POLITECNICO DI TORINO

Master degree course in Data Science and Engineering

Master Degree Thesis

Differentiable Neural Architecture Search Algorithms for MLPerf Tiny benchmarks

Politecnico di Torino

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

Candidate
Fabio Eterno
matricola: 286787

Anno accademico 2021-2022

A Isabella e Ugo

Summary

Nowadays, Artificial Intelligence (AI), especially in the form of Machine Learning (ML) and Deep Learning (DL), is becoming the go-to approach to solve complex problems in several sectors such as Computer Vision (CV), Speech Recognition, Natural Language Processing (NLP) and many others. Despite the huge effort spent by public and private actors to reach state-of-the-art results, the design of Deep Neural Networks (DNNs) is still a manual process heavily based on empirical rules and heuristics, thus requiring designers with strong expertise.

This inspired researchers to define a new set of algorithms called Neural Architecture Search (NAS). NAS algorithms are becoming very popular in the MLPerf Tiny/TinyDL domain where the choice of the specific ML/DL model structure is of primal importance. Indeed, deploying DNNs on tiny devices (e.g., small microcontrollers, IoT nodes, etc.) requires considering not only the final accuracy reached by the model, but also the hardware constraints in terms of memory footprint, latency, and energy consumption related to the target device.

This thesis focuses on the development of a toolkit to facilitate future developers in the training and evaluation phases of a family of state-of-the-art NAS techniques called mask-based Differentiable NAS (DNAS) for MLPerf Tiny use-cases.

First, this thesis pursues the development of FlexNAS i.e., a flexible library for testing and comparing different DNAS techniques. In particular, a complete set of unit-tests has been designed to verify the correct behavior of different steps involved in the search-phase of DNAS.

Second, we consider the industrial grade MLPerf-Tiny benchmark suite. The tasks therein represent industrial-relevant use-cases for which it is relevant to explore and measure the trade-offs between accuracy, latency, and energy of DL networks when deployed on embedded devices. This benchmark suite has been originally developed in TensorFlow. Due to the different

libraries involved (PyTorch for FlexNAS and TensorFlow for MLPerf Tiny benchmarks), a complete refactor of MLPerf-Tiny scripts has been necessary to make them compatible with the FlexNAS ecosystem.

The final library allows the user to easily compare FlexNAS over MLPerf Tiny datasets, and it is designed to be easily extended to other NAS-able models and benchmarks, constituting a solid basis for future research.

Acknowledgements

Ripensando a questo percorso universitario è facile fare parallelismi con il mondo sportivo. In questo senso, il ciclismo secondo me è quello che si accosta meglio a questa esperienza. Uno sport dove viene premiata la costanza, dove l'aspetto mentale è importante quanto quello fisico, uno sport di squadra e allo stesso tempo tremendamente individuale.

Del percorso appena concluso le prime persone che voglio ringraziare sono i miei relatori di tesi Daniele Jahier Pagliari, Matteo Risso e Alessio Burrello. Grazie per avermi dato la possibilità di approfondire il tema della Neural Architecture Search, ho apprezzato moltissimo la disponibilità, la competenza e l'aiuto fornitomi durante questi mesi di tesi.

Voglio ringraziare il mio capo Giuseppe Gennaro per avermi dato la possibilità di mettere in pratica nel mondo lavorativo le tecnologie apprese in questo corso di laurea e i miei colleghi d'ufficio in Reale Ites.

Grazie a mia mamma Piera e alle mie tre sorelle Cristina, Martina e Agnese per essermi sempre state a ruota e avermi "tirato" nei momenti difficili.

Grazie a tutta la mia famiglia, da San Maurizio Canavese a Specchia, passando per gli zii di Torino, per avermi sempre sostenuto in questo percorso. Un pensiero a nonna Maria Teresa, che voleva essere sempre aggiornata su come andavano gli esami quando scendevo, sarà contenta di sapere che mi sono laureato.

Grazie a mia moglie Isabella, paragonabile all'ammiraglia nel gergo del ciclismo. Esattamente come in una grande corsa a tappe, senza di te raggiungere il traguardo non sarebbe stato possibile, letteralmente. Grazie per avermi supportato e sopportato sempre, soprattutto nei momenti più complicati.

Grazie a mio papà Ugo, che è stato uno studente lavoratore prima di me e senza smart working. Spero tu sia orgoglioso di me.

Vorrei chiudere con un pensiero ai miei nipoti Alice, Beatrice, Anita, Giulio e Lorenzo. Spero che il vostro percorso scolastico vi aiuti a realizzarvi come persone e ad essere felici.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last few decades, the global society has experienced a revolution in the sector of information technology. The rapid pace of progress in communication and computing technologies is shaping new scenarios, creating opportunities that were unthinkable a few years ago. This period has been defined as the fourth industrial revolution [1]. Recalling the previous industrial revolutions, it is evident that the speed at which the last one is progressing is far way much higher than the previous three. In 1765 the invention of the steam engine and the massive extraction of coal moved the economy from agriculture to industry, introducing the mechanization of the work. The second industrial revolution started a century later, around 1870, and it has been driven by the discovery of electricity, gas, and oil that led to the development of the combustion engine. It is necessary to wait another century before seeing the third industrial revolution, that started in 1969 with nuclear energy and electronics.

The advent of internet and information technology broke the precedent status quo. Before the internet, information goods were following the rules of the physical goods, with many constraints in terms of storage, distribution and transportation. Internet transformed the information goods into non-rivalry goods making negligible the costs of storage, distribution and replication.

Within this domain, we can identify several enabling technologies that are fueling the fourth industrial revolution:

- *cloud computing* which breaks the link between the physical hardware infrastructure and the software.

- *big data architectures* which paired to the cloud computing allow a horizontal scalability.
- *internet of things (IoT)* which is supported by an increasing coverage of the mobile broadband.

Nevertheless, today the main innovation driver is represented by AI. Historically different disciplines have been recognized as AI. Nowadays, the term is mainly used as a synonym for Machine Learning and Deep Learning. *Machine Learning* (ML) identifies a set of algorithms whose goal is to solve in an automatic way tasks of various kinds: in a classification task the output is a label, in a regression task the output is a number, in a clustering task the objective is to find group of objects with similar characteristics. In general, a ML model is trained to identify patterns and extract meaningful analytics on data. Usually, conventional ML techniques require several sequential steps such as preprocessing, feature extraction and feature selection. Those steps usually are strictly related to the domain and the type of problem to be solved and a long and manual preliminary design phase is required.

Deep Learning (DL) [2] represents an evolution of ML algorithms where the feature engineering phase is performed automatically within the training of the model. In this way, there is no need of human intervention avoiding possible harmful biases during features selection. In the last few years, DL models started to outperform ML techniques achieving new state-of-the-art results in many different sectors such as Computer Vision (CV) [3], Natural Language Process (NLP) and Time Series Analysis [4, 5].

The DL models productivity keeps increasing thanks to the availability of huge amount of data and with the improvement of the parallel computational power allowing the solution of tasks of increasing complexity. In the past, Graphic Processing Units (GPUs) were special-purpose application accelerators, developed mainly for graphical applications. Modern GPUs are fully programmable for general-purpose data intensive processing [6] reinvigorating the academic and industial research in DL and reducing the time required for the training of the models compared to the usage of Central Processing Units (CPUs).

DL models are resource hungry not only regarding computing power, but also pose high requirements in terms of memory (RAM and storage disk) and energy consumption. These two latter aspects are usually less monitored when the computation is executed on a powerful cloud server where the hardware is assumed sufficient. In fact, the possibility to change on demand the underlying physical hardware thanks to virtualization to better serve the

actual workload is a key aspect of the success of cloud computing. However, hardware constraints gain a huge importance in the context of the Internet of Things. The application of Deep Learning in low-power embedded devices and IoT sensors due to their constrained nature can be significantly challenging. In fact, the optimal solution needs to be found within the complex envelope of contrastive requirements in terms of accuracy, memory footprint, latency and energy requirements.

The declination of the previous problem is the definition of the best trade-off between two opposite approaches: a completely centralized computation, where all the inferences are calculated on a cloud server, and a completely decentralized approach, where all the inferences are performed by edge devices. In the first approach, all the raw data are sent from the edge to the cloud server, despite the latter has not hardware constraints some other aspects should be considered. The amount of information needed to be transmitted may require a huge bandwidth stressing a lot the network capacity. This problem is particularly true for computer vision tasks. Moreover, the latency between a request by the edge device and a response by the cloud is hardly predictable. For this reason low latency applications are not suitable in such scenario. Furthermore, the transmission of large amount of data through the network has an high energy consumption. While the required energy for computations is progressively diminishing, the same is not true for the transmission task where the required energy does not decrease at the same rate. Moreover, there are also privacy aspects to consider since transmitting raw data can create some concerns in terms of data protection.

On the other hand a completely decentralized approach is extremely challenging due to the hardware limitations related to IoT devices.

The best compromise is founding deep learning algorithms suitable to be deployed on low-power embedded devices in order to perform locally the inferences and reduce the amount of data to be transmitted to the cloud server. The advantages are a significant reduction of the latency and energy requirements and a higher level of protection of data.

Several optimizations have been proposed in the literature to tackle this challenge. Quantization techniques reduce the complexity of the numerical operations. A common and well-established quantization strategy consist in passing from floating point to fixed point representations [7, 8]. Another orthogonal direction of optimization regards modifying the model's architecture. DL architectures are highly redundant, with over-parametrized designs that try to have enough capacity to solve complex task. Pruning techniques

aim to remove unimportant parts to reduce the size of the network at a negligible cost in terms of accuracy [9, 10]. Another way to shrink the hardware demand by DL models is the applications of more efficient layers capable to reduce the number of parameters involved [11, 12].

The work of this thesis will focus on a family of optimizations techniques denoted as Neural Architecture Search (NAS) [13, 14]. Neural Networks (NN) present a high number of hyper-parameters whose selection is tricky and based on empirical intuitions and rules of thumb. The purpose of NAS is the optimization of the network topology to achieve the best accuracy with the simplest possible architecture. In particular, in the context of this thesis we will start from a state-of-the-art NAS, namely Pruning In Time (PIT) [15]. PIT during the training phase progressively tries to change the values of the network hyper-parameters such as number of output channels, receptive field size and dilation in order to minimize the memory footprint and the number of multiply-accumulate operations (MACs).

This NAS technique has been inserted into a novel library called FlexNAS. PIT is the first algorithm added into FlexNAS, but the objective of this work is to built a standard framework capable of easily integrate different NAS algorithms and evaluate them over several training datasets.

The high variety of neural networks provides a plethora of possible combinations among any type of layers. FlexNAS takes as input a Pytorch model and it substitutes the layers of interest into specific custom objects containing trainable architectural parameters.

To increase FlexNAS stability a set of unit tests has been designed to check different peculiarities:

- the correct translation of 1D and 2D convolutional layers, fully-connected layers and depthwise separable layers.
- the values of the masks to train the number of channels, the receptive field size and the dilation.
- the possibility to exclude specific layers from NAS search.
- the behavior of the regularization loss.
- the export of PIT model into a standard Pytorch model.

The second main part of this thesis is related to the MLPerf Tiny industrial benchmark library. MLPerf Tiny presents four use-cases specifically designed to evaluate both the accuracy and efficiency of deep learning models. Each

benchmark is based on a specific dataset and uses TensorFlow library to train and test a reference model. Our final purpose is testing the developed NAS library over these four tasks. In this thesis PIT will demonstrate its ability to discover new models capable to reach higher accuracy with less parameters.

A *conditio sine qua non* for the compatibility between FlexNAS and MLPerf Tiny suite has been the porting of MLPerf Tiny framework from TensorFlow to Pytorch, since FlexNAS accepts only the latter one. For each benchmark the data collection, the preprocessing phase, the model definition and the training and testing phase has been redesigned.

To demonstrate the reliability of the Pytorch version each benchmark has been equipped with an example script. The pytorch model performance are completely verifiable and replicable, and of course the final results are compliant with the original TensorFlow version.

To facilitate future extensions to other benchmarks a standard group of functions have been defined in order to provide a uniform interface. In this way the complexity has been handled at lower level and final users can easily evaluate the same technique passing from one benchmark to another.

The developed FlexNAS version of PIT has been tested over the four tasks within the benchmark library. The obtained results will be further described in Chapter 6.

Chapter 2

Background

Machine Learning refers to a set of technologies and algorithms able to learn autonomously how to solve a task starting from a set of input examples. The goal is to extract the knowledge contained into data to solve specific tasks. ML is typically applied to automate repetitive jobs, usually performed by humans. Moreover, ML can be exploited to handle tasks which cannot be managed by humans due to the huge size and complexity of the input data (i.e., it would be unfeasible carry out an efficient fraud detection activity using human resources to check all the transactions).

Deep Learning denotes a subset of Machine Learning algorithms where the feature engineering phase is completely automatized. DL models achieved great results especially in fields where the manual extraction of features is a strong limitation in terms of performance. This is particularly true in fields such as Computer Vision and Speech Recognition.

In the following section the mathematical foundations of DL models are detailed.

2.1 Neural Network fundamentals

2.1.1 Neuron

The neuron is the basic unit of neural networks. A DL neuron is an atomic component which performs a weighted sum of the input data x_i followed by a non-linear activation function h which returns a scalar output y:

$$y = h(\sum_i w_i x_i + b) \tag{2.1}$$

where w_i and b_i are commonly identified as weights and biases. These terms are tuned in the training process in order to obtain a properly trained NN.

2.1.2 Activation Functions

The activation function h of Eq. 2.1 represents a crucial element of neurons and, in general, in DL architectures. In particular, an activation function bounds the output of the weighted sum and introduces non-linearity in the training process. This aspect is fundamental to learn non-linear mappings within input data and the particular task to be solved.

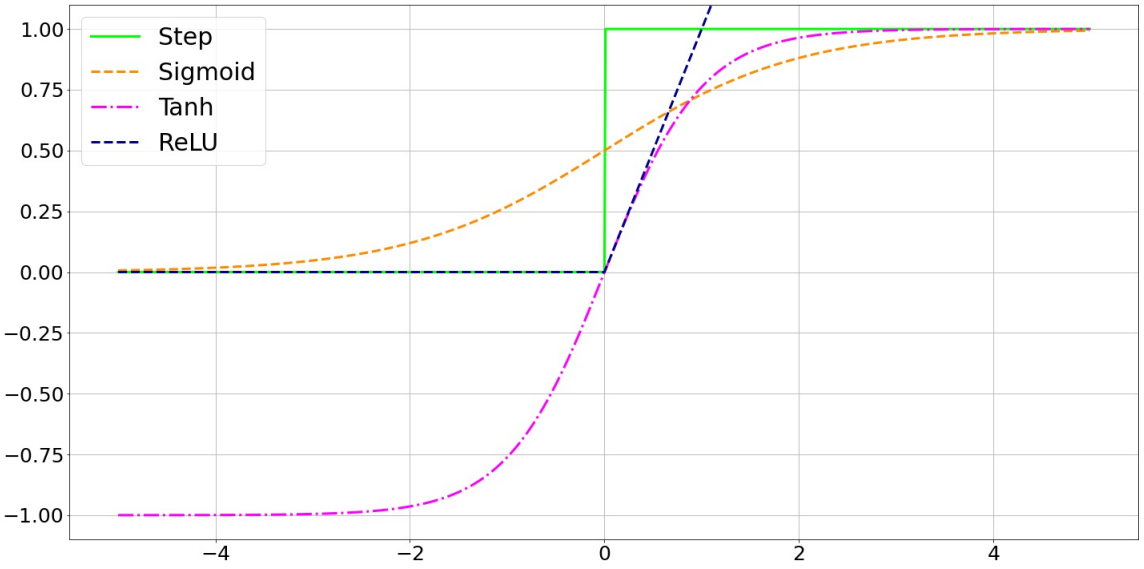


Figure 2.1: Most common activation functions.

The most common activation functions are:

- **Step Function**, represented in Figure 2.1 in green, is the first activation function proposed in the literature:

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

  Today, this *step* function is not used anymore due to the non-derivable point in $x = 0$. As will be described in Sec. 2.1.4, in order to properly train a NN is important avoiding non-differentiable functions.

- **Sigmoid Function**, represented in Figure 2.1 in orange, has been designed to substitute the previous step function. The *sigmoid* output range is between [0,1] and it is fully derivable.

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

  It bounds any functions given in input preventing the explosion of the gradients. The drawback of this activation function is the low response to inputs which are close to the positive and negative extremes of the function. If the input values are not close to 0 the *sigmoid* response saturates and this causes the so-called vanishing gradient problem, hampering the training of weights and biases. For these reasons, the *sigmoid* is no more used inside architectures but is only applied as output layer in binary classification tasks.

- **Hyperbolic Tangent Function (*tanh*)**: The *tanh* has a s-shape as the *sigmoid* but it ranges between [-1, +1] (it is represented in Figure 2.1 in fuchsia).

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

  The *tanh* can be described as a *sigmoid* function with an extended codomain. This provides more stable gradients and better performance, however also the *tanh* suffers of vanishing gradient. Furthermore, the *tanh* solves the non-centrality problem of the *sigmoid*, indeed while the $sigmoid(0)=\frac{1}{2}$ the $tanh(0) = 0$, so the negative inputs remain mapped to the negative domain while in the *sigmoid* all the inputs are mapped to the positive domain. In the latter case, all the weights associated to a given *sigmoid* will be updated in the same manner (requiring more steps to converge), while with *tanh* during backpropagation phase (which will be described in Sec. 2.1.4) the weights are updated differently based on their original domain, either negative or positive. Moreover the derivatives of *tanh* around zero is greater compared to *sigmoid*, providing a better flow gradient.

