0	alpha branch 1	alpha branch 2	alpha branch 3
0.00	0.01	0.97	0.02

Table 5.2. Loss ICV alpha values.

with the hard softmax option 5.3. Using this option the α values consist of a one-hot vector as described in Section 4.3.3.

alpha branch 0	alpha branch 1	alpha branch 2	alpha branch 3
0.0	0.0	1.0	0.0

Table 5.3. Gumbel hard softmax alpha values.

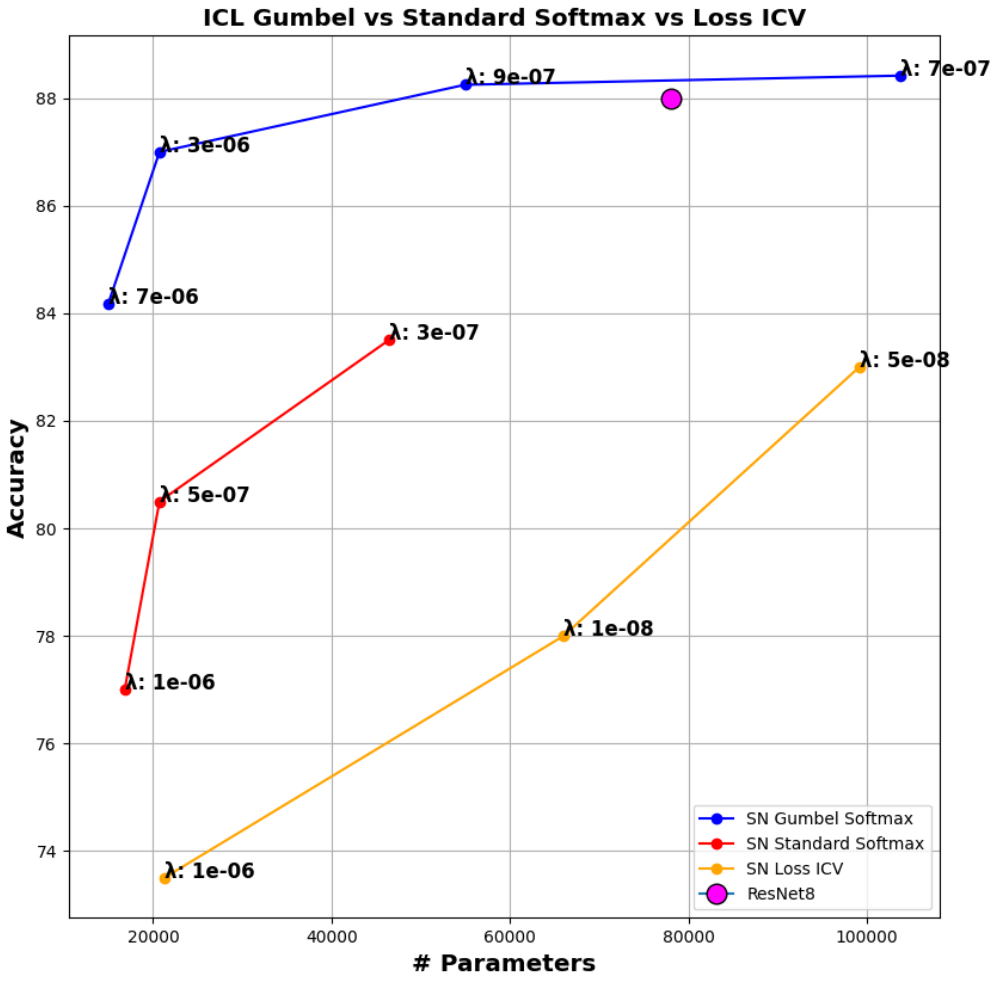


Figure 5.3. ICL SuperNet results with diffrent softmax functions and loss icv.

Lambda (λ)	Model's size (# parameters)	Accuracy
3e-7	46414	83.5
5e-7	20718	80.5
1e-6	16814	77.0

Table 5.4. ICL SuperNet with Standard Softmax Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
5e-8	99166	83.0
1e-8	66014	78.0
1e-6	21258	73.5

Table 5.5. ICL SuperNet with Loss ICV Pareto points.

Figure 5.3 shows the comparison between the three different techniques employed while testing the SuperNet method on the ICL benchmark. Compared to the ones found with the Gumbel softmax (blue curve, results in Section 5.2) the other two curves clearly show some significant accuracy drops. The standard softmax results show how the non-polarization of the α values during the search phase could lead to not well-optimized models. One possible solution to this problem is the usage of a loss icv (described in Section 4.3.3), however, the search phase with this technique converged too fast to architectures that are clearly not performant. While it is probably possible to solve this problem by tuning the strength of the loss icv regularizer, in this thesis this solution has not been explored since the Gumbel softmax technique offered way better results right away. Similar results have been found on the KWS benchmark.