- **Rectified Linear Unit (*ReLU*)**: *ReLU*, represented in Figure 2.1 in blue, is nowadays the most applied activation function of neural networks.

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x > 0 \end{cases} \tag{2.5}$$

Differently from *sigmoid* and *tanh*, the *ReLU* function does not show a saturating behavior avoiding the vanishing gradient problem. Nonetheless, this function is not upper bounded and may suffer of the dual problem of exploding gradients. *ReLU* looks like a linear function in the positive and negative domain, but it is nonlinear thanks to the non-derivable point in $x = 0$. The main drawback of the *ReLU* is related to kill neurons whose outputs are all negative since the *ReLU*'s output would be 0. To overcome this problem a different version of the *ReLU* has been designed called *Leaky ReLU* in which the negative outputs are put to values close to 0.

2.1.3 Loss Functions

In order to update iteratively the network parameters a function to evaluate the actual set of parameters is needed. This function is commonly known as loss function.

The objective of the training algorithm is to learn the best set of weights and biases in order to minimize the error computed through the loss function. A neural network is trained to match the distribution of input data. The loss function computes how similar is the predicted distribution with the respect to the target distribution of training data.

There is not a unique loss function, the choice depends on the typology of the task to be solved. The most common loss functions are:

- The **cross-entropy** loss evaluates the performance of a classification model which returns a probability value between 0 and 1 and it is defined as:

$$H(p, q) = -\sum_{x \in X} p(x) \log(q(x)) \tag{2.6}$$

  where p is the predicted probability and q the ground truth.

- The **Mean Square Error (MSE)** is the loss function usually used in the regression tasks. It minimizes the squared difference between the predicted and the correct target values:

$$L(y, \bar{y}) = \frac{1}{N} \sum_{i=0}^{N} (y - \bar{y})^2 \tag{2.7}$$

where $\bar{y}$ are the predicted values.

- The **Mean Absolute Error (MAE)** is another common loss function for regression tasks, the main difference with the respect to the MSE is the use of the absolute value instead of the power of 2:

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

The MSE in general is preferred because it does not treat the errors in the same ways, but it penalizes more bigger errors while do not emphasize too much small errors. Another reason to choose MSE is because MAE has a non derivable point in 0. The choice of MAE is more convenient in outlier detection scenario where the outlier error will be much greater than the normal sample.

2.1.4 Gradient-Based Learning

Training a neural network means finding iteratively the best set of model parameters capable to minimize the loss function. The high cardinality of the number of parameters to be trained make unfeasible a brute-force approach.

The first idea is to perform a random local search, so applying a little random perturbation to the weights values and comparing the loss function results before and after this action.

To choose which is the best direction along which update the model parameters the gradient of the loss function is computed. The gradient is the vector of the partial derivatives in each dimension and it corresponds to the direction of the steepest descent, the direction along with is possible to minimize the loss function.

$$f(x, y) = xy \quad \nabla f = [\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}] = [y, x] \tag{2.9}$$

The Gradient Descent algorithm provides the instrument to solve this optimization problem and it is defined as:

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

where θ represents the network parameters, η is the learning rate, $\mathcal{L}(\theta)$ is the loss function associated to θ and ∇ the corresponding gradient.

The hyper-parameter η is one of the most crucial one and it defines the size of the update step to be performed in the gradient direction. If the learning rate is too low (as represented in Figure 2.2 with the blue line) it would require a huge number of epochs before reaching the best result. On the other side with a learning rate too high the training of the network would diverge (as represented in Figure 2.2 with the yellow line) .

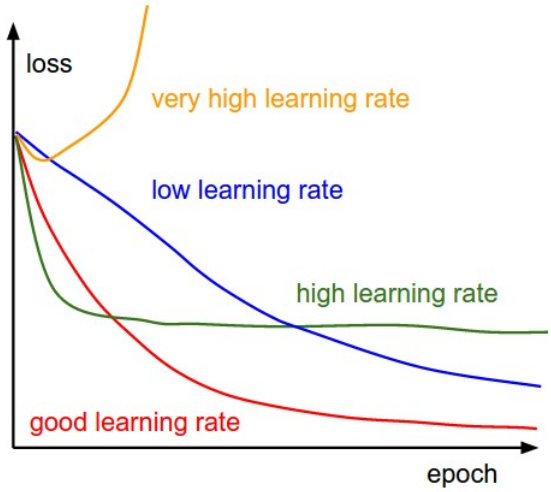


Figure 2.2: Learning rate behaviors.

From a theoretical point of view, the computation of the gradient would require to evaluate the loss function on the entire training set for each iteration. However, from a more pragmatic point of view this would require an high cost in terms of time and computational effort.

Furthermore calculating the exact gradient value is not the goal of the training phase, the most important purpose is discovering the best direction along which update the model weights. To reduce the training time usually the gradients are not computed over the whole training dataset, but using only a random subset of the input data. This approach is known as Stochastic Gradient Descent (SGD).

The training process follows the gradient direction applying the *backpropagation algorithm*, which is composed by two phases: the first one is the *forward pass*, where the batch of the input data is fed to the network and the value of the loss function is computed. The second phase is the *backward pass* where the gradients are evaluated using the chain rule:

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta q}\frac{\delta q}{\delta x} \tag{2.11}$$

The chain rule allows to compute the desired partial derivative through the multiplication of the intermediate gradients.

2.1.5 Regularization Techniques

An underestimated aspect of neural networks is the necessity of quite large datasets to achieve satisfactory results. The more a given dataset is representative of the real distribution of the data, the higher will be the performance of the trained model.

However, the models are often trained over datasets which are not large enough or not capable to incorporate a representative sample of the target distribution, these cause the so-called *overfitting*.

The overfitting happens when a general machine learning algorithm (not only neural networks, but also decision trees, support vector machines...) gains a very high accuracy over the training set, but then it performs very poorly with real input data. The explanation is quite simple, when the model is overfitted the network parameters have been trained such that the model can correctly recognize only data belonging to the training set without the capability to generalize to new cases and scenario.

To prevent overfitting usually an additional term is added to the loss function, called *regularization* loss. The purpose of this term is to somehow makes a bit harder the training of the NN in order to prevent the model to overfit the training set.

There are different regularization techniques:

- **L2 regularization**: L2 is the most common regularization technique and it consists in adding the squared magnitude of all the parameters in the loss function:

$$\mathcal{L}_2 = \lambda ||\theta||_2^2 = \lambda \sum \theta_i^2 \tag{2.12}$$

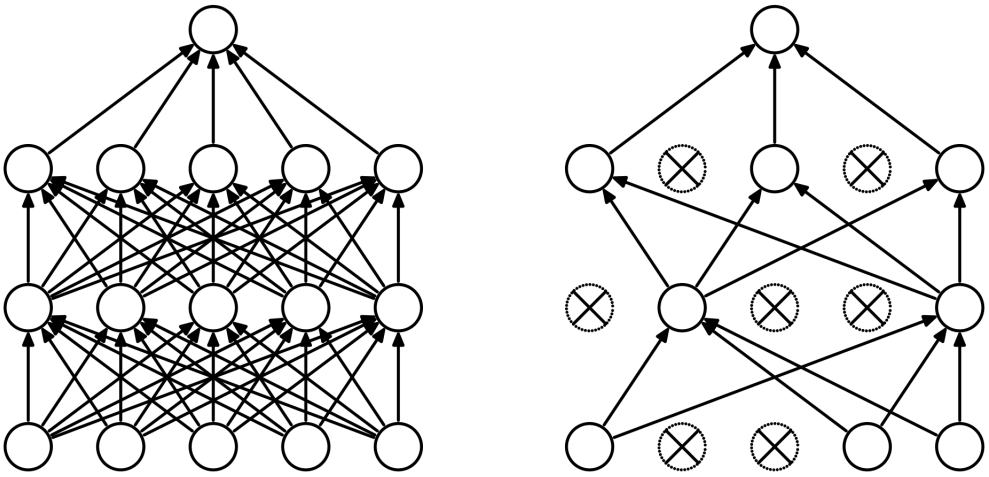
  L2 regularization discourages the accumulation of large weights in few neurons, it encourages a more regular diffusion of the network parameters fostering a uniform weights utilization.

- **L1 regularization**: L1 consists in adding to the loss function the L norm of the model parameters:

$$\mathcal{L}_1 = \lambda ||\theta||_1 = \lambda \sum |\theta|_i \tag{2.13}$$

  The main effect of the L1 regularization is to make more sparse the weight vector during the optimization, for this reason in general the L2 technique provides better performance.

- **Dropout**: the Dropout [16] is a regularization technique which simply turns off with probability p each neuron during the training of the model. The basic intuition is to force the network to improve its accuracy at each epoch with different configurations, decreasing the risk of overfitting because the final trained model is an averaged outcome of an ensemble of different networks.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.3: Dropout [16] application during the training phase.

2.2 Multi Layer Perceptron

The first and simplest neural network is the Multi Layer Perceptron (MLP) where several different neurons build a more complex architecture. Neural Networks are designed as Directed Acyclic Graph (DAG) to avoid circle that would generate an infinite loop in the forward pass. In MLP the neurons are divided into fully-connected layers such that two adjacent layers are fully pairwise connected with no connection between neurons belonging to the same layer. The MLP is composed by three types of layers: input, hidden and output layers.

The *input* layer is composed by a number of neurons equal to the number of samples composing a single training datum.

The *hidden* layer contains all the intermediate layers which separate the input and the output layers. These represent the engine of networks, collecting most of the model parameters. Originally the networks were shallow because they were composed by a single hidden layer. Nowadays the number of the hidden layers is progressively increasing thanks to hardware improvements and the model's depth has grown, founding the Deep Learning.

The *output* layer structure depends on the type of task: if it is a regression task, the output layer is composed by a single neuron returning a single value, if it is a classification task instead for each output class there is a corresponding output neuron, so the final result will be a vector of probabilities, where each value represents the confidence score computed by the network for the each class. Normally, the output higher in magnitude is taken as the final network prediction.

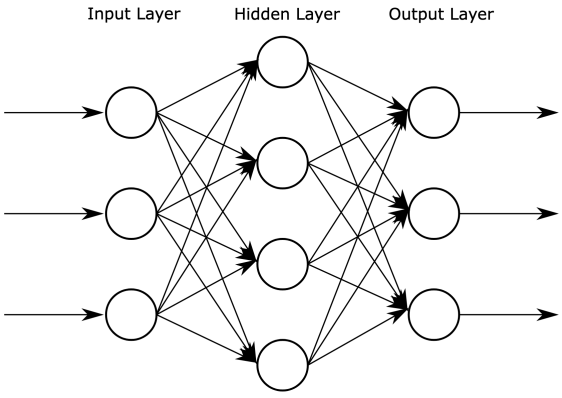


Figure 2.4: Multi Layer Perceptron (MLP).

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [17] are actually the de-facto standard architecture for Computer Vision tasks. MLP networks cannot handle in an efficient manner images as input data. In a MLP network each input neuron is connected with each input data. This requires to associate an input neuron to every pixel in the input image. For instance, considering an RGB image of resolution 200×200, the resulting number of neurons would be $200 \times 200 \times 3 = 120.000$. For this reason, a fully-connected NN is not able to scale, allowing to process only images with very poor resolution. Moreover, in an image each pixel tends to have an high correlation with its neighbors. Associating a single parameter to each one the MLP's inputs completely ignores this correlation.

A Convolutional Neural Network is composed by different type of layers (as shown in Figure 2.5): the *convolutional* layer which performs a convolution operation, the *pooling* layer, the *normalization* layer and the *dense* layer, which is a fully-connected layer typically used as last layer to output the class scores.

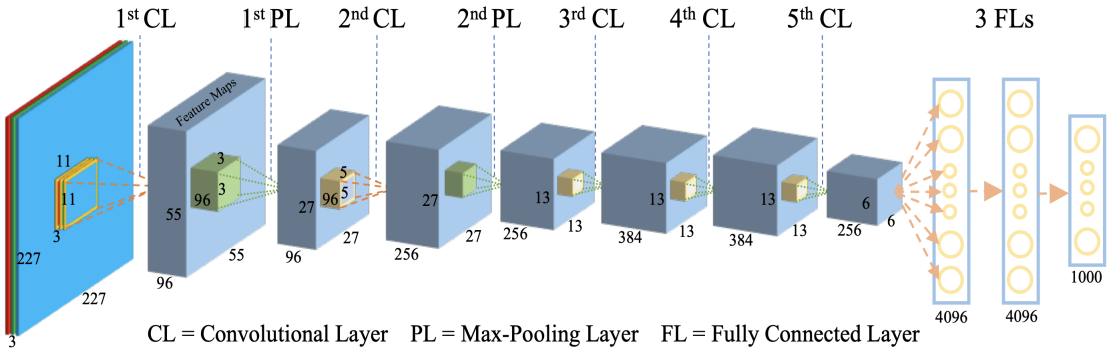


Figure 2.5: Convolutional Neural Network [17].

2.3.1 Convolutional Layer

The convolutional layer is based on a set of trainable filters, the most common in CV tasks is the 2D convolutional layer. These special neurons are defined considering 3 dimensions: width, height and depth. A convolutional filter does not cover completely the entire portion of the image, otherwise the same problem of the fully-connected layer would occur. Conversely, the filter

is applied over a small portion of the input image (for instance a region of 3×3 pixels) and the sum of the dot products is computed between the filter weights and the pixels values over the different channels (e.g., RGB) of the input image (as depicted in Figure 2.6).

Since the entire input image must be processed, the filter (whose size is usually defined as *kernel*) slides over the entire input with a step parameter defined as *stride*. The convolutional layer takes into account the local correlation between pixels close one to each other, furthermore it is always applied to the full depth of the input data, which in case of an image is generally 3 (red, green and blue channels). The *dilation* parameter describes the distance between two adjacent weights inside the kernel, in general the default value is equal to 1.

The main advantage of a convolutional layer is its parameters sharing feature: each neuron in a single depth slice uses the same weights and biases, reducing the total number of parameters needed and the total amount of time needed in the training phase.

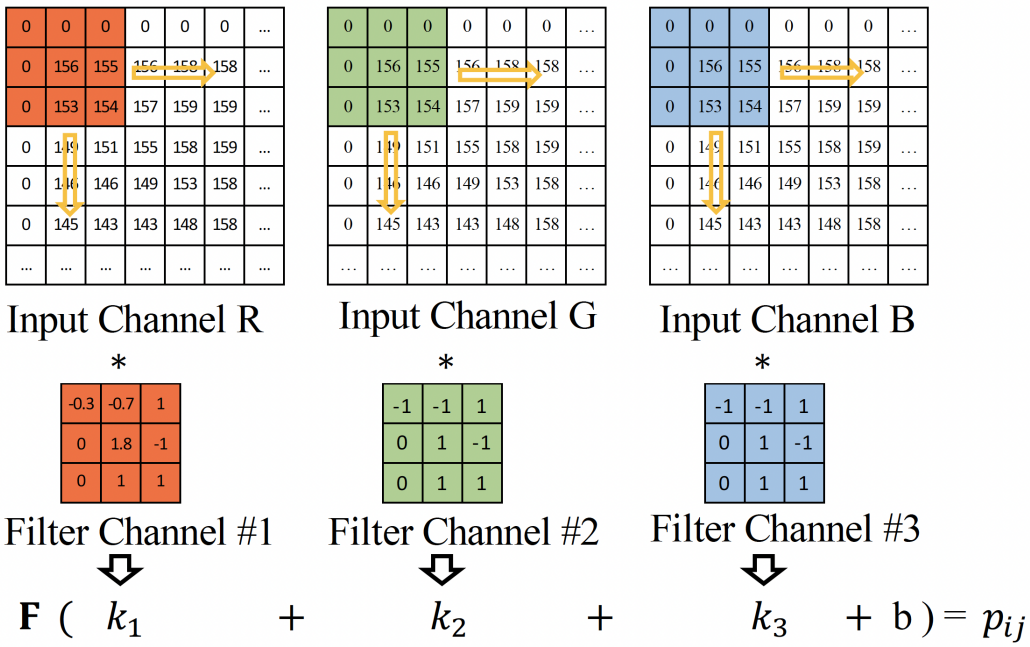


Figure 2.6: Convolutional Layer [17].

2.3.2 Pooling Layer

The pooling layer performs a down-sampling operation over the input feature map reducing the output spatial size, which corresponds to a reduction of the number of operations in subsequent layers. The pooling layer applies a sliding filter independently on every depth slice, as the convolutional layer, and it extracts a single value for window using the max or the average operation. The pooling progressively shrinks the spatial dimension of the feature maps providing a more compact representation of the knowledge extracted. The pooling is also a technique useful in order to reduce the overfitting and increase the model generalization capability.

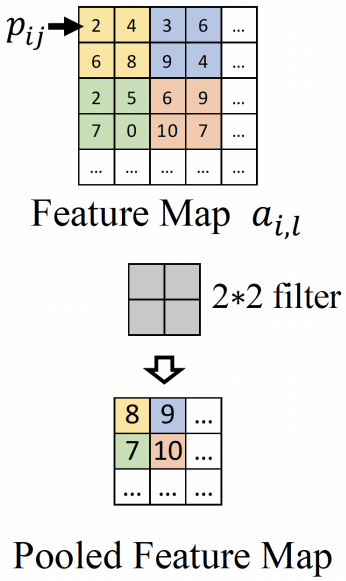


Figure 2.7: Max Pooling Layer [17].

2.3.3 Normalization Layer

A good practice during NN training and inference phase is to normalize the values of the input data in order to reduce phenomenon like exploding gradients. However the normalization is applied not only to input data, but also after the activation function over the intermediate feature maps in order to improve converge (thus reducing training time) and overfitting.

The most famous normalization technique is the *Batch Normalization* (BN) [18]. For a d dimensional layer $x = (x^{(1)}, ..., x^{(d)})$ the BN layer normalizes each dimension:

$$y^{(k)} = \gamma^{(k)} \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}] + \epsilon}} + \beta^{(k)} \tag{2.14}$$

where the expected value $\mathbb{E}[x^{(k)}]$ and the variance $\mathrm{Var}[x^{(k)}]$ are not calculated over the entire training set, but over a single batch. ϵ is a small number added to the denominator to avoid divisions by 0. Then the normalized input data with 0 mean and unit variance are further elaborated with the two trainable parameters γ and β.

2.4 Temporal Convolutional Networks

Similarly to CNNs which became the reference architecture in CV tasks where the input data are images, in the field of one dimensional time series the RNNs emerged as the stardard structure to solve those tasks.

A new kind of network called Temporal Neural Network (TCN) [19] has been designed in order to achieve state-of-the-arts results with significant advantages from the computational aspect.

Time series data represent sequence of data points indexed in time order. Given an input vector at time t the inference y_t can be performed using only the inputs available at that given instant, so $x_0, ..., x_t$.

To satisfy this requirement the TCN is composed by *causal convolutions*, where in order to infer an output at instant t only the previous elements in time are considered as source.

The main problem of causal convolution is the ability at looking back in time with a capacity linearly proportional to the depth of the network. This means that a high number of hidden layer or a large window filter is needed to increase the amount of history taken into account, with the consequent high overhead in terms of parameters and computations.

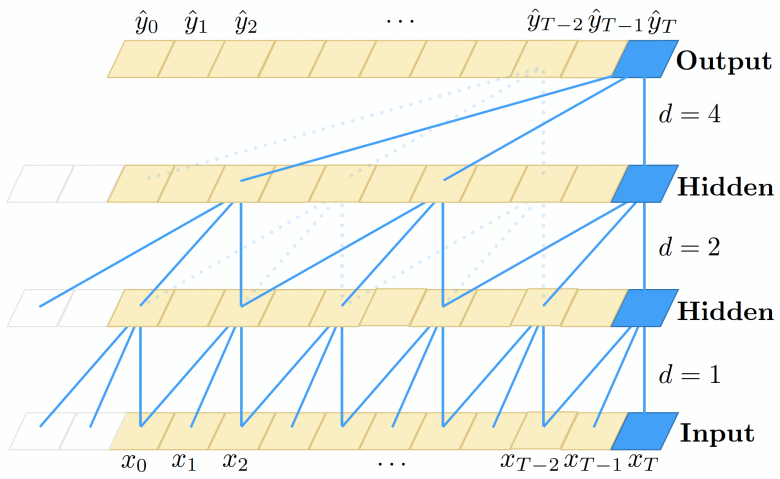


Figure 2.8: Dilated causal convolution with filter size $k = 3$ and dilation factors $d = 1,2,4$ [19].

The second peculiarity of TCNs is the definition of a dilation parameter different than 1 (as shown in Figure 2.8). This is exactly the same hyper-parameter of any convolutional layer, but in this scenario the data

are 1-dimensional instead of 2-dimensional. Increasing the value of the dilation allows to obtain larger receptive field without increasing the number of weights. As best practise the dilation parameter is increased following the powers of 2 based on the depth of the architecture, so for a given layer at level i the corresponding dilation d is set to $d = 2^i$.

TCN provides remarkable advantages: it allows the parallel computation of the convolution differently from RNN increasing the training speed of the model. It gives the possibility to easily tune the receptive field size acting on the number of stacked causal convolutions or on the dilation parameter while maintaining the capability of accepting inputs of variable length like in the RNNs.

Chapter 3

Related works

3.1 Reinforcement Learning NAS

The first NAS approach proposed to tackle the challenge of exploring in an automatic manner many different network configurations was based on Reinforcement Learning [20]. At each iteration of the algorithm, a network is selected among all the explorable ones by a model known as controller. This network is trained on the training set and provides an accuracy result over the validation set. The accuracy is used as reward signal to update the policy gradient of the controller. The controller, in the next passage will sample with higher probability an architecture configuration with improved performance.

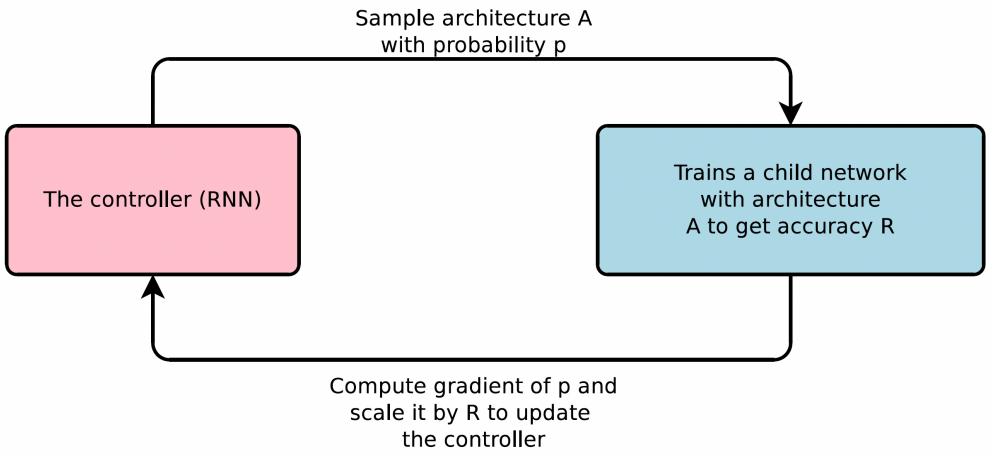


Figure 3.1: Neural Architecture Search with Reinforcement Learning [20].

As depicted in Figure 3.1 the controller generates architectural hyperparameters of a DNN by means of a RNN. The NAS technique proposed in [20] is specifically designed for CNN. At each time step the network predicts the structure of a single layer (number of channels, kernel size etc...) which is the input of the layer that will be generated in the following timestep as shown in Figure 3.2:

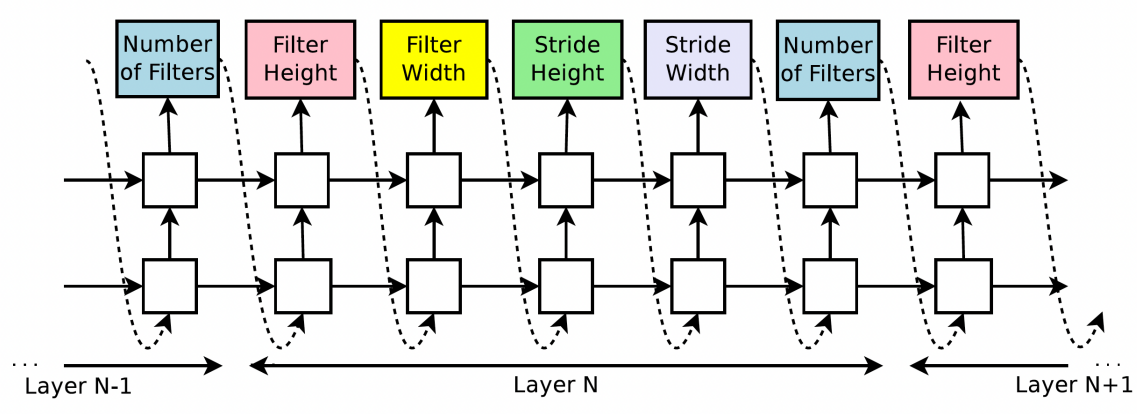


Figure 3.2: Neural Network layers generation process [20].

The list of steps necessary for the controller to define a network architecture can be defined as a list of actions $a_{1:T}$. The performance of the resulting network is measured with the computation of an accuracy score R over the validation set. The controller aims to maximize the expected reward to discover the best architecture:

$$J(\theta_c) = \mathop{\mathbb{E}}_{P(a_{1:T};\theta_c)}[R] \tag{3.1}$$

The REINFORCE rule [21] is used to define a policy gradient method which update iteratively θ_c since the reward signal R is not differentiable:

$$\nabla_{\theta_c} J(\theta_c) = \sum_{t=1}^{T} \mathop{\mathbb{E}}_{P(a_{1:T};\theta_c)}[\nabla_{\theta_c} \log P(a_t|a_{t-1:1};\theta_c)R] \tag{3.2}$$

The previous quantity 3.2 is empirically approximated as:

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T} \nabla_{\theta_c} \log P(a_t|a_{t-1:1};\theta_c)R_k \tag{3.3}$$

In this approximation, m represents the total number of different neural networks sampled by the controller in a single batch, T is the number of

hyperparameters predicted by the controller in the design of the architecture and R_k is the validation accuracy of the k-th neural network obtained after the training.

The quantity in 3.3 is an unbiased estimate with high variance. To reduce the variance, a baseline b is subtracted to the reward accuracy R_K. In this application b is the exponential moving average of the previous architectures accuracy R_K:

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T}\nabla_{\theta_c}\log P(a_t|a_{t-1:1};\theta_c)(R_k-b) \tag{3.4}$$

The usage of a RNN as controller provides flexibility in the search of variable-length architectures.

3.2 SuperNet-based Differentiable NAS

Although NAS algorithms based on RL have been extensively applied in many research activities, rapidly the impracticability of the computational aspect pushed the researchers to envisage more efficient methods. For instance, to discover state-of-the-arts neural networks for a standard dataset such as CIFAR-10 the computational effort reported in [22] required 500 GPUs across 4 days resulting in 2.000 GPU-days.

The main reason of inefficiency about this typology of NAS algorithms is related to the discrete search-space domain. The training needs many trials to find the best network and each of this attempt requires tons of GPUs hours. A different approach is needed to allow scalability over bigger datasets and bigger search-spaces.

A new generation of NAS was born with the creation of DARTS [23] (Differentiable ARchiTecture Search). DARTS' main novelty is the capacity of translate a discrete domain into a continuous one and exploit a gradient-based optimization approach which is much more efficient than the previous one. DARTS achieves state-of-the-art performance with orders of magnitude less computational resources, providing at the same time a rich search-space among which discover the best configuration.

DARTS builds a directed acyclic graph composed by several building block connected by edges. A node $x^{(i)}$ depicts a data representation (e.g. a set of feature maps in a convolutional network) while an edge (i, j) is related to a specific data transformation step $o^{(i,j)}$. Each node is connected by different edges, as shown in Figure 3.3. The idea is to train a model able to find the

path which leads to the best NN and, at the same time, learn the trainable weights of the network.

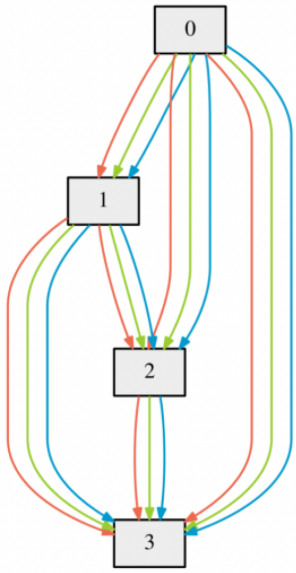


Figure 3.3: DARTS [23] architectural overview.

Given a set of possible operations $\mathcal{O}$, in order to define a continuous search-space, a softmax over all the candidate operations is computed and a parameter vector $\bar{o}^{(i,j)}$ is obtained:

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} exp(\alpha_{o'}^{(i,j)})} o(x) \tag{3.5}$$

Each edge (i, j) is associated to a parameter α (as depicted in the equation 3.5) which is updated during the training phase. At the end a softmax is applied and the path associated to the highest value is chosen.

The equation 3.5 provides a set of continuous variables $\alpha = \{\alpha^{(i,j)}\}$ which can be trained in combination with the set of weights w. In this way it is not necessary to completely separate the two phases such as in RL, defining first a complete fixed structure and train it from scratch, but these two groups of parameters α and w can be learnt in combination.

The final purpose is to discover the set of architectural parameters α^* that minimizes the validation loss $\mathcal{L}_{val}(w^*, \alpha^*)$, where the weights w^* minimize the training loss of that specific architecture:

$$\min_{\alpha} \mathcal{L}_{val}(w^*(\alpha), \alpha) \quad \text{s.t.} \quad w^*(\alpha) = \arg\min_{w} \mathcal{L}_{train}(w, \alpha) \tag{3.6}$$

However the exact calculation of $w^*(\alpha)$ can be not affordable in some cases. Instead of completely training until convergence equation 3.6 is possible to approximate $w^*(\alpha)$ considering the weights w after a single training step.

At the end of the training, a single discrete architecture is extracted replacing the vector of possible edge operations $\bar{o}^{(i,j)}$ with the most probable one $o^{(i,j)} = \arg\max_{o \in \mathcal{O}} \alpha_o^{(i,j)}$.

3.3 Mask-based Differentiable NAS

Neural Architecture Search techniques based on Reinforcement Learning [20] were the forerunners in this field, however they need a huge amount of computational resources and a high number of trial-and-error attempts. The result is an unsustainable amount of time and computational power whenever the size of the datasets grow in volume and complexity.

SuperNet DNAS-based techniques provide a feasible and scalable algorithm based on the choice of the best path between multiple edges representing each one a different operation. The SuperNet drawback is mainly related to the dimension of the search-space: indeed compared to other techniques it is quite narrow and it requires the definition in memory of all the candidate edges.

To overcome this issue a new kind of Differentiable NAS has been proposed called DMaskingNAS or mask-based DNAS. MorphNet [24] and FBNetV2 [14] belong to this category. The main idea is to search over the spatial and channel dimensions which the previous techniques did not consider, this is practically done applying some masks to the layers in order to make their parameters trainable as the weights of the network. In section 3.3.1 and 3.3.2 MorphNet and FBNetV2 DMaskingNAS algorithms are described.

3.3.1 MorphNet

MorphNet [24] aims to provide a simple approach to build automatically neural network architectures taking into account some constraints required to deploy the final model on edge devices.

The proposed approach is scalable to large datasets and models, and allows the optimization of a DNN with the respect to some specific requirements,

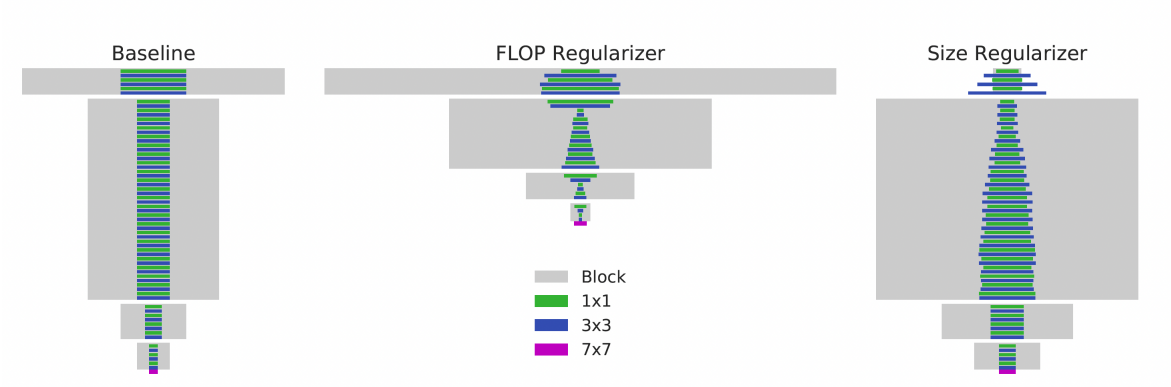


Figure 3.4: Examples of NN architectures shaped by MorphNet [24].

such as the number of parameters or the FLOPs per inference, and furthermore it does not need an auxiliary network to discover a more accurate and efficient architecture.

MorphNet focuses on the optimization of the output widths of all the layers. An initial seed network $O^o_{1:M}$ is defined where M is the number of layers. The network constraints $\mathcal{F}$ are bounded by a threshold ζ fixed initially: $\mathcal{F}(O_{1:M}) \leq \zeta$. In general the optimization problem solved by MorphNet has the following form:

$$O^*_{1:M} = \underset{\mathcal{F}(O_{1:M}) \leq \zeta}{\arg\min} \min_{\theta} \mathcal{L}(\theta) \tag{3.7}$$

where θ represents the ensemble of the parameters of the network and $\mathcal{L}$ is the loss function. $\mathcal{L}$ combines the loss computed based on the task of the DNN and the loss calculated considering the network constraints.