5.2 Image Classification (ICL)

ICL is an image classification task available in the MLPerf Tiny benchmarks suite. Image classification is a task that consists of classifying images into some predefined classes or categories. It represents one of the most relevant tasks of Computer Vision (CV) and also one of the first tasks where Deep Learning raised lots of attention because it was able to outperform many other methods. The dataset used in this task is CIFAR-10 [30] which consists of 60000 32x32 RGB images in 10 classes, with 6000 images per class. The 10 classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The dataset is divided into five training batches and one test batch, each with 10000 images. The reference or seed model used in the tests is ResNet-8 [17] composed of 78052 parameters it achieves an 88% maximum accuracy on the task. The reference model has been modified by replacing each convolutional operation with a PITSuperNetModule except for the convolutions of the residual branches of the network.

Lambda (λ)	Model's size (# parameters)	Accuracy
5e-7	62543	87.5
5e-6	35326	86.67
1e-5	31218	85.42
3e-5	17222	83.17
5e-5	11400	78.08

Table 5.6. ICL PIT Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
7e-7	103722	88.42
9e-7	54954	88.25
3e-6	20714	87
5e-6	14954	84.17

Table 5.7. ICL SuperNet Pareto points.

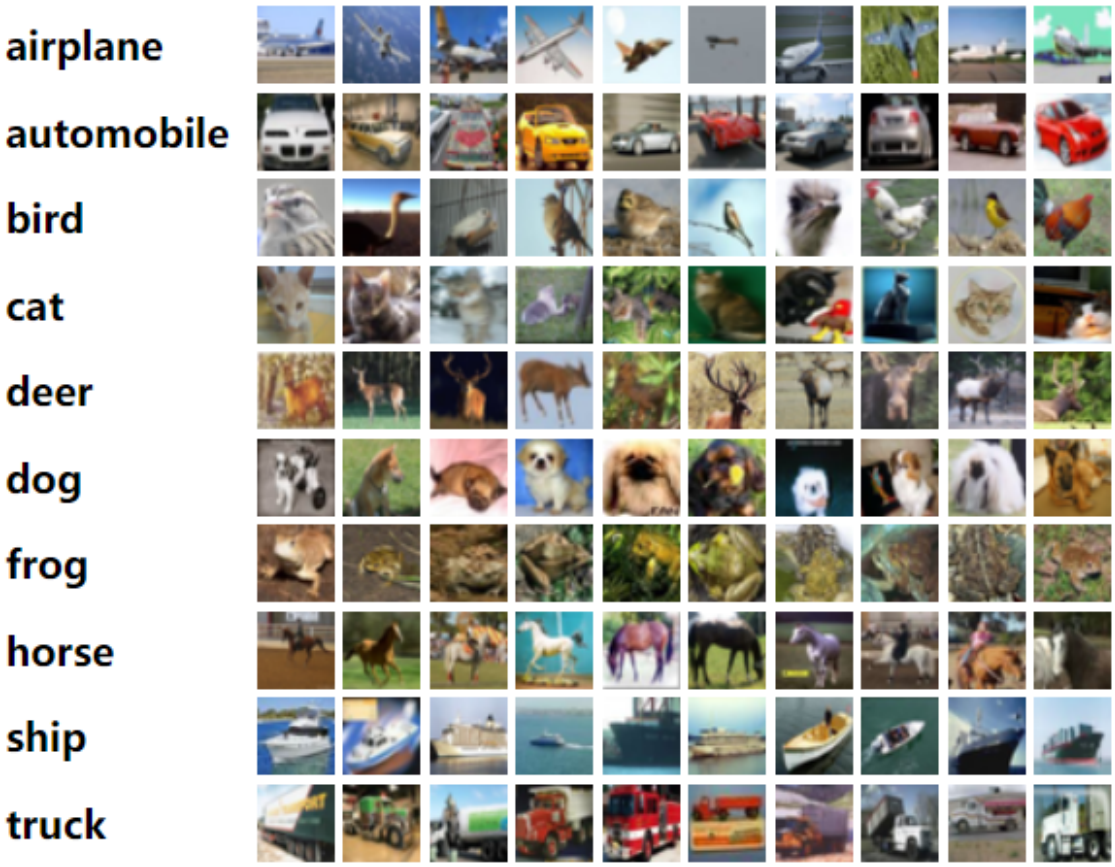


Figure 5.4. CIFAR-10 [30] samples.

Lambda (λ)	Model's size (# parameters)	Accuracy
9e-7	78338	88.5
2e-6	39568	84.75
3e-6	38828	82.58
5e-6	16810	81.5

Table 5.8. ICL PITSuperNet Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
5e-7	45330	88.5
1e-5	35489	88
2e-5	14729	80.7
3e-5	9266	77.2

Table 5.9. ICL PIT after SuperNet λ=9e-7 Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
1e-6	20157	85
1e-7	17483	83.5
3e-5	14290	81.2
5e-5	7854	79.1

Table 5.10. ICL PIT after SuperNet λ=3e-6 Pareto points.

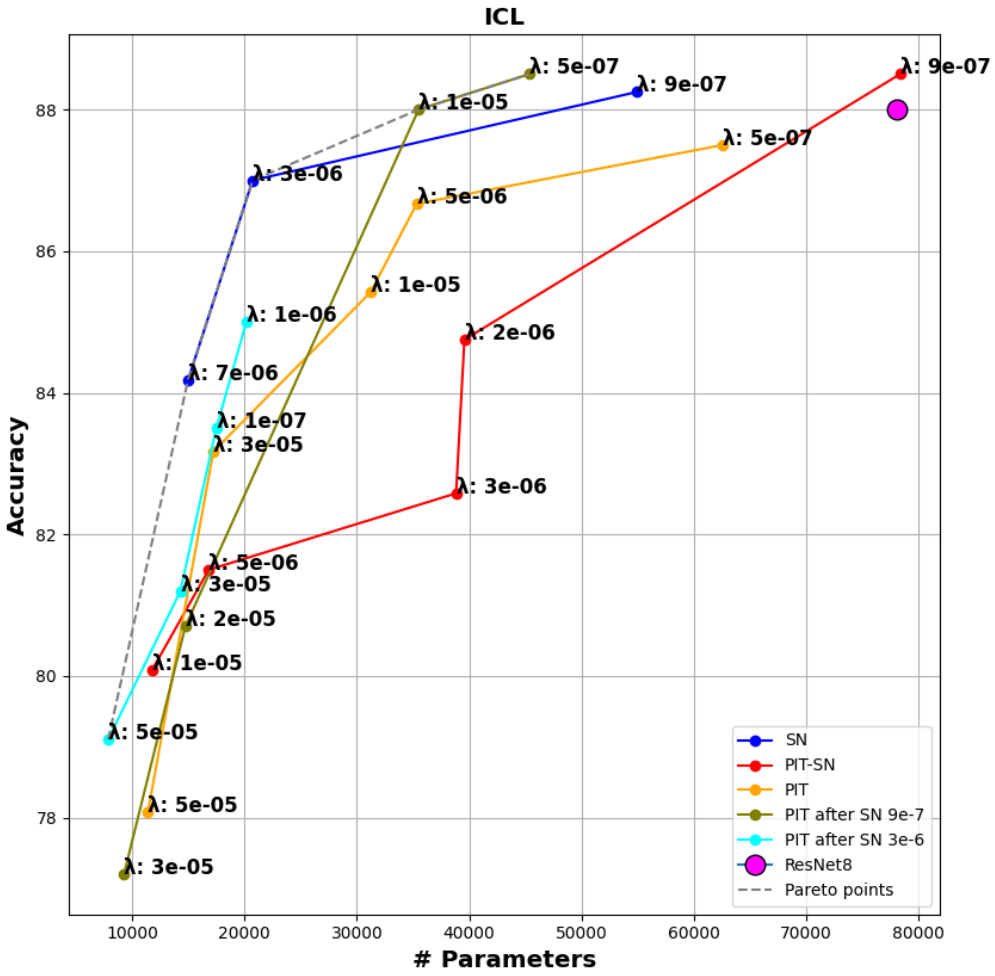


Figure 5.5. ICL results.

By looking at the blue curve in Figure 5.5 we can see that SuperNet achieved an 88.25% accuracy with a 54954 parameters model (λ = 9e-7) that compared to the original ResNet-8 model corresponds to a size reduction of 29.6% with 0.25% accuracy improvement. By performing a fine-grained search (see green curve) on the model obtained with the coarse-grained one performed by SuperNet a further accuracy improvement was found bringing the total improvement of accuracy compared to the ResNet-8 model to 0.5% (λ=5e-7). This improvement comes with an even smaller model of only 45330 parameters corresponding to a size reduction of 42% with respect to ResNet-8. In the same green curve, an even smaller model was found (λ=1e-5) with 35489 parameters corresponding to a size reduction of 54.5% and a final accuracy result that matches the one of ResNet-8.

More in general, while the approach of performing the fine-grained search after the coarse-grained one was effective for some of the points just mentioned improving both PIT (orange curve) and SuperNet (blue curve) results, the approach of mixing and performing both searches at the same time led to unsatisfying results with significant accuracy drops (see the red curve of PITSuperNet).

The reasons for PITSuperNet's unsatisfying results could be related to the complexity of the optimization problem since PITSuperNet must optimize a significant number of architectural parameters having to manage both PIT and SuperNet parameters. Moreover, the fine-grained optimization performed by PIT could be suffering from having to select a shared mask for all the candidate layers of a PITSuperNetModule, thus probably having to find a trade-off between the actual more suitable mask that each candidate layer would prefer for itself.

5.3 Keyword Spotting (KWS)

Keyword spotting is a task that consists of detecting phrases or keywords within audio signals representing utterances. Keyword spotting allows for the identification of specific commands given with the voice thus being particularly useful for modern voice assistants like Apple's Siri [31], Amazon's Alexa [32], or Google Assistant [33]. These devices need to remain idle while waiting to detect specific wake-up words while consuming as little energy as possible. As soon as a keyword is detected they wake up and perform the speech processing. The dataset employed in the keyword spotting task available in the MLPerf Tiny suite is the Speech Commands dataset [34]. The classes of the dataset are represented by the following list of 10 words: "Yes", "No", "Up", "Down", "Left", "Right", "On", "Off", "Stop", and "Go". Two additional classes bring the count to 12 and they are "Silence" and "Unknown". Each sample has a duration of one second and since most of the samples simply represent silence or background noise the false positives must be minimized in order to obtain good accuracy results. The reference model for the KWS task is a Depthwise Separable Convolutional Neural Network (DSCNN) composed of 40396 parameters and achieves a 91.88% top-1 accuracy. The reference model has been modified by replacing each depthwise separable convolutional operation with a PITSuperNetModule.