Algorithm 1 The MorphNet Algorithm

1: Train the network to find $\theta^* = \arg\min_\theta \{\mathcal{L}(\theta) + \lambda \mathcal{G}(\theta)\}$.
2: Find the new widths $O^{'}_{1:M}$ based on θ^*.
3: Find the largest ω s.t. $\mathcal{F}(\omega \cdot O^{'}_{1:M}) \leq \zeta$
4: Set $O^o_{1:M} = \omega \cdot O^{'}_{1:M}$ and repeat from step 1 as many times desired.
5: **return** $\omega \cdot O^{'}_{1:M}$

The network structure is shaped by means of the width multiplier ω, already used in MobileNet [35]. The purpose is to find the highest ω such that $\mathcal{F}(O_{1:M}) \leq \zeta$. However the application of this naive approach leads to a significant loss in terms of accuracy which must be compensated.

A more articulated approach to increase the sparsity in the network consists in the definition of an additional regularization term $\mathcal{G}(\theta)$ which assigns a higher cost to the neurons which concur more the constraint $\mathcal{F}(O_{1:M})$. This additional term allows to train the weights of the network finding the best trade-off between the accuracy provided by $\mathcal{L}$ and the constraints given by $\mathcal{F}$. However the new sets of weights $\theta^* = \arg\min_\theta\{\mathcal{L}(\theta) + \lambda\mathcal{G}(\theta)\}$ does not guarantee the respect of $\mathcal{F}(O_{1:M}) \leq \zeta$.

MorphNet algorithm detects the best architecture with an iterative process where the size of the network decreases in the first two steps and increases in the third step.

The novelty of this new methodology is the capacity to shape the network structure during the training phase reducing the computational requirements and determine some boundary constraints to be respected in order to deploy over small memory devices.

3.3.2 FBNetV2

FBNetV2 [14] aims to improve SuperNet-based DNAS algorithms proposing new solutions for its weak spots. The first drawback to be adressed is related to the amount of memory required to train the model: since all the possible candidate layers must reside in the GPU for the training phase, there is a physical constraint which limits the dimension of the architectural search-space. Furthermore, each additional candidate layer increases the computational cost linearly, preventing the model to scale up to bigger datasets. Instead of creating a complex sets of candidates layers, the intuition at the basis of FBNetV2 is to start from a seed network and add some masking and shape propagation mechanisms in order to tune the number of parameters of each layer during the training of the model.

To augment the architecture search-space the first novelty in FBNetV2 is the *Channel Search*. To learn the optimal number of channels there are some aspects to consider: the shape of the channels must be compatible between adjacent layers in order to obtain a working model (Figure 3.5, Step A). The most straightforward solution would be to set the number of channels of all the layers equal to the highest value, however this is costly and inefficient (Figure 3.5, Step B).

There is a much more efficient method to make the layers compatible, which is multiplying the layers with a column vector composed by i leading 1 and $k-i$ trailing zeros (Figure 3.5, Step C). Since the layers have the same number of filters is it possible to make an approximation and use the same

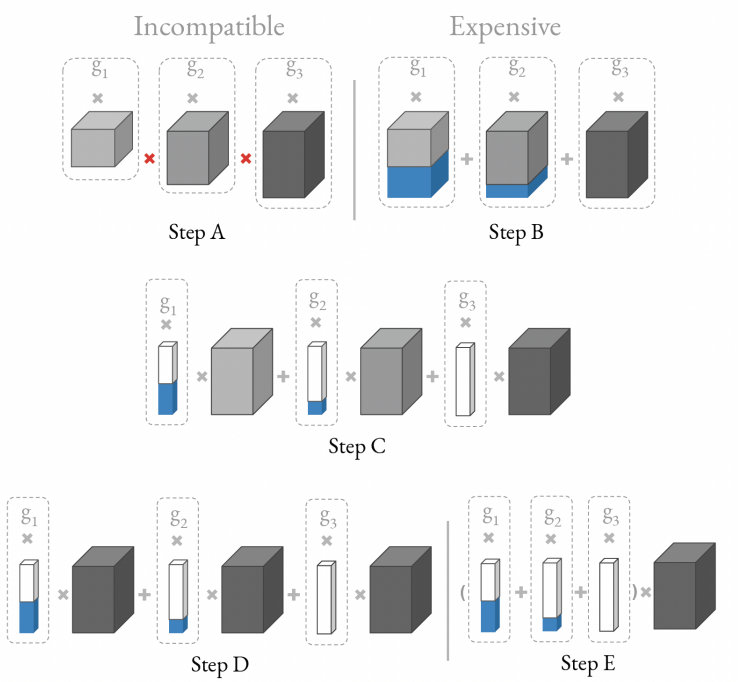


Figure 3.5: FBNetV2 [14] channels search overview.

weights over all these layers (Figure 3.5, Step D) and define the expression:

$$y = \sum_{i=1}^{k} g_i(b(x) \circ \mathbb{1}_i) \tag{3.8}$$

In equation 3.8 $\mathbb{1}_i$ represents the column vector with i leading 1, b a block composed by i filters and g_i the Gumbel softmax weights. This can be further simplified (Figure 3.5, Step E) summing all the masks and compute once the multiplication with the layers to prune the channels corresponding to 0 in the column vector. The requirements for this last step are only one forward step and one feature map, so it is very lightweight and effective at the same time.

The second novelty of this technique is the *Input Resolution Search*. Different features maps after the masking mechanism could be incompatible for their pixel resolutions (Figure 3.6, Step A).

To solve this issue the size of the feature maps is augmented with a zero-pad interspersed spatially (Figure 3.6, Step C). In fact the application of a simple zero padding around the layers (Figure 3.6, Step B) would not solve the incompatibility problem. Another question to be managed is the misalignment between receptive fields. Applying the same receptive field

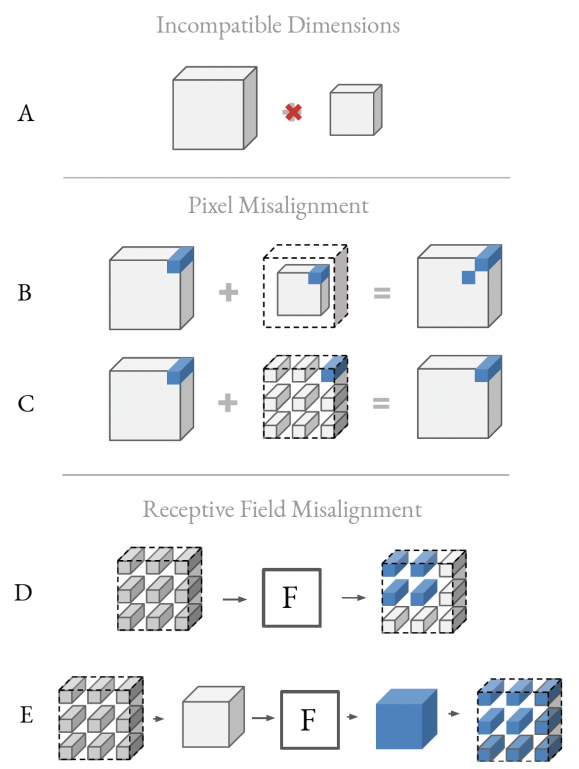


Figure 3.6: FBNetV2 [14] input resolution search overview.

to the new padded feature map would reduce the resulting receptive field (Figure 3.6, Step D), for this reason the convolution is calculated over an input subsample (Figure 3.6, Step E).

Chapter 4

Neural Architecture Search (NAS)

4.1 Pruning In Time (PIT)

This thesis starts from the work presented in the Pruning In Time [15]. PIT belongs to the family of DMaskingNAS, and similarly to MorphNet starts from a seed network and progressively balances the network complexity with accuracy with a structured pruning of weights.

PIT introduces two novelties in its approach: first of all it is the first work which focuses specifically in the optimization of Temporal Convolutional Networks which represent a good alternative to RNNs for time-series processing tasks as explained in Sec. 2.4. The vast majority of NAS literature focuses on the optimization of 2D-CNN for Computer Vision tasks, although there are many use-cases related to edge-computing whose input source is a uni-dimensional time dependent signal (keyword spotting, anomaly detection, etc.). Secondly, PIT is the only mask-based DNAS algorithm able to optimize the dilation factor.

PIT target layers originally were fully-connected layers and 1D convolutional layers. Nevertheless, the original algorithm has been extended in order to apply a PIT-like NAS also for also for CV tasks such as Image Classification and Visual Wake Words.

PIT, as other DMaskingNAS algorithms, is not capable to choose between different kind of layers, instead it defines a seed network which is progressively pruned. However, the advantages of this choice are multiple: first, it achieves a significant reduction of the search-time and of the computational resources

needed; second, it explores an huge and fine-grained search-space.

Each fully-connected and convolutional layer are transformed into a so-called *PIT layer*. Each layer of this type can be represented as a function $L_n(W^{(n)};\theta^{(n)})$, where $W^{(n)}$ refers to the original weights tensor of the layer, while $\theta^{(n)}$ denotes a new set of architectural parameters. In general, given a network with N layers, the PIT search-space is defined as:

$$\mathcal{S} = \{L_n(W^{(n)};\theta^{(n)})\}_{n=0}^{N-1} \tag{4.1}$$

For each convolutional layer PIT explores different hyper-parameters settings. In particular, the NAS optimizes the number of output channels (C_{out}) controlled by the parameters $\alpha^{(n)}$, the receptive field (F) controlled by the parameters $\beta^{(n)}$, and the dilation factor (d) controlled by the parameters $\gamma^{(n)}$. The binary masks used to remove specific portion of the weights tensor are derived from these three sets of parameters and are treated independently providing the possibility to optimize the parameters jointly or in a separate way. The joint search-space has the following size:

$$|S| \approx \Pi_{n=0}^{N-1}(C_{out,seed}^{(n)} \cdot F_{seed}^{(n)} \cdot \lceil \log_2(F_{seed}^{(n)}) \rceil) \tag{4.2}$$

The logarithmic term associated to the dilation factor in Eq. 4.2 is due to the choice of consider only power-of-2 dilation factors.

4.1.1 Channels Search

To optimize the number of output channels, PIT treats each channel independently by means of a specific parameter α. Each fully-connected or convolutional layer has associated an array of parameters α of the same length of the original number of output neurons or output channels. For the sake of simplicity from now on the term output channels will be used to identify also the output neurons.

The pruning of the output channels is based on a binary mask Θ_A which is obtained applying a Heaviside binarization function $\mathcal{H}$ to the parameters α with a fixed threshold $th = 0.5$:

$$\mathcal{H}(x, th) = \begin{cases} 1, \text{ if } x > th \\ 0, \text{ otherwise} \end{cases} \tag{4.3}$$

$$\Theta_A = \mathcal{H}(|\alpha|) \tag{4.4}$$

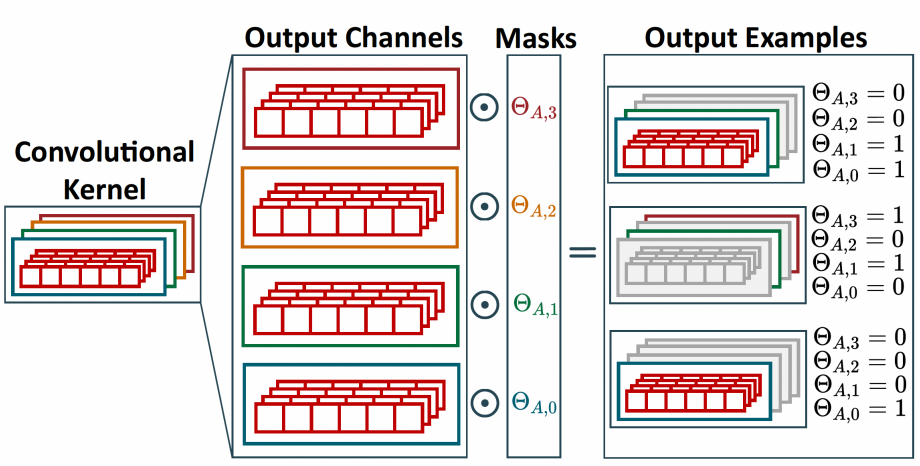


Figure 4.1: Example of output channels search [15]. Each $\Theta_{A,m} = 0$ eliminates the corresponding channel.

The binary masks $\Theta_{(n)}$ eliminate part of the network computing the Hadamard product between the weights $W^{(n)}$ and the binary masks themselves. When an output channel is multiplied by a 0-mask is pruned, as shown in Fig. 4.2. The pruning steps update the PIT model as follows:

$$\hat{\mathcal{S}} = \{L_n(W^{(n)} \odot \Theta^{(n)})\}_{n=0}^{N-1} \tag{4.5}$$

However to perform a standard gradient training to learn both the weights $W^{(n)}$ and the architectural parameters $\theta^{(n)}$ the binary values generated by the Heaviside function should not be used in the backward pass. To allow a correct gradient flow the Straight-Through Estimator (STE) [25] technique have been applied: the binary values are computed applying the Heaviside function only in the forward pass, while in the backward pass the step function is replaced by the identity function making the gradient flow possible.

In principle, during the training phase an entire layer could be pruned by the PIT algorithm. To avoid this situation at least one output channel for each layer is preserved.

4.1.2 Receptive Field Search

The second hyper-parameter explored by PIT is the receptive field F. In a classical convolution the receptive field coincides with the kernel size, however when the dilation factor d is > 1 then the general relation is given by $F =$

$(K-1)\cdot d+1$. The optimization of the receptive field F and of the dilation factor d indirectly leads to optimize also the filter dimension K.

Similarly to the output channels search, an array of trainable parameters $\beta^{(n)}$ of length F_{seed} is associated to each layer to optimize.

The pruning step in this scenario should take into account also the causality characteristic of the TCN. The receptive field cannot be shrunk arbitrarily, the pruning must start from the oldest slice of F and follow the time order.

The binary mask Θ_B is obtained by combining the parameters β with the following form:

$$\Theta_{B,i} = \mathcal{H}(\sum_{j=1}^{F_{seed}-i} |\beta_{F_{seed}-j}|) \tag{4.6}$$

During the forward pass a time-slice of the tensor W is multiplied with $\Theta_{B,i}$. With this approach the first weight to be pruned is always the oldest in time since with $i > j$ then $\Theta_{B,i} \leq \Theta_{B,j}$. To ensure at least one time step as input the parameter β_0 is always fixed to 1.

Practically, the binary mask Θ_B is obtained by means of the product between an upper triangular matrix of 1s generated at the beginning and the parameters β. The product is then binarized with the Heaviside function $\mathcal{H}$:

$$\Theta_B = \mathcal{H}(C_\beta \cdot |\beta|) \tag{4.7}$$

4.1.3 Dilation Search

The optimization of the dilation hyperparameter is an innovation introduced by the PIT algorithm. As previously mentioned, only power-of-2 dilation factor are taken into consideration, this is because the majority of the inference libraries apply this approach in order to generate regular access pattern to the memory, which allows the deployment of more efficient models on edge devices.

Analogously to the two previous search, also for the dilation optimization an array of trainable parameters γ is defined with length $len(\gamma) = \lceil \log_2(F_{seed}) \rceil$.

Before obtaining a binary mask Θ_Γ it is necessary the computation of an intermediate array Γ defined as:

$$\Gamma_i = \mathcal{H}(\sum_{j=1}^{len(\gamma)-i} |\gamma_{len(\gamma)-j}|) \tag{4.8}$$

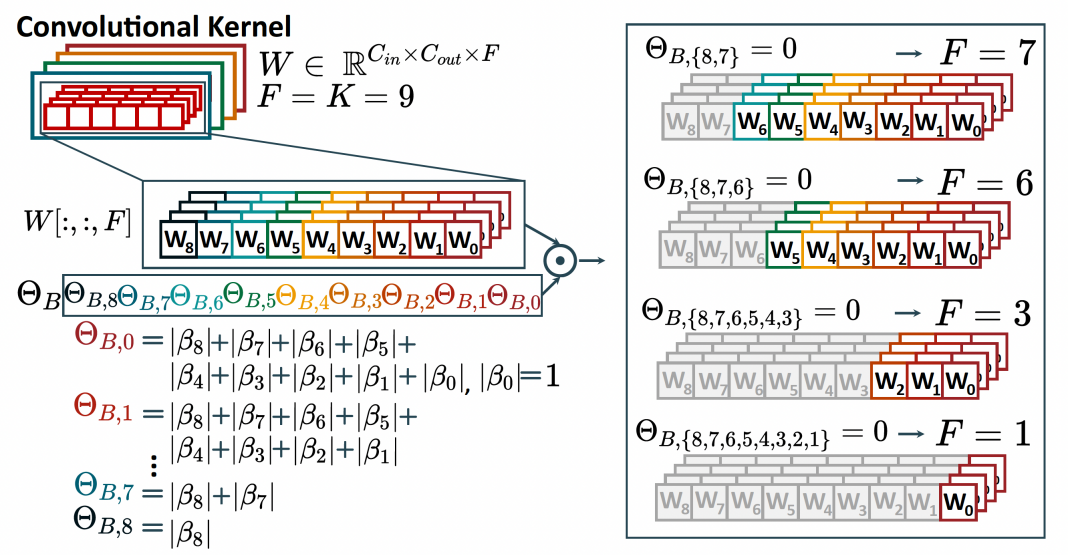


Figure 4.2: Example of receptive field search [15]. Each $\Theta_{B,i} = 0$ eliminates the contribution of 1 time step from the convolution output.