Lambda (λ)	Model's size (# parameters)	Accuracy
1e-7	40255	92.36
3e-5	22802	91.52
5e-5	12482	88.95
7e-5	9082	86.13

Table 5.11. KWS PIT Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
1e-6	40908	92.05
9e-6	35852	90.64
3e-5	25740	88.72
7e-5	20684	84

Table 5.12. KWS SuperNet Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
5e-6	47058	89.95
1e-5	18438	88.82
2e-5	11023	87.56
3e-5	7623	84.20

Table 5.13. KWS PITSuperNet Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
4e-6	32607	91.32
1e-5	24794	91.23
2e-5	13647	88.85
3e-5	11895	88.46

Table 5.14. KWS PIT after SuperNet λ=9e-6 Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
1e-5	18699	88.52
3e-5	14111	87.88
4e-5	8669	84.99

Table 5.15. KWS PIT after SuperNet λ=3e-5 Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
2e-5	18593	83.26
3e-5	14971	83.07
4e-5	10718	82.72

Table 5.16. KWS PIT after SuperNet λ=7e-5 Pareto points.

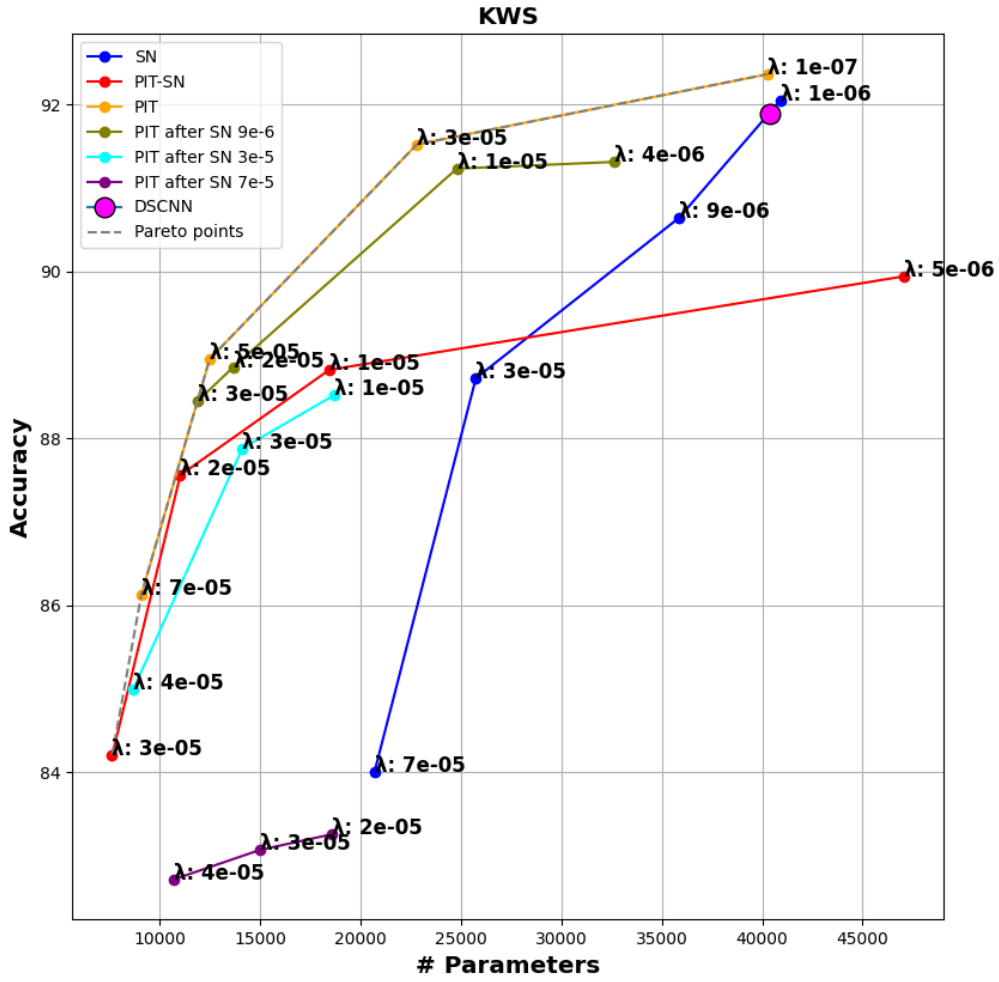


Figure 5.6. KWS results.

In this benchmark, the combined coarse and fine-grained search did not provide any benefit either with the mixed approach or the separated one. These results show how the baseline hand-tuned model is already well-suited for the task and trying to find different alternative architecture layers with a SuperNet coarse-grained search (blue curve in Figure 5.6) does not provide any improvements. The fine-grained search performed by PIT appears to be the best approach to further improve, with small adjustments, an already well-performing architecture and in fact, none of the other methods could outperform PIT's results (orange curve).

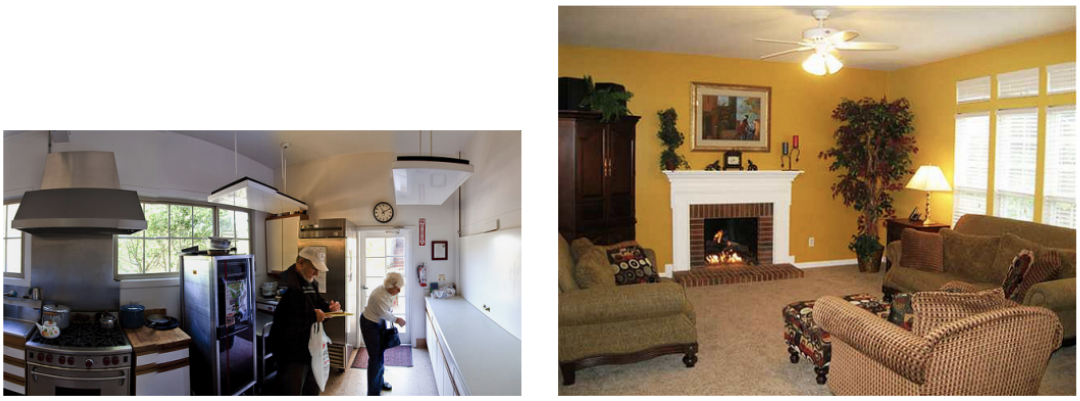
PIT was able to find an accuracy improvement of almost 0.5% with a model of 40255 parameters (λ=1e-7) corresponding to a small size reduction of 0.3% with respect to the seed model. Another interesting point found by PIT is with λ=3e-5 where a model of just 22802 parameters (size reduction: 43.6%) achieved an accuracy of 91.52% just 0.36% lower than the one achieved by the seed model.

Moreover, it is worth pointing out how again the separated approach to combine the coarse- and fine-grained search seemed to be more effective than the mixed approach with PITSuperNet even though in this case it did not provide improvements to the fine-grained search.

5.4 Visual Wake Words (VWW)

Visual Wake Words is a classification task that aims to detect whether a person is present in an image or not. It is a particularly useful task for the IoT camera sensors that are becoming more and more popular with time. The dataset employed is the Visual Wake Words Dataset [35], a subset of the MSCOCO dataset [36], composed of 115000 images and 2 classes that are "Person" and "Not-person". The task has been called visual wake words because in the same fashion as the keyword spotting task, the involved devices get woken up by the presence of a person in the frame with the necessity of consuming little power while being idle. Whenever a person is detected, higher computing resources are delivered to enable image processing. The reference model for this task is MobileNetV1 [37] composed of 213586 parameters and archives an 83.32% top accuracy. The reference model has been modified by replacing each depthwise separable convolutional operation with a PITSuperNetModule.

This benchmark has a significantly bigger dataset and a much deeper seed model compared to ICL and KWS and for this reason the training times required are noticeably higher.



(a) 'Person' (b) 'Not-person'

Figure 5.7. VWW dataset [35] samples.

Lambda (λ)	Model's size (# parameters)	Accuracy
1e-10	13736	84.82
1e-8	12751	84.7
1e-6	9914	84.49
1e-5	2564	80.78

Table 5.17. VWW PIT Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
7e-8	155786	84.82
9e-8	105674	84.68
1e-6	55418	83.52
1e-5	49554	82.01

Table 5.18. VWW SuperNet Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
1e-10	13908	85.14
5e-7	10581	84.19
1e-6	8881	83.86
5e-6	4148	81.76

Table 5.19. VWW PIT after SuperNet λ=7e-8 Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
5e-7	10585	84.36
1e-6	9280	84.32
5e-6	8074	83.80
1e-5	4485	81.69

Table 5.20. VWW PIT after SuperNet λ=9e-8 Pareto points.

Lambda (λ)	Model's size (# parameters)	Accuracy
1e-6	4253	83.76
5e-6	3933	83.68
1e-5	2903	82.29

Table 5.21. VWW PIT after SuperNet λ=1e-6 Pareto points.