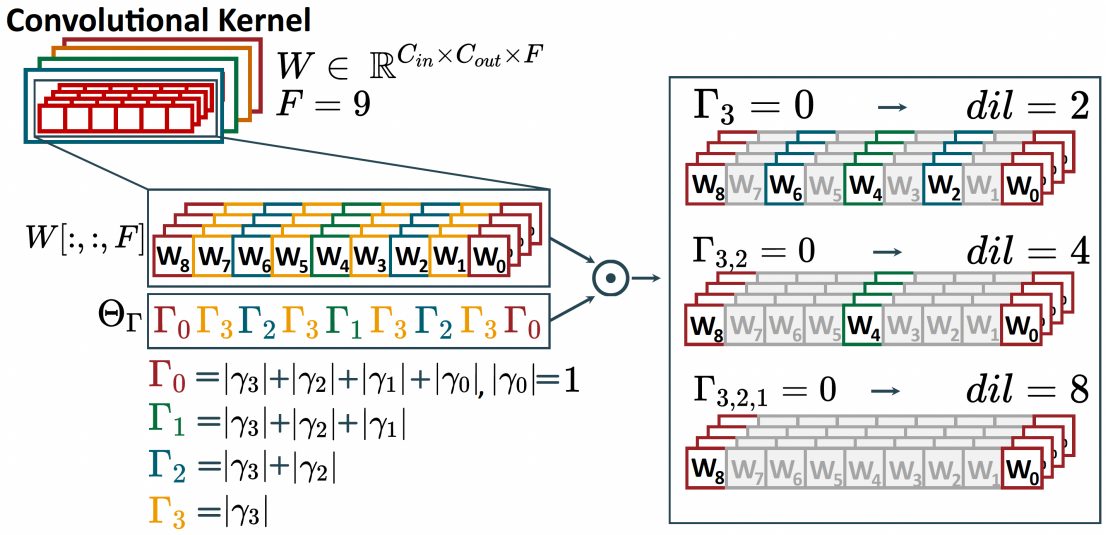


Figure 4.3: Example of dilation search [15]. Each $\Gamma_i = 0$ increases the dilation by a factor of 2.

Then, the Kronecker's Delta function δ is used to define the final binary masks $\Theta_{\Gamma,i}$:

$$\Theta_{\Gamma,i} = \Theta_{k(i)} \tag{4.9}$$

where

$$k(i) = \sum_{p=1}^{len(\gamma)} 1 - \delta(i \mod 2^p, 0) \tag{4.10}$$

and with

$$\delta_{ij} = \begin{cases} 0 \text{ if } i \neq j \\ 1 \text{ if } i = j \end{cases} \tag{4.11}$$

Practically, each time a Γ_i is equal to 0 the dilation factor increases of a factor of 2. To prevent a resulting dilation hyperparamter equal to 0, similarly to the channel and receptive field search, the parameter γ_0 is a fixed constant equal to 1.

The binary mask Θ_Γ is computed with a simple matrix multiplication between a constant matrix C_γ of 0s and 1s, which can be retrieved by the value of F_{seed}, and the array of parameters γ:

$$\Theta_\gamma = \mathcal{H}(C_\gamma \cdot |\gamma|) \tag{4.12}$$

4.1.4 Regularization

PIT training aims to find the best possible trade-off between the highest accuracy and the simplest architecture to minimize memory or number of operations. These two aspects are implemented in the NAS loss function which sums the loss $\mathcal{L}_{task}$ related to the task of the network with a regularization term $\mathcal{R}$ which encodes the hardware costs:

$$\mathcal{L} = \mathcal{L}_{task}(W; \theta) + \lambda \mathcal{R}(\theta) \tag{4.13}$$

The regularization term $\mathcal{R}$ can be of two types: $\mathcal{R}_{size}$ computes the number of effective parameters of the pruned network, $\mathcal{R}_{ops}$ calculates the number of operations required for an inference step.

The number of effective channels $\mathcal{C}^{(n)}_{out,eff}$ and the effective kernel size $\mathcal{K}^{(n)}_{eff}$ given a n-th layer are defined as:

$$\mathcal{C}^{(n)}_{out,eff} = \sum_{i=0}^{\mathcal{C}^{(n)}_{out,seed}-1} \tilde{\Theta}^{(n)}_{A,i} \tag{4.14}$$

$$\mathcal{K}^{(n)}_{eff} = \sum_{i=0}^{F^{(n)}_{seed}-1} \frac{\tilde{\Theta}^{(n)}_{B,i}}{F_{seed} - i} \cdot \frac{\tilde{\Theta}^{(n)}_{\Gamma,i}}{len(\gamma) - k(i)} \tag{4.15}$$

It is worth pointing out in the computation of the effective kernel size $\mathcal{K}_{eff}^{(n)}$ the normalization of the binary masks $\tilde{\Theta}_{\Gamma,B}$ in order to not obtain a cost greater than the real filter size.

The total number of effective parameters $\mathcal{R}_{size}$ left in the resulting architecture is computed by:

$$\mathcal{R}_{size} = \sum_{n=0}^{N-1} (\mathcal{R}_{size}^{(n)}) = \sum_{n=0}^{N-1} \mathcal{C}_{out.eff}^{(n-1)} \cdot \mathcal{C}_{out.eff}^{(n)} \cdot \mathcal{K}_{eff}^{(n)} \tag{4.16}$$

The total number of operations $\mathcal{R}_{ops}$ is given by weighting the number of parameters of each n-th layer layer with the output sequence length T:

$$\mathcal{R}_{ops} = \sum_{n=1}^{N} (\mathcal{R}_{size}^{(n)} \cdot T^{(n)}) \tag{4.17}$$

The λ parameter (equation 4.13) defines the strength of the regularization term, the higher is the more PIT will try to create a more constrained network preferring hw-related metrics to the inference accuracy. On the other side the lower is λ the lower will be the pruning of the network and the lower will be the loss in terms of accuracy.

4.1.5 Training procedure

The training is divided in three phases: the *warmup loop*, the *search loop* and the *fine-tuning loop*.

Algorithm 2 PIT Algorithm

1: **for** i $\leftarrow 1, ...,$Steps$_{\text{wu}}$ **do** *warmup* loop
2: Update W based on $\nabla_W \mathcal{L}_{task}(W)$
3: **end for**
4: **while** not converged **do** *search loop*
5: Update W and θ based on $\nabla_{W,\theta}(\mathcal{L}_{task}(W;\theta) + \lambda\mathcal{R}(\theta))$
6: **end while**
7: **for** i $\leftarrow 1, ...,$Steps$_{\text{ft}}$ **do** *fine-tuning* loop
8: Update W based on $\nabla_W \mathcal{L}_{task}(W)$
9: **end for**

In the warmup phase the architectural parameters θ (i.e: α, β, γ) are initialized to 1 and frozen, so the binary masks Θ will be composed by all

1s. The warmup loop coincides with a classical training of the seed network based on minimizing only the task loss function $\mathcal{L}_{task}$.

In the second phase, the search loop, the PIT algorithm enters in action to learn the best architectural parameters θ and the best network weights W to minimize the global loss function obtained by the sum between the task loss and the regularization term described in Eq. 4.13. The search goes on until the value of the task loss function $\mathcal{L}_{task}$ on the validation set does not decrease for 20 epochs.

In the last phase, the architectural parameters θ are frozen and a new architecture is built. Then the network weights W are retrained from scratch or fine-tuned considering only the task loss function $\mathcal{L}_{task}$.

The described algorithm can be repeated with many different values of λ in order to obtain a collection of Pareto points in the accuracy vs cost space.

4.2 Library organization

The PIT algorithm previously described has been refactored and included in the *flexnas* library. The purpose of this section is to provide an overall overview on how the code is organized.

The development of a collection of unit tests has been fundamental to discover the software bugs. The main aspects covered by the unit tests are:

- The conversion of the fully-connected and convolutional layers into *PIT layers*. In particular some problems in the output channels calculation occurred for the depthwise separable convolutional layers.

- The definition of the masks associated to number of output channels, receptive field size and dilation. Based on the architecture some layers should share the same mask (i.e., two convolutional layers after an add operation will share the same mask). To cover the highest number of possibilities different small NNs have been designed in order to check the expected behavior. Moreover, another functionality which has been covered by the unit tests is the possibility to enable the NAS targeting a specific architectural aspect (i.e., searching the best number of output channels without modifying the receptive fields size and dilation).

- The update of the model weights and masks values by the loss function. In particular many tests have been conducted to analyze the impact of the regularization term $\mathcal{R}$ during the training.

- The conversion of the trained PIT model into a standard Pytorch model with the new architectural layout.

4.2.1 Flexnas

The main code of flexnas is organized as follows:

- `flexnas` : main library code
  - `methods` : NAS code
    - `dnas_base` : common code for all DNASes
    - `pit` : PIT-specific code

Figure 4.4: The main two subdirectories of Flexnas library.

- **dnas_base**: an abstract class containing the common code for generic Differentiable NAS algorithms. It takes as attributes the model to be optimized, the name of the supported cost regularization functions and the name and type of layers that should be excluded by the neural architecture search.
- **pit**: the specific Pruning In Time implementation. The `PIT` class receives the Pytorch model to optimize and then it searches for the NAS-able target layers, which are the fully-connected and the convolutional layers. Each of these layer is transformed into a `PITLayer`, an abstract class which provides the interface implemented by all the possible PIT layers. Then each specific layer requires a specific implementation, so the following classes are implemented: `PITLinear`, `PITConv1d`, `PITConv2d`, `PITBatchNorm1d`, `PITBatchNorm2d`. The conversion is handled by the `pit_converter` script and each set of architectural hyperparameters α, β, γ is managed by a specific class. The number of effective output channels is trained with `PITFeaturesMasker` class, the size of the receptive field by `PITTimestepMasker` class and the dilation search by `PITDilationMasker`.

Beside the PIT implementation in FlexNAS, a complete set of unit tests has been developed to increase the reliability, testability and to quickly spot harmful bugs. Since this algorithm could be applied to any kind of neural networks, in the *models* folder a collection of neural networks has been

- `flexnas` : main library code
  - `methods` : NAS code
  - `unit_test` : unit tests
    - `models` : some DNNs used to test the library
    - `test_methods` : unit tests for DNAS methods
    - `test_utils` : unit tests for utility functions

Figure 4.5: Unit test folder of Flexnas library.

defined. The purpose is to cover as much as possible all the different combination between layers and connections of layers in order to check the proper behavior of the NAS algorithm. These networks, which have been defined as *Toy models*, are quite small and their goal is to study the PIT response with many different layer structures. The toy models are then recalled into the full test panel defined inside the `test_methods` directory. The unit tests are divided in different typologies to clearly identified the tests which are targeting the masking aspect from the ones applied to the conversion phase. The `test_utils` folder contains tests related to complementary aspects such as the conversion of the model into a `networkx` directed graph.

One of the most important aspect that has been pursued during the development is the definition of an easy to use library to facilitate the integration of new algorithms and to improve the user-friendliness.

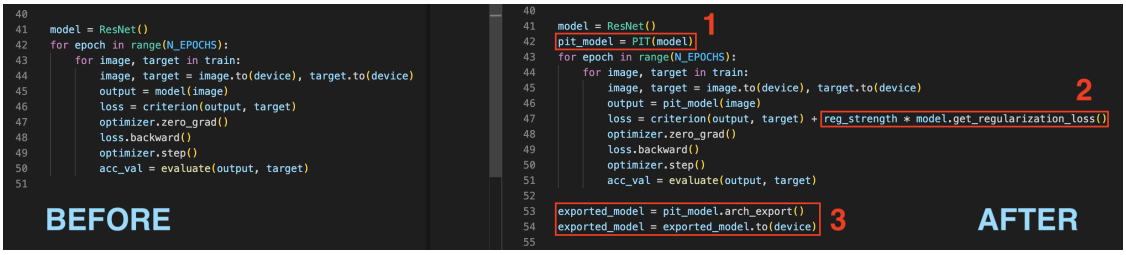


Figure 4.6: Normal training loop (left) vs PIT training loop (right)

Indeed, Figure 4.6 shows how easy is to apply the PIT technique with respect to a standard training loop concerning any kind of model. With FlexNAS, a laborious refactor of the code is absolutely not needed, Figure 4.6 higlights with squared red boxes the 3 small differences: first, to transform the Pytorch model into a NAS-able model is sufficient to call the PIT class and pass as argument the model to be translated. Then the new PIT model can be trained regularly as any Pytorch model computing the loss

function and backpropagating the gradient. In order to train not only the model parameters such as weights and biases, but also the architectural parameters a simple regularization term needs to be added. As reported in the second red square box in Figure 4.6, when the overall loss is calculated the contribution of the regularization term is considered and added to the task loss by simply calling the `get_regularization_loss()` function. To return back to a standard Pytorch model and extract the new discovered network the `arch_export()` function is applied over the PIT model.

Chapter 5

MLPerf Tiny Benchmarks

The technology progression of low-power Machine Learning systems and the data privacy concerns are pushing to set higher quality stardards for DL applications deployed on the edge.

Considering IoT scenarios the focus of the decentralization is centered exclusively around the inference phase. In fact, the main goal in this case is preserving the best possible performance at the minimum cost in terms of hardware resources.

In order to evaluate the impact of optimization techniques aimed at running ML models on IoT devices, the availability of standard benchmarks is fundamental; it permits to assess and compare the effectiveness of different optimizations on a common set of relevant use-cases, and in identical conditions. MLPerf Tiny [26] is the outcome of a collaboration among more than 50 industrial and academic actors to define the first industry-standard benchmark suite for ultra-low-power tiny ML systems. Since the network optimization must take into account the accuracy, the latency and the energy a direct comparison of different solutions is not straightforward. MLPerf Tiny is an open-source inference benchmark suite which provides four standards benchmarks: *Anomaly Detection*, *Image Classification*, *Visual Wake Words* and *Keyword Spotting*. The challenges faced by MLPerf Tiny are several:

- **Low Power**: the power consumption of a ML model is one of the most critical aspect to consider to deploy it on an edge device.
- **Hardware Heterogeneity**: MLPerf Tiny tackles the problem of hardware heterogeneity between different Micro Controller Units (MCUs).
- **Software Heterogeneity**: MLPerf Tiny offers different model deployment options based on TensorFlow framework.

- **Limited Memory**: Low-power MCUs have a really reduced memory size, this represents a pain point for the deployment of the NN in the devices.

- **Cross Product**: the huge differentiation at each level of the technological stack, as shown in Fig. 5.1, constitutes a big issue to perform correctly comparison between different tools and approaches.

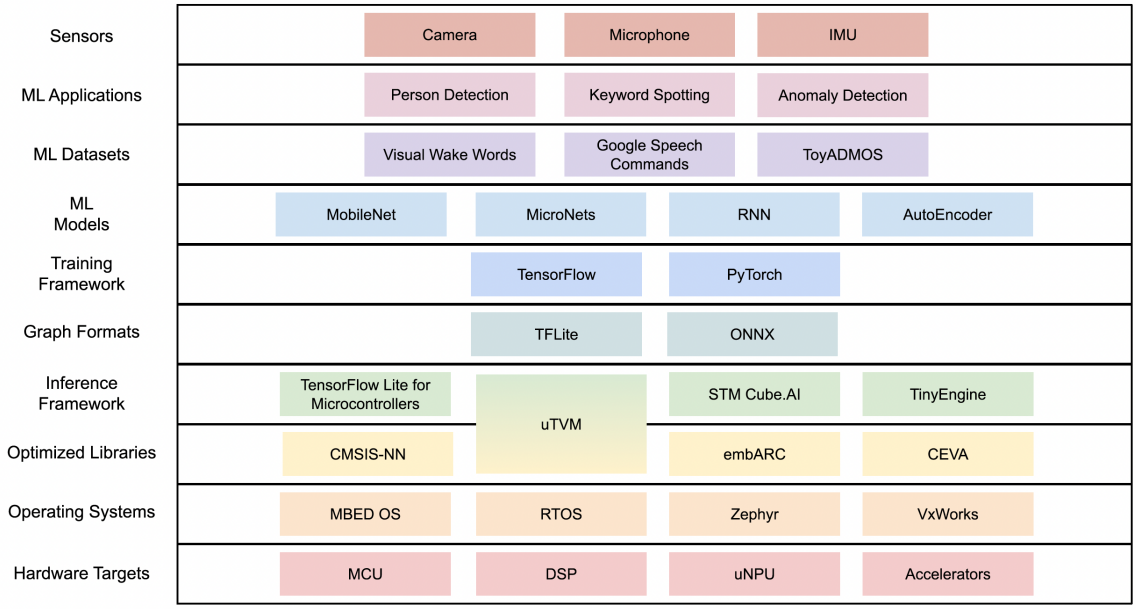


Figure 5.1: The MLPerf Tiny Machine Learning Stack summary [26] displays how challenging is a standardization.

A modular approach has been applied in the development of the MLPerf Tiny. Each benchmark presents a reference implementation which contains everything from the training scripts to the reference hardware framework. The reference submission provides baseline results which allows researchers to demonstrate the quality of their novel algorithms. Furthermore, due to the collaborative nature of the suite, everyone can submit its own implementation to improve the baseline performance. Next, we describe each of the four benchmarks that compose the MLPerf Tiny suite in detail.

5.1 Anomaly Detection

Anomaly Detection [27] refers to the process of recognition and separation of data points that significantly deviate from the vast majority of the instances.

One of the most famous applications of outlier detection are fraud detection algorithms, but the range of possibilities is very wide: healthcare, financial surveillance, risk management and industrial manufacturing are only few examples of possible use-cases for Anomaly Detection.

Indeed, in the industrial sector the predictive maintenance is a topic in which many big companies are investing. The detection of early machine anomalies provides many benefits in terms of cost reduction, increase of production and safety.

The Anomaly Detection process has to face many aspects which make very difficult the outliers discovery: first of all, anomalies are associated to unknown and abrupt behaviors and distributions. Due to their nature, anomalies may present completely different characteristics from one to another, i.e., are heterogeneous. The rarity and the sparsity of anomalies creates a high imbalance in the classes of the dataset, which in turn makes harder a correct learning process by ML and DL algorithms. Consequently often outlier detection processes suffer of low detection rate and many false positives.

Due the high complexity and unpredictability a fully supervised learning method is often unfeasible, since new types of anomalies not present in the training dataset can always occur. The complexity of labeling and designing representative datasets moved most of the research efforts in the direction of unsupervised learning methods, where the labels relative to anomalous samples are not present during training.

The training dataset in the MLPerf Tiny benchmark is the ToyADMOS [28] (Anomaly Detection in Machine Operating Sounds) in which machine-operating sounds and environmental noise are individually recorded for simulating various noise levels. The samples are related to three different tasks: product inspection, fault diagnosis for a fixed machine and fault diagnosis for a moving machine. In each task multiple machines of the same class are used to obtain a more representative set. The released dataset contains more than 180 hours of normal machine-operating sounds and over 4000 samples of anomalous sounds collected with four microphones at a 48-kHz sampling rate for each task.

The reference architecture is an AutoEncoder (AE) which aims to learn a low-dimensional feature representation space where the given data samples can be well reconstructed. The main idea is to force the model to learn how

to extracts features which contain most of the important regularities of the data. Since the outliers are very different from the main distribution of the input instances the reconstruction error for an anomaly is expected to be much bigger than the one for a normal sample.

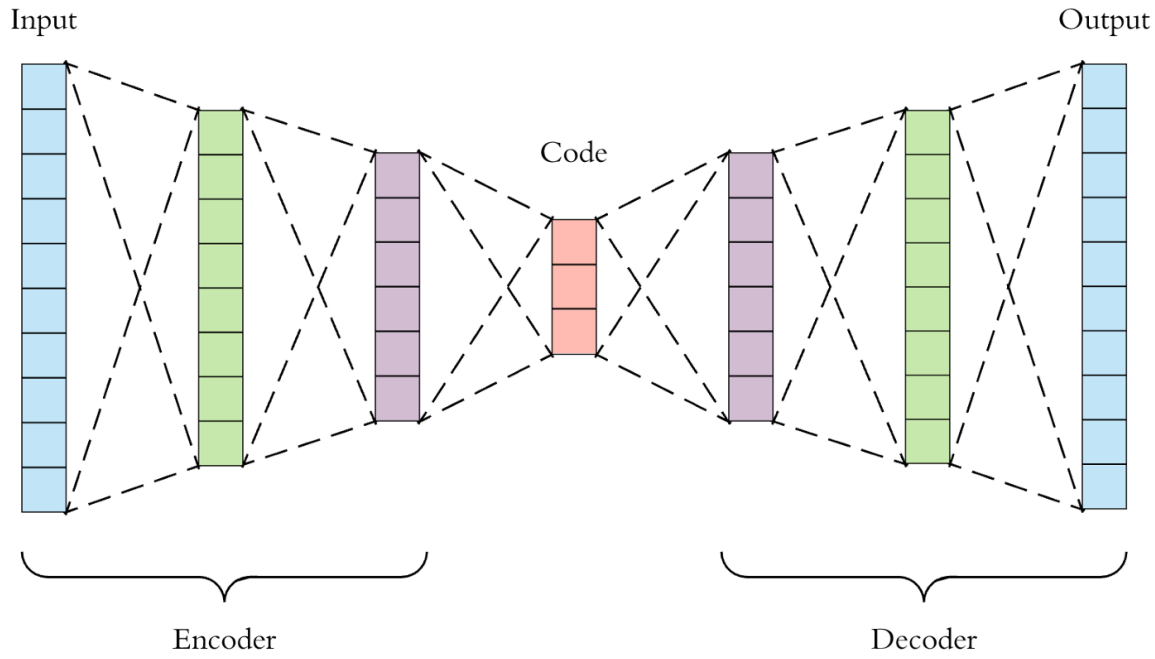


Figure 5.2: General deep autoencoder architecture.

The AutoEncoder is the reference model for the majority of the literature on anomaly detection tasks. The AE used in this benchmark is constituted by only fully-connected (FC) layers, convolutional layers have not been applied. The encoder and the decoder have the same structure composed by 4 FC layers with 128 units each, all followed by BatchNorm layer and ReLU activation. The bottleneck has 8 units.

The model is not directly applied on the audio samples, but rather to some features extracted with a preliminary preprocessing step. In particular, the log-spectrogram with 128 bands and 32 ms of frame size is extracted from each recording, then the model is applied over a sliding windows of 5 frames.

The anomaly score is computed considering the Mean Square Error of the reconstruction error averaged over the central 6.4 seconds part of the spectrogram. To obtain a binary classification outcome it is necessary to set a threshold that allows to separate normal from anomaly scores. To obtain an output quality metric that does not depend on the specific threshold value selected, the AUC-ROC (Area Under The Curve, Receiver Operating

Characteristics) has been used. The proposed AutoEncoder model reaches a AUC-ROC value of 85.5%.

5.2 Image Classification

The boom of Deep Learning models in Computer Vision tasks started with the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC-2012). The dataset [29] of the competition is composed by more than 1 million of images and 1000 classes. In that occasion for the first time a convolutional neural network has been used to solve that classification challenge [30]. The network AlexNet achieved a top-5 error of 15.3%, more than 10.8 percentage points lower than the one of the runner up.

This event showed to the entire world the potentiality of Deep Learning whose training had been made feasible parallelizing the computation with GPUs. Computer Vision is one of the field of major success and application of DL because all the preceding ML techniques were not able to reach that level of performance.

While the ImageNet dataset is still the main reference benchmark for Computer Vision networks in the high performance domain, it is too large and complex to be representative of the typical tasks that can be solved by IoT devices. Therefore, the MLPerf Tiny suite includes a simpler classical computer vision benchmark called CIFAR-10 [31] composed by 60000 32x32 RGB images, with 6000 images per class. The 10 different classes represent airplanes, cars, birds, cats, deers, dogs, frogs, horses, ships and trucks. The dataset is split in 5 training batches and 1 testing batch, each one contains 10000 images.

The reference model is a ResNet-8 architecture [32]. This family of DL models was proposed to improve the training behavior when the number of layers starts to grow. The novelty introduced are residual skip connections which simply performs an identity mapping whose output is added to the output of some stacked layers. Identity layers do not add trainable parameters. This intuition brought huge improvements in the performance achieved. ResNet won the ILSVRC 2015 competition and made feasible the training of deeper networks.

The accuracy reached by the reference model provided by MLPerf Tiny over a set of 200 CIFAR-10 images is 88%. Since CIFAR-10 has been historically used as target dataset in many TinyML systems, MLPerf Tiny developers decided to create a reference target set to compare future works with

previous ones.

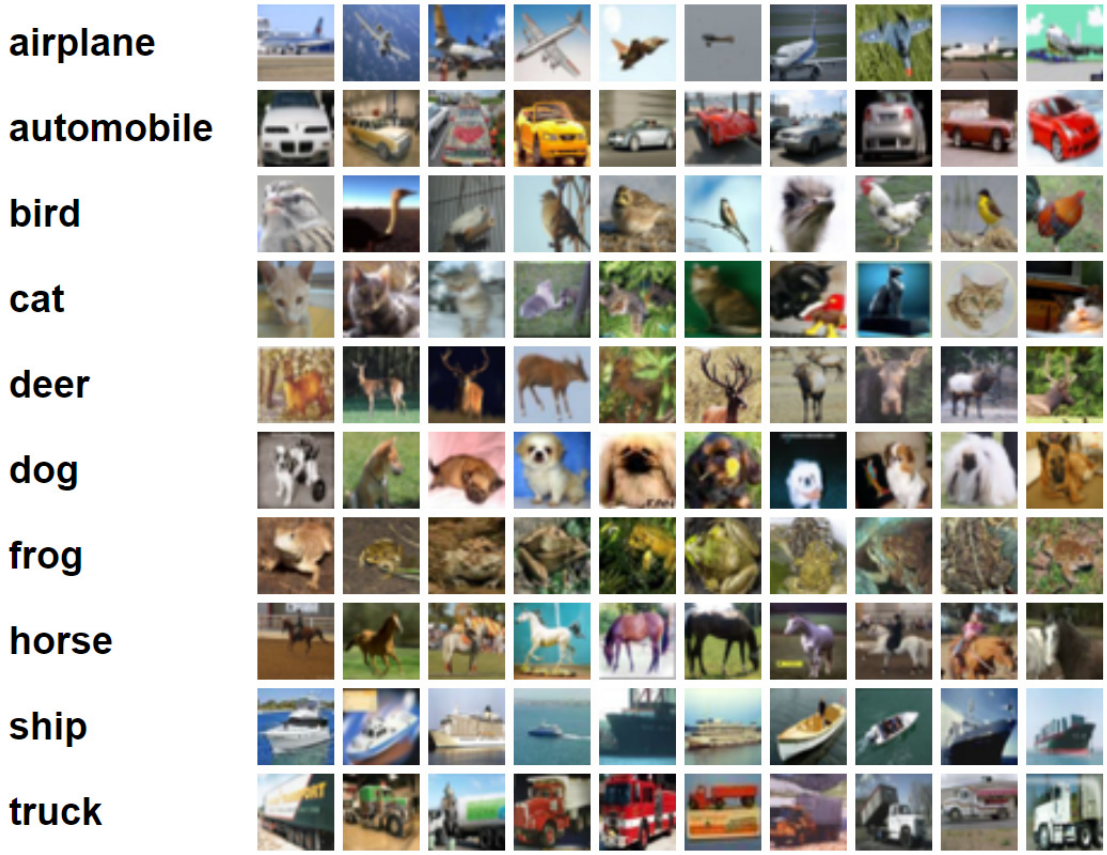


Figure 5.3: CIFAR-10 [31] dataset samples.

5.3 Visual Wake Words

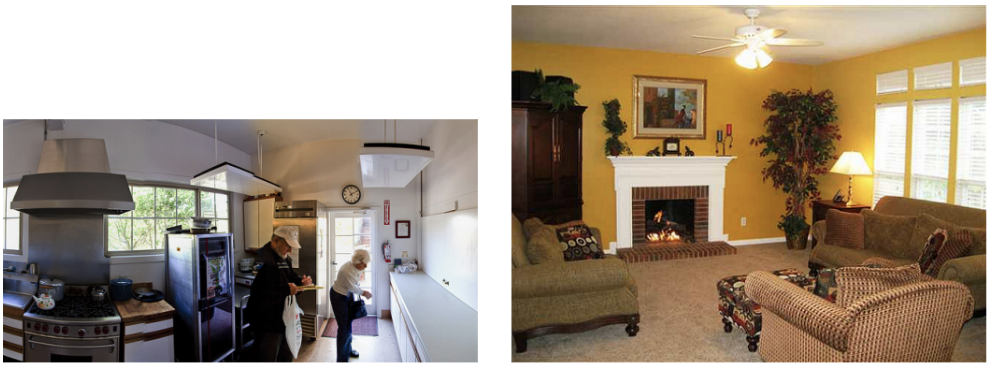
The Visual Wakewords benchmark [33] is a binary classification task based on detecting whether at least one person is in an image. To facilitate the development of models suitable for microcontrollers, the Visual Wake Words benchmark provides a dataset specifically designed for a typical MCU use-case, which is sensing whether a person is present in the camera field of view or not. The detection of an object of interest can trigger an alert or simply be collected for further analysis.

Image sensor are becoming very popular in the equipment of IoT devices since the price is quite affordable and the range of application goes from industrial to home automation.

However, the vast majority of image datasets are not suited for a microcontroller use case. ImageNet [29] has too many classes for a classical microcontroller use-case and it does not provide samples of a person class. CIFAR-10 [31] dataset is designed to train models that perform inferences on the edge but it has some drawbacks. The limited resolution of the images (32x32) limits the capacity of the models trained over it.

Therefore, the Visual WakeWord MLPerfTiny benchmark uses a subset of the public MSCOCO dataset and is composed by 115k images divided in training and validation subsets, each containing examples of person and not-person images. The original MSCOCO dataset (Microsoft COCO Common Objects in COntext) [34] is a very popular benchmark for segmentation and object detection tasks. It contains 91 object categories and more than 2.5 million labeled instances gathering complex everyday scenes presenting common objects in their natural context.

The researchers called the MLPerf Tiny version of this dataset Visual Wake Words because, similarly to how keywords are used in Speech Recognition, the images allow the MCU to wake up if a person is spotted. Starting from the COCO dataset if an image contains a bounding box with a person greater than the 2.5% of the area, that image is categorized under the "person" label, otherwise it becomes a "non-person" picture as shown in Fig. 5.4.



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

Figure 5.4: Visual Wake Words [33] dataset samples.

The baseline architecture is a MobileNet [35] network. MobileNet has been specifically designed to build small and low-latency models for mobile and embedded vision scenarios. The cornerstones of this network are three:

depthwise separable convolutions, the *width multiplier* and the *resolution multiplier*.

Depthwise separable convolutions are composed by two layers: a depthwise convolution and a pointwise convolution. This separation allows to decrease the number of parameters involved and to augment the efficiency of the network.

The second novelty introduced is the width multiplier, which is a simple parameter $\alpha \in (0,1]$ that multiplies the number of input and output channels of each layer. With an $\alpha < 1$ the number of parameter and the computational cost is reduced of around α^2.

The third new aspect is the resolution multiplier parameter $\rho \in (0,1]$. This parameter reduces the resolution of the input images and consequently the internal representations of the following layers. Exactly as the width multiplier, also the resolution multiplier cuts the computational cost of ρ^2.

The baseline accuracy for this benchmark is 83.32%.

5.4 Keyword Spotting

Nowadays speech technologies recognition are omnipresent, especially with the huge proliferation of voice assistants such as Apple's Siri, Amazon's Alexa and Google's Assistant. An important aspect of such application is the power consumption. To maximize the battery life, the complete set of functionalities of these models is not always available.

By default, voice assistants monitor the audio streams looking for specific wake-up words. Once a specific keyword is detected the model becomes fully operative and a larger processor is enabled to support it. Keyword Spotting [36] is defined as the task of identifying keywords in audio streams comprising speech. In this way during the inactivity period the energy consumption is reduced to the minimum. Deep Learning models outperform previous Machine Learning techniques in Speech Recognition, similarly to the Computer Vision field, therefore the majority of today's systems are based on DL architectures.

A general pipeline of a modern keyword spotting deep learning system is composed by the following steps: a speech feature extractor which converts the input signal from time domain to frequency domain with the Short Time Fourier Transformation (STFT) and produces the spectrogram. Human ears do not perceive the differences in Hertz scale linearly. The log-Mel scale is a non-linear transformation of the frequency scale designed to mimic the

human perception. It emphasizes the lower frequency region more than the higher frequencies. The computed magnitude spectrogram is mapped to a Mel scale and to a log operator to compute the log-Mel spectrogram. Once finished this preprocessing step, the spectrogram becomes the input of the model.

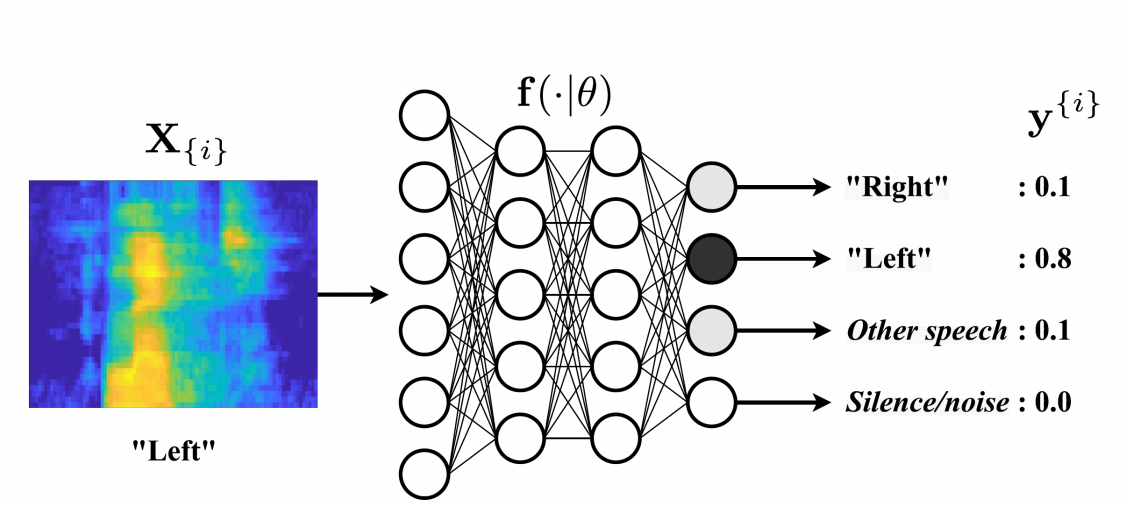


Figure 5.5: Speech recognition Deep Learning model [36].