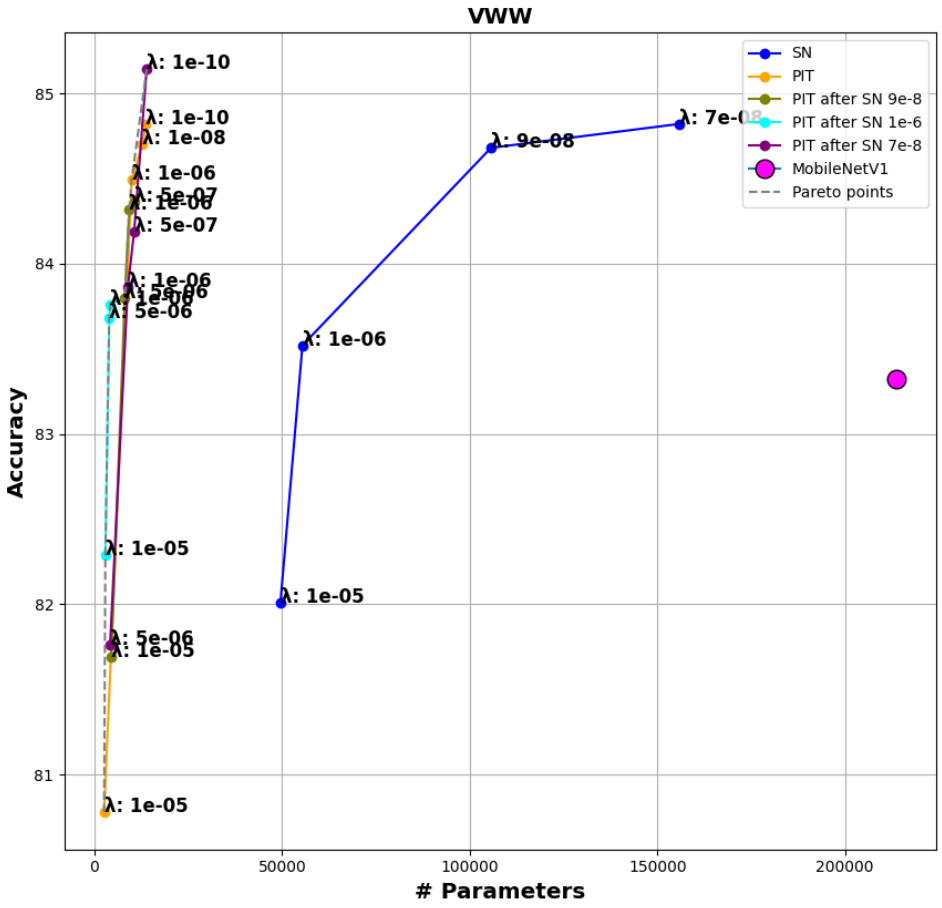


Figure 5.8. VWW results.

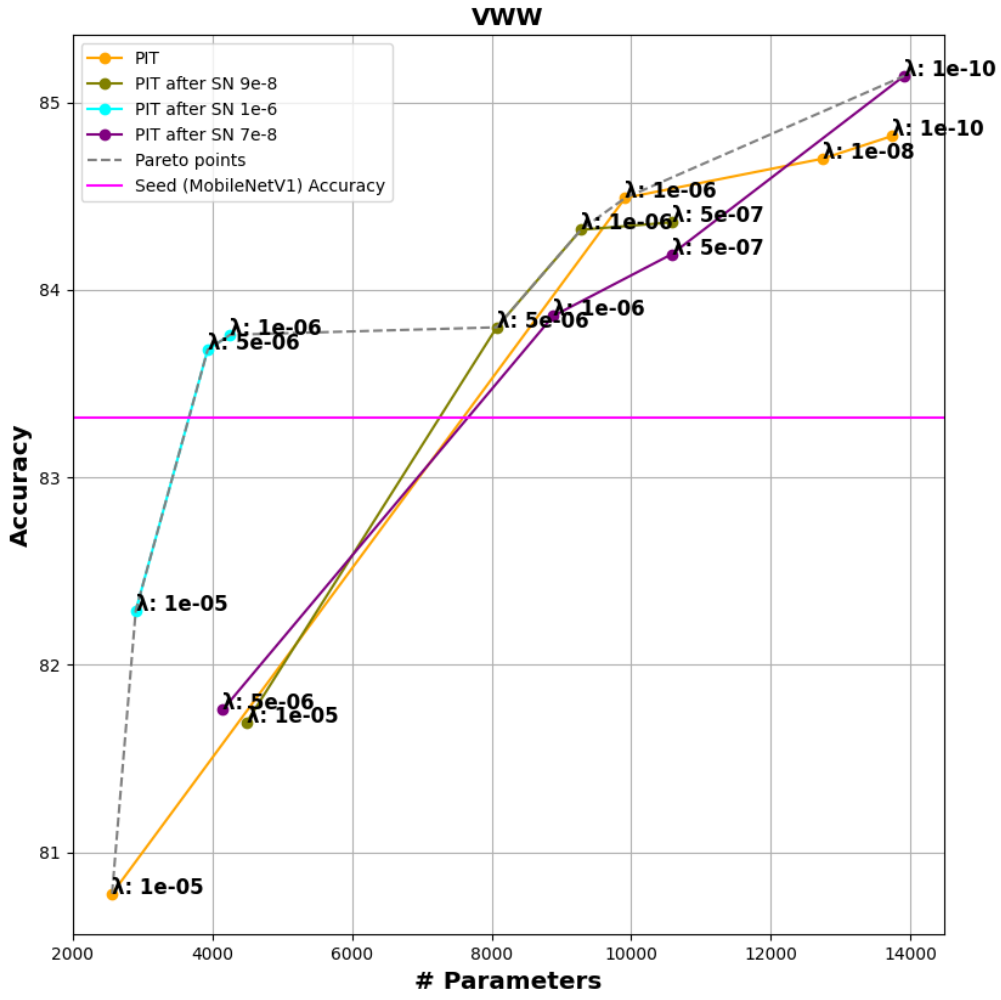


Figure 5.9. VWW results (zoom PIT).