The MLPerf Tiny benchmark for keyword spotting uses the Speech Commands dataset [37]. Its primary goal is to provide a manner to train and test small models to detect when a given word is spoken between a set of ten or fewer words. Since most of the input audio is silence or background noise, the false positives must be minimized to achieve good performance. Furthermore, the models must deal with low quality equipments, noisy environments, people talking and conversational speechs. To create a dataset as faithful all possible to these situations, all samples have been recorded through laptop or phone microphones. The utterances were captured by many different subjects to reduce as much as possible the personalization and build speaker-independent models. Any personally-identifiable information, like gender or ethnicity, has not been collected to guarantee the privacy of the subjects involved. To facilitate the training and testing phases, each word has been captured in isolation and each sample has a standard duration of 1 second. The vocabulary in this scenario is composed by 10 words: “Yes”, “No”, “Up”, “Down”, “Left”, “Right”, “On”, “Off”, “Stop”, and “Go”, which are very common in IoT scenarios, plus the 2 additional classes “Silence” and “Unknown”.

Similarly to the previous benchmark, Visual Wake Words, the reference baseline model is a composed by depthwise separable convolutions with only 38.6K parameters, and reaches an accuracy of 91.88%.

5.5 Library organization

The MLPerf Tiny suite is only provided with code using the TensorFlow deep learning framework, and without a unified API to work on different tests using the same set of functions. Having such a common API would greatly simplify the application and evaluation of optimization techniques, and in particular, in the case of this thesis, of the PIT NAS, to all four benchmarks included in the suite. Therefore, during the thesis, part of the work has been devoted to porting the suite to the Pytorch framework, and to the development of a common API to provide a standard approach to evaluate different NAS algorithms over four MLPerf Tiny datasets.

The MLPerf Tiny benchmarks have been included in the library with a dedicated subfolder containing the refactored MLPerf Tiny scripts in Pytorch. Then in the library root folder the FlexNAS algorithm is applied over the four different datasets. In the following two sections an overall description of the organization and the structure of these two components is provided. Moreover, the set of common functions of the API will be explained.

5.5.1 Pytorch benchmarks

The *pytorch-benchmarks* package developed during the thesis contains the MLPerf Tiny benchmarks implementation in Pytorch. In this repository the PIT algorithm is not called, the purpose of the folder is to reproduce exactly the same results obtained with the original implementation in TensorFlow using Pytorch.

A set of standard functions and scripts has been defined at the beginning in order to facilitate the test and the evaluation of DNAS algorithms such PIT through several different datasets without caring of the differences in terms of shape of input data, reference neural networks etc.

Every benchmark is associated a corresponding subpackage, characterized by the same set of python files: data.py, model.py and train.py. The data.py module exports the functions responsible for data collection, data preprocessing and data transformation.

In every data.py there are two functions:

- `pytorch_benchmarks`
  - `anomaly_detection`
    - `data.py`
    - `model.py`
    - `train.py`
  - `image_classification`
    - `data.py`
    - `model.py`
    - `train.py`
  - `keyword_spotting`
    - `data.py`
    - `model.py`
    - `train.py`
  - `visual_wake_words`
    - `data.py`
    - `model.py`
    - `train.py`

Figure 5.6: Pytorch-benchmark library structure.

- `get_data`: this function, as the name suggest, is responsible for the download of the dataset. To avoid loss of time this function checks if the dataset has been already downloaded previously in order to save time in the training phase. It returns three torch Datasets, the training dataset, the validation dataset and the test dataset.

- `build_dataloaders`: The torch DataLoader is the structure over which the training loop is executed. It takes as input a torch Dataset and it allows to specify the batch size, the number of workers needed and if the data should be shuffled or not. This methods returns three torch Dataloaders, respectively training, validation and test.

The model.py submodule exports the functions needed to retrieve the reference neural network architecture used in the MLPerf Tiny benchmark, or in some cases, also alternative architectures that achieve good results on a given task. The main function to retrieve the model is `get_reference_model`:

- `get_reference_model`: this function takes as input parameter the name of the Pytorch model to return, this allows to choose between different architectures in the training phase.

The train.py submodule exports the functions needed to retrieve the loss function, the optimizer and the learning rate scheduler that are used in the training loop of the original MLPerf Tiny benchmark. It provides also a default training loop structure and a default evaluation function that could be used when there are not specific modifications required in the training phase:

- `get_default_optimizer`: this method returns the optimizer used in MLPerf Tiny library with the learning rate and weight decay to apply.
- `get_default_criterion`: this method returns the loss function applied in MLPerf Tiny benchmark.
- `get_default_scheduler`: this method takes as input the optimizer and it returns a scheduler object to manage the learning rate scheduling.
- `train_one_epoch`: this function implements the training loop for a single epoch using the standard protocol defined by the benchmark. It is recalled by the main scripts for the number of epochs needed.
- `evaluate`: the standard evaluation function over the validation or test set. This structure reproduces the standard evaluation method proposed in the MLPerf Tiny benchmark.

The `train_one_epoch` and `evaluate` functions are not retrieved in the NAS phase because they are not applying the regularization term.

- `pytorch_benchmarks`
  - `anomaly_detection`
  - `image_classification`
  - `keyword_spotting`
  - `ivisual_wake_words`
- `anomaly_detection_example.py`
- `image_classification_example.py`
- `keyword_spotting_example.py`
- `visual_wake_words_example.py`

Figure 5.7: The folder structure of the benchmarks package.

In the root folder of pytorch-benchmarks for each dataset a default implementation has been provided. This allows other developers to reproduce and

verify the results obtained by this library and it also shows how the standard functions should been used in the scripts.

```python
import torch
from pytorch_model_summary import summary
import pytorch_benchmarks.image_classification as icl
from pytorch_benchmarks.utils import seed_all

# Check CUDA availability
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Training on:", device)

# Ensure deterministic execution
seed = seed_all(seed=42)

# Get the Data
datasets = icl.get_data()
dataloaders = icl.build_dataloaders(datasets)
train_dl, val_dl, test_dl = dataloaders

# Get the Model
model = icl.get_reference_model('resnet_8')
if torch.cuda.is_available():
    model = model.cuda()

# Model Summary
input_example = torch.unsqueeze(datasets[0][0][0], 0)
print(summary(model, input_example.to(device), show_input=False, show_hierarchical=True))

# Get Training Settings
criterion = icl.get_default_criterion()
optimizer = icl.get_default_optimizer(model)
scheduler = icl.get_default_scheduler(optimizer)

# Training Loop
N_EPOCHS = 500
for epoch in range(N_EPOCHS):
    _ = icl.train_one_epoch(epoch, model, criterion, optimizer, train_dl, val_dl, device)
    scheduler.step()
test_metrics = icl.evaluate(model, criterion, test_dl, device)

print("Test Set Loss:", test_metrics['loss'])
print("Test Set Accuracy:", test_metrics['acc'])
```

```python
import torch
from pytorch_model_summary import summary
import pytorch_benchmarks.visual_wake_words as vww
from pytorch_benchmarks.utils import seed_all

# Check CUDA availability
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Training on:", device)

# Ensure deterministic execution
seed = seed_all(seed=42)

# Get the Data
datasets = vww.get_data()
dataloaders = vww.build_dataloaders(datasets)
train_dl, val_dl, test_dl = dataloaders

# Get the Model
model = vww.get_reference_model('mobilenet')
if torch.cuda.is_available():
    model = model.cuda()

# Model Summary
input_example = torch.unsqueeze(datasets[0][0][0], 0)
print(summary(model, input_example.to(device), show_input=False, show_hierarchical=True))

# Get Training Settings
criterion = vww.get_default_criterion()
optimizer = vww.get_default_optimizer(model)
scheduler = vww.get_default_scheduler(optimizer)

# Training Loop
N_EPOCHS = 50
for epoch in range(N_EPOCHS):
    _ = vww.train_one_epoch(epoch, model, criterion, optimizer, train_dl, val_dl, device)
    scheduler.step()
test_metrics = vww.evaluate(model, criterion, test_dl, device)

print("Test Set Loss:", test_metrics['loss'])
print("Test Set Accuracy:", test_metrics['acc'])
```

Figure 5.8: Image Classification (left) and Visual Wake Words (right) implementation comparison.

The application of a standard methodology allows to manage most of the differences between the benchmarks within the package, in this case inside the data.py, model.py and train.py scripts, and uniforms at high level the code, providing a simple and uniform interface. In the Fig. 5.8 are shown the implementation regarding Image Classification and Visual Wake Words benchmarks. The code structure is exactly the same, to switch from a benchmark to another a simple change of the package pointer, the number of epochs and the name of the network is required.

5.5.2 NAS Application Code

The NAS methods defined in flexnas and the uniform training and testing API defined in the aforementioned benchmarks package are designed to be easily combined together, thus facilitating the evaluation of new NAS approaches. This is achieved by importing the flexnas repository and the pytorch-benchmarks submodules to make available all the methods and functionalities needed to perform the optimization search. A submodule is linked to a specific commit of the repository. In this way the code execution remains stable, and when some bugs have been corrected in the source code the submodule is updated.

In the thesis, thanks to the uniform interface, we managed to apply the NAS to all four MLPerf Tiny benchmarks. To this end, we developed four similar search scripts. The given structure decouples the three sections and it allows to extend separately each of the libraries autonomously.

In this way the implementation of each modules can proceed in parallel and facilitate future developments.

Chapter 6

Experimental Results

In this chapter the experimental outcomes of the tested NAS algorithm are shown. In particular, we show that, for all four MLPerf Tiny benchmarks, PIT has been able to achieve better accuracy results compared to the baseline model with a reduced number of parameters. The details for each benchmark are reported in the following sections.

6.1 Anomaly Detection

The Anomaly Detection benchmark is an unsupervised classification task whose baseline model is an AutoEncoder with 267928 parameters. As described in Sec. 5.1 this is an unsupervised learning task where there are no labels available during the training and validation phase, the performance of the model are calculated considering the AUC-ROC curve. The reference top AUC value for the baseline architecture is 85.5%. In the Table 6.1 are reported the AUC-ROC results obtained by PIT with different lambda regularization strength.

Lambda	Auc	# Parameters
8e-3	**79.14**	**26370**
8e-4	83.95	70031
5e-4	85.38	123567
3e-4	**83.97**	**103396**
8e-5	**85.66**	**121375**
3e-5	85.30	127954
8e-6	86.07	126615

Lambda	Auc	# Parameters
5e-6	85.10	137410
3e-6	84.53	125786
8e-7	85.29	141121
3e-7	84.16	125641
9e-8	84.10	125189
8e-8	85.35	128439
7e-8	84.81	130021
6e-8	84.14	126783
5e-8	85.19	122674
3e-8	**86.66**	**135964**
8e-9	85.15	131522
4e-9	85.34	129810
3e-9	85.74	147510
8e-10	84.83	147971
5e-10	85.65	147385
3e-10	85.14	148426
8e-11	85.62	152091
5e-11	85.65	154987
3e-11	85.15	152091
7e-12	84.17	189451
6e-12	85.02	203077
7e-13	84.54	267789
7e-14	86.35	265670
7e-15	84.36	267789
7e-16	85.38	267789

Table 6.1: Pruning In Time algorithm outcomes for the Anomaly Detection benchmark (Pareto points highlighted in bold).

The Pareto points have been plotted in the AUC vs Number of Parameters space in Fig. 6.1. The numbers close to each star correspond to the regularization strength used to obtain it. With a lambda equal to 8e-5 PIT reached an AUC value of 85.66% with only 121375 model parameters, a reduction of -54.70% with the respect to the original model.

The highest AUC has been achieved by the PIT model with regularization strength equal to 3e-8, with an AUC of 86.66%, 1.16% higher than the reference value, and a model composed by 135964 parameters, which implies a -49.25% of size reduction.

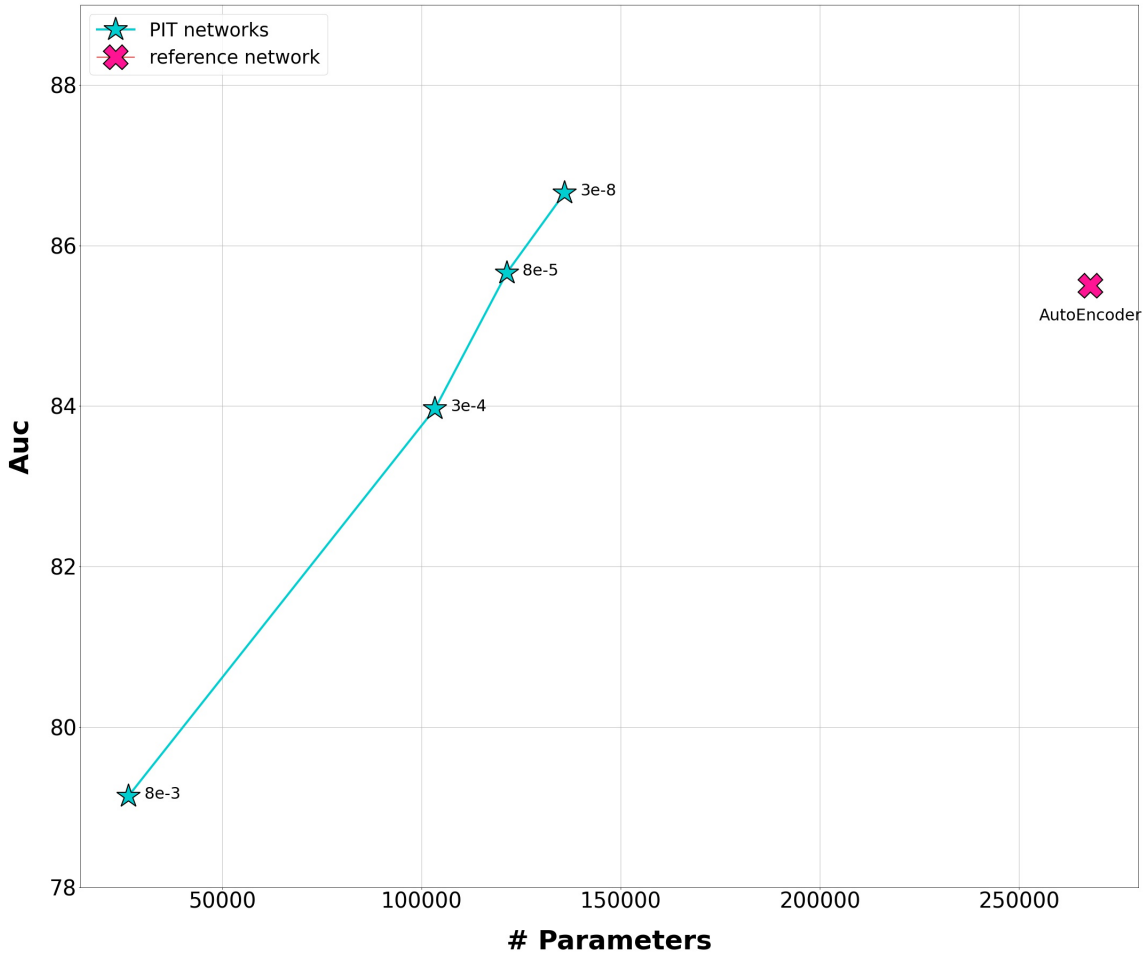


Figure 6.1: Pareto chart in the AUC vs Number of Parameters space for Anomaly Detection.

In other words, this NAS algorithm discovered two new architectures able to obtain the same or better results with roughly half of the size of the original one. This is an important result especially considering further deployment to constrained low memory devices.

6.2 Image Classification

The Image Classification benchmark is a supervised classification task where the reference model is a ResNet-8 composed by 78052 parameters achieving a top accuracy of 88%. During the testing phase 32 different regularization strength have been applied in order to deeply evaluate the behavior of PIT. In Table 6.2 are reported for each lambda the accuracy reached and the number of parameters of the optimized architecture.

Lambda	**Accuracy**	**# Parameters**
5e-4	22.5	126
1e-4	65.5	2553
8e-5	**66.5**	**4270**
7e-5	**72**	**6465**
5e-5	76	9925
3e-5	**84.5**	**19417**
2e-5	80	19775
1e-5	**87.5**	**26828**
9e-6	86	25448
8e-6	86	47125
7e-6	84	28890
6e-6	86.5	31023
5e-6	**89.5**	**57110**
3e-6	**89.5**	**68976**
1e-6	87.5	68976
9e-7	87	54369
7e-7	88.5	66942
5e-7	87.5	61446
2e-7	87.5	77319
9e-8	89	74427
8e-8	88.5	74718
7e-8	89	72258
5e-8	89	72231
2e-8	88.5	74427
1e-8	87	70515
5e-9	89	69516
3e-9	**90**	**71805**
2e-9	88	70947

Lambda	Accuracy	# Parameters
1e-9	89.5	75848
5e-10	88.5	70947
5e-11	89.5	76452
5e-12	87	69657

Table 6.2: Pruning In Time algorithm outcomes for the CIFAR-10 dataset (Pareto points highlighted in bold).