The results in Figure 5.8 highlight how the model employed in this task has plenty of room for improvements and optimization. By looking at PIT results (orange curve), it is possible to see how the architectures found are significantly shrunk down in terms of size while being able to reach higher accuracy results. For instance, the architecture obtained with λ=1e-10 is made of 13736 parameters corresponding to a size reduction of 93.6% and an accuracy improvement of 1.5%. SuperNet alone is able to find more performing models than the seed MobileNetV1 but it does not reach the level of size reduction obtained by PIT for the same performance.

To test the combination of the coarse- and fine-grained search for this benchmark only the second sequential approach has been tried, while PIT-SuperNet has not been tested for time reasons (VWW benchmark has significantly higher training times compared to ICL and KWS) and because it was not providing improved results in the preceding benchmarks.

Noteworthy, by cascading the output of λ=7e-8 SuperNet output with PIT (purple curve in Figure 5.9) with λ=1e-10 we achieved an architecture of 13908 parameters (93.5% size reduction) and an accuracy of 85.14% which is 1.8% higher than the seed one.

Another interesting result is shown by the light blue curve in Figure 5.9 which corresponds to PIT applied on the SuperNet model obtained with λ=1e-6. The architecture found with λ=1e-6 has only 4253 parameters and is still able to increase the accuracy by 0.4% with the huge size reduction of 98%.

Chapter 6

Conclusions and Future Works

The number of Artificial Intelligence applications has been significantly growing over the past few decades allowing the automation of a large number of different tasks that before were only possible with human intervention. This growth has been driven by the increasing availability and improvements of computing power and by the emergence of new Deep Learning techniques that have made possible a higher level of automation. Deep Learning methods automate the extraction of useful information from the data thus requiring less human intervention and knowledge to perform the tasks, however, lots of expertise and time are still required to design Neural Networks that are best suited for the needed task. For this reason, a lot of effort is being given to the objective of automating the process of designing Neural Networks in order to avoid a time-consuming manual trial-and-error approach. Neural Architecture Search represents the answer to this quest, and Differentiable NAS is certainly one of the most promising approaches in this field. To meet the increasing demand for AI applications on IoT edge devices, NAS techniques have to consider not only the model performance but also some other metrics order to be deployed on small devices such as to name some, the size of the model, the latency, the memory occupation, and the power consumption.

This work contributed to the development and expansion of the PLiNIO library which aims to provide a user-friendly tool to enable the comparison and testing of different DNAS methods in order to ease the research in the NAS field and the deployment of AI on small edge devices. In particular, firstly a SuperNet method has been developed to allow a coarse-grained kind

of search different from the fine-grained one performed by the already existing PIT algorithm. Secondly, the SuperNet method has been transformed into PITSuperNet, a new method able to combine the two types of search. PITSuperNet still allows for a SuperNet-only search if required but it also enables one to perform a PIT search at the same time as the SuperNet one. The combination of the coarse- and fine-grained search has been tested on three of the MLPerf Tiny benchmarks following two approaches: a mixture of the two methods and a sequential application of SuperNet and PIT. The first mixed approach did not give the hoped results while the second showed some improvements over both the fine-grained- or coarse-grained-only search.

In summary, with the combination of the fine- and coarse-grained search it was possible to obtain for the ICL benchmark an architecture of 45330 parameters corresponding to a size reduction of 42% with respect to the seed model and an accuracy improvement of 0.5% or an architecture of 35489 parameters corresponding to a size reduction of 54.5% with the same accuracy of ResNet-8, the seed model. For the VWW benchmark, it was possible to obtain an architecture of 13908 parameters i.e., a 93.5% size reduction with respect to MobileNetV1 (seed model) with a 1.8% accuracy improvement or a 4253 parameters architecture corresponding to a 98% size reduction with an accuracy improvement of 0.4%.

As already said, the first mixed approach of combining a fine- and coarse-grained search with PITSuperNet did not provide the hoped results however, since this approach of PITSuperNet is for sure way more convenient than having to perform N PIT searches for each SuperNet point found, it could be interesting in future works to further examine the PITSuperNet algorithm, for instance, trying new training scheduling techniques which could possibly consist of freezing PIT parameters for a predefined number of epochs at the start of the training.

Bibliography

[1] Hubara, I., Courbariaux, M., Soudry, D., & Bengio, Y. (2016). Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. arXiv. https://doi.org/10.48550/arXiv.1609.07061

[2] Blalock, D., Ortiz, J. J., Frankle, J., & Guttag, J. (2020). What is the State of Neural Network Pruning?. arXiv. https://doi.org/10.48550/arXiv.2003.03033

[3] Cai, H., Zhu, L., & Han, S. (2018). ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. arXiv. https://doi.org/10.48550/arXiv.1812.00332

[4] Gordon, A., Eban, E., Nachum, O., Chen, B., Wu, H., Yang, T., & Choi, E. (2017). MorphNet: Fast & Simple Resource-Constrained Structure Learning of Deep Networks. arXiv. https://doi.org/10.48550/arXiv.1711.06798

[5] Wan, A., Dai, X., Zhang, P., He, Z., Tian, Y., Xie, S., Wu, B., Yu, M., Xu, T., Chen, K., Vajda, P., & Gonzalez, J. E. (2020). FBNetV2: Differentiable Neural Architecture Search for Spatial and Channel Dimensions. arXiv. https://doi.org/10.48550/arXiv.2004.05565

[6] M. Risso et al., "Lightweight Neural Architecture Search for Temporal Convolutional Networks at the Edge" in IEEE Transactions on Computers, vol. 72, no. 3, pp. 744-758, 1 March 2023, doi: 10.1109/TC.2022.3177955.

[7] eml-eda. PLiNIO Github Repository. https://github.com/eml-eda/plinio

[8] Banbury, C., Reddi, V. J., Torelli, P., Holleman, J., Jeffries, N., Kiraly, C., Montino, P., Kanter, D., Ahmed, S., Pau, D., Thakker, U., Torrini, A., Warden, P., Cordaro, J., Di Guglielmo, G., Duarte, J., Gibellini, S., Parekh, V., Tran, H., . . . Xuesong, X. (2021). MLPerf Tiny Benchmark. arXiv. https://doi.org/10.48550/arXiv.2106.07597

[9] Frank Rosenblatt. «The perceptron: a probabilistic model for information storage and organization in the brain.» In: Psychological review (1958).

[10] ASSESSMENT OF ARTIFICIAL NEURAL NETWORK FOR BATHYMETRY ESTIMATION USING HIGH RESOLUTION SATELLITE IMAGERY IN SHALLOW LAKES: CASE STUDY EL BURULLUS LAKE. - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network_fig4_303875065

[11] Dubey, S. R., Singh, S. K., & Chaudhuri, B. B. (2021). Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark. arXiv. https://doi.org/10.48550/arXiv.2109.14545

[12] cardiGAN: A Generative Adversarial Network Model for Design and Discovery of Multi Principal Element Alloys - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/ReLU-activation-function-vs-LeakyReLU-activation-function_fig2_358306930

[13] Haykin, S. (1994). Neural networks: a comprehensive foundation. Prentice Hall PTR.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. Commun. ACM 60, 6 (June 2017), 84–90. https://doi.org/10.1145/3065386

[15] Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv. https://doi.org/10.48550/arXiv.1409.1556

[16] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2014). Going Deeper with Convolutions. arXiv. https://doi.org/10.48550/arXiv.1409.4842

[17] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. arXiv. https://doi.org/10.48550/arXiv.1512.03385

[18] Deep Cybersecurity: A Comprehensive Overview from Neural Network and Deep Learning Perspective - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/An-example-of-a-convolutional-neural-network-CNN-or-ConvNet\-including-multiple_fig5_350216987

[19] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv. https://doi.org/10.48550/arXiv.1502.03167

[20] Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. arXiv. https://doi.org/10.48550/arXiv.1412.6980

[21] Geoffrey Hinton In Lecture 6e of his Coursera Class https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[22] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12:2121–2159, 2011.

[23] David E. Rumelhart; James L. McClelland, "Learning Internal Representations by Error Propagation," in Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations, MIT Press, 1987, pp.318-362.

[24] Zoph, B., & Le, Q. V. (2016). Neural Architecture Search with Reinforcement Learning. arXiv. https://doi.org/10.48550/arXiv.1611.01578

[25] From federated learning to federated neural architecture search: a survey - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/An-overview-of-RL-based-NAS-method_fig4_348224791

[26] Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., & Dean, J. (2018). Efficient Neural Architecture Search via Parameter Sharing. arXiv. https://doi.org/10.48550/arXiv.1802.03268

[27] Liu, H., Simonyan, K., & Yang, Y. (2018). DARTS: Differentiable Architecture Search. arXiv. https://doi.org/10.48550/arXiv.1806.09055

[28] Jang, E., Gu, S., & Poole, B. (2016). Categorical Reparameterization with Gumbel-Softmax. arXiv. https://doi.org/10.48550/arXiv.1611.01144

[29] Maddison, C. J., Mnih, A., & Teh, Y. W. (2016). The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. arXiv. https://doi.org/10.48550/arXiv.1611.00712

[30] A. Krizhevsky, V. Nair, and G. Hinton. «Cifar-10». (canadian institute for advanced research). (2009) https://www.cs.toronto.edu/~kriz/cifar.html

[31] Apple's Siri. https://www.apple.com/it/siri/

[32] Amazon's Alexa. https://developer.amazon.com/it-IT/alexa

[33] Google Assistant. https://assistant.google.com/intl/it_it/

[34] Warden, P. (2018). Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. arXiv. https://doi.org/10.48550/arXiv.1804.03209

[35] Chowdhery, A., Warden, P., Shlens, J., Howard, A., & Rhodes, R. (2019). Visual Wake Words Dataset. arXiv. https://doi.org/10.

48550/arXiv.1906.05721

[36] Lin, T., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., & Dollár, P. (2014). Microsoft COCO: Common Objects in Context. arXiv. https://doi.org/https://arxiv.org/abs/1405.0312v3

[37] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv. https://doi.org/10.48550/arXiv.1704.04861