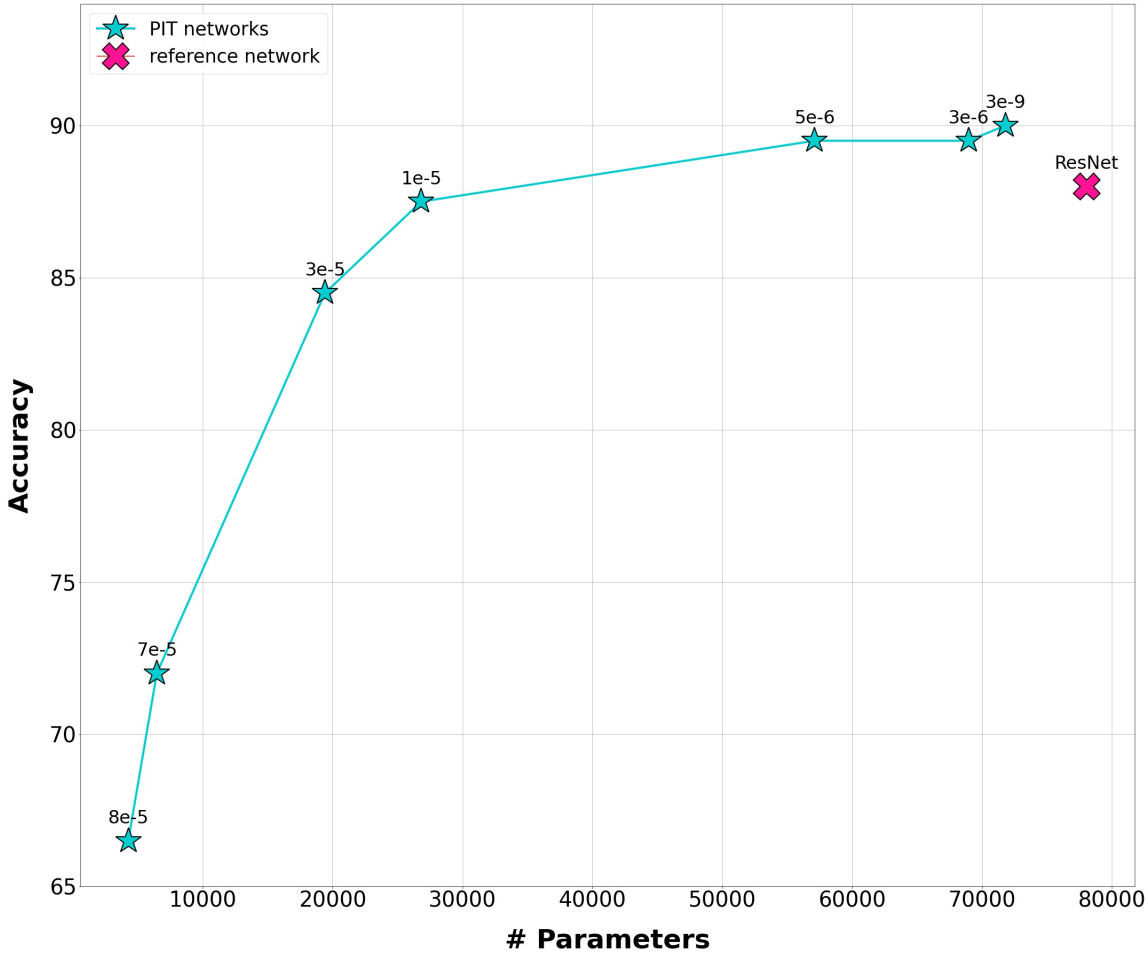


Figure 6.2: Pareto chart in the Accuracy vs Number of Parameters space for Image Classification.

The Pareto points have been plotted in the Accuracy vs Number of Parameters space in Fig. 6.2. PIT achieved an accuracy of 90% with 71805 parameters corresponding to a size reduction of -8%. Furthermore, applying

PIT with a lambda of 5e-6, the network found by the NAS reaches an accuracy of 89.5%, but with a significant reduction of -26.8% in the number of parameters. Lastly, if we accept an accuracy degradation of -0.5%, PIT finds a solution that only requires 26828 parameters, reducing the model size of -65.63%.

6.3 Visual Wake Words

The Visual Wake Words benchmark is a supervised classification task where the reference model is a MobileNet network composed by 213586 parameters whose best result is an accuracy of 83.32%. Several regularization strength have been tested in order to find the best architecture, the results have been collected in following table:

Lambda	**Accuracy**	**# Parameters**
5e-4	51.99	242
4e-4	51.99	242
7e-5	**78.47**	**1607**
5e-5	80.80	2712
3e-5	80.80	2712
8e-6	82.66	4330
3e-6	84.08	8897
5e-7	**84.86**	**17612**
3e-7	85.23	18675
7e-8	**85.28**	**19537**
5e-8	84.87	19483
3e-8	84.18	19405
5e-9	85.04	19347
3e-9	84.99	19398
5e-10	84.90	19447
8e-10	84.86	19409

Table 6.3: Pruning In Time algorithm outcomes for the Visual Wake Words benchmark (Pareto points highlighted in bold).

The Pareto points have been plotted in in the Accuracy vs Number of Parameters space in Fig. 6.3. Applying a regularization strength of 5e-7 PIT discovered a new architecture able to obtain an accuracy of 84.86%, 1.54% higher than the baseline, with only 17612 parameters which implies a global reduction of -91.75% of the original model size. The highest accuracy (85.28%) has been reached with a regularization strength set to 7e-8. With this lambda the obtained model is composed by 19537 parameters, reducing the model size of -90.85% with respect to the baseline.

In this benchmark the architectures found by the NAS have been able to improve the reference accuracy reducing the model size of one order of

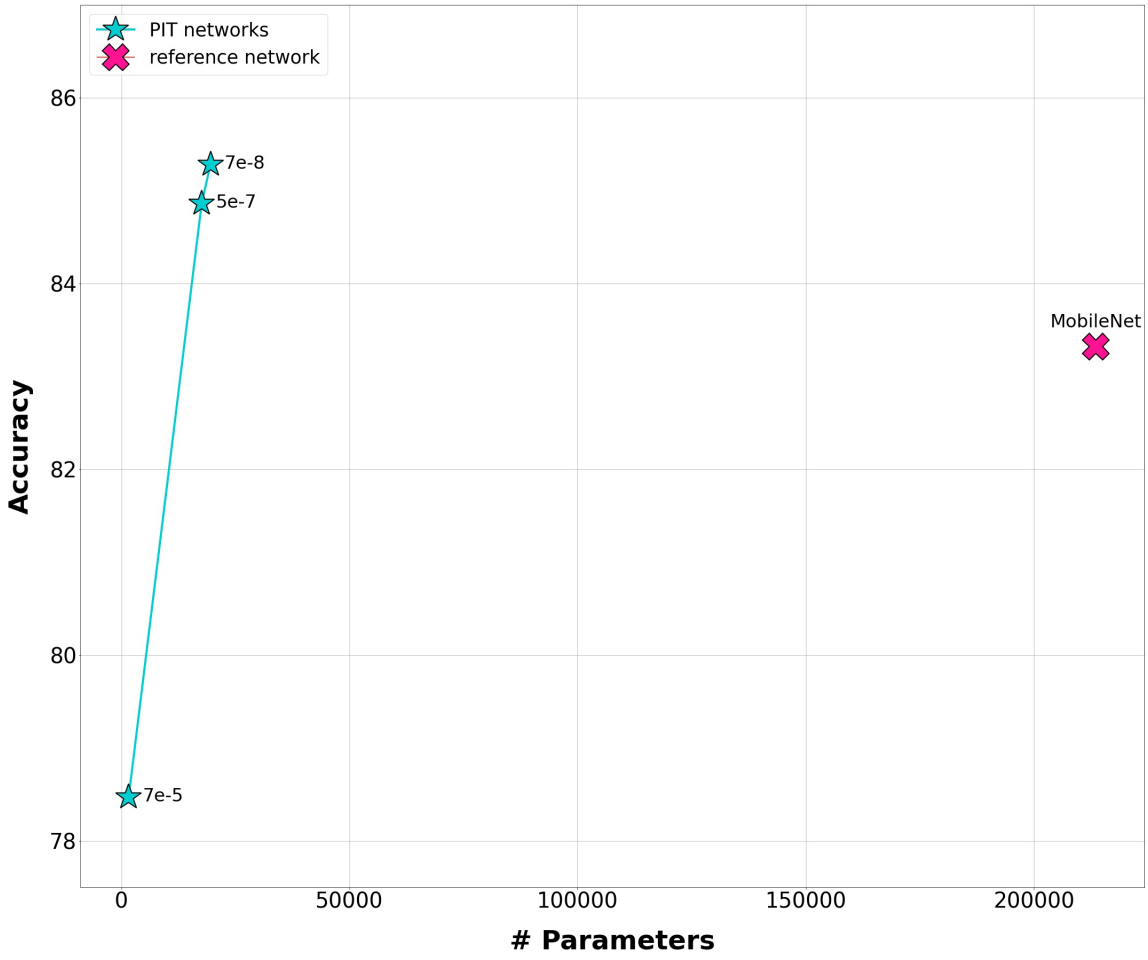


Figure 6.3: Pareto chart in the Accuracy vs Number of Parameters space for Visual Wake Words.

magnitude, indeed the novel networks contain less than 10% of the initial number of parameters, providing a huge improvement for deployment on edge devices.

6.4 Keyword Spotting

The Keyword Spotting is a supervised learning tasks where the reference model is a Depthwise Separable Convolutional Neural Network (DSCNN) with 40396 parameters and achieving a top accuracy of 91.88%. The tested regularization strength have been collected in the following table:

Lambda	Accuracy	# Parameters
5e-5	**83.29**	**14749**
3e-5	**91.64**	**22079**
8e-6	**92.37**	**31547**
5e-6	93.21	36123
3e-6	93.50	37541
8e-7	**93.49**	**37541**
5e-7	93.39	40255
3e-7	86.44	39973
3e-8	88.00	40396
3e-9	88.00	40396
5e-10	89.12	40396
3e-10	88.00	40396

Table 6.4: Pruning In Time algorithm outcomes for the Keyword Spotting benchmark (Pareto points highlighted in bold).

With a regularization strength of 8e-6, the accuracy reached by PIT is 92.37% (0.49% better than the baseline) with a model containing 31547 parameters, i.e., a -21.90% size reduction.

The best accuracy (93.49%) is achieved with a regularization strength of 8e-7, with 37.541 parameters, corresponding to a size reduction of -7%. If we accept an accuracy drop of -0.24%, then PIT finds a model with only 22079 parameters, i.e. -45% less than the baseline.

The DSCNN is designed with depthwise separable convolutional layers which are more efficient layers to perform a convolutional operation. For this reason the overall reduction in term of size and the global improvement of the model have not reached huge results in absolute value like in other benchmarks.

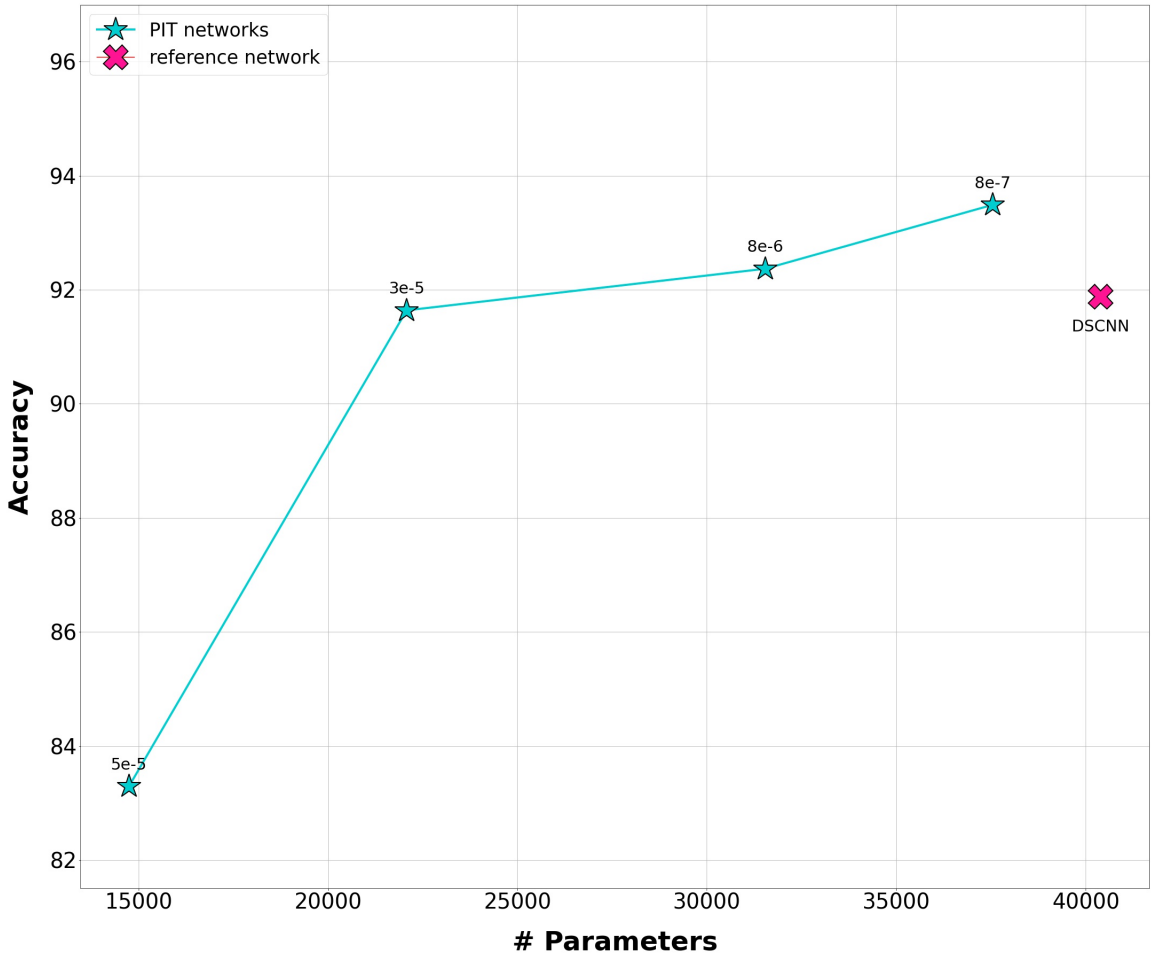


Figure 6.4: Pareto chart in the Accuracy vs Number of Parameters space for Keyword Spotting.

Chapter 7

Conclusion and future works

Artificial Intelligence is experiencing an exponential growth in terms of applications and performance. Deep Learning models nowadays reached a very high popularity thanks to the capability to provide state-of-the-art solutions to several problems in domains such as Computer Vision where the handcrafted feature engineering struggled to achieve acceptable performance.

Despite the astonishing results already obtained, there are many more aspects where Deep Leaning needs further improvements. These aspects foster the growth of new research fields.

A critical feature of DL is the design of the architecture. This task still requires high experience and manual effort, a burden that researchers are trying to alleviate with Neural Architecture Search tools. In particular, NAS for constrained IoT devices tries to optimize the model architecture not only from the perspective of its predictive performance, but also considering complexity, often expressed in terms of number of model parameters or number of Floating point Operations Per Second.

Part of the work of this thesis consisted in studying and reviewing the current NAS literature, including early algorithms based on Reinforcement Learning, and more lightweight Differentiable NAS techniques. Then, the novel Pruning In Time NAS tool has been studied in more depth, since it has been one of the building blocks of this work.

In fact, the purpose of this thesis has been the development of a library to provide the possibility to evaluate PIT, or other similar NAS algorithms over the four benchmarks defined in the MLPerf Tiny suite. A standard background has been created by a complete refactor of the MLPerf Tiny

code using the Pytorch framework. Then a set of common functions has been defined in order to provide a standardized API to the final user. Future researchers or developers can easily evaluate and reproduce the PIT algorithm results over the four datasets. This library has been specifically built to facilitate the cross testing of several NAS algorithms and its modular structure can be further extended by future works, constituting a solid basis for fair and easy testing of TinyML models and optimization tools.

The results obtained over the four benchmarks demonstrated the capability of PIT to discover more efficient model architectures, able to obtain better performance with a significant reduction in terms of number of parameters. As inference on the edge is growing its importance and use cases, PIT constitutes a reliable techniques to deploy more efficient and accurate neural networks on edge devices.

Bibliography

[1] XU, Min, et al. «The fourth industrial revolution: Opportunities and challenges». In: International journal of financial research, 2018, 9.2: 90-95. (cit. on p. 13).

[2] Alzubaidi, L., Zhang, J., Humaidi, A.J. et al. «Review of deep learning: concepts, CNN architectures, challenges, applications, future directions». In: J Big Data 8, 53 (2021) (cit. on p. 14).

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton., «Imagenet classification with deep convolutional neural networks». In: Communications of the ACM 60.6 (2017) (cit. on p. 14).

[4] Zachary C Lipton, John Berkowitz, and Charles Elkan. «A critical review of recurrent neural networks for sequence learning». In: arXiv preprint arXiv:1506.00019 (2015) (cit. on p. 14).

[5] R. Sarikaya, G. E. Hinton and A. Deoras, «Application of Deep Belief Networks for Natural Language Understanding». In: IEEE/ACM Transactions on Audio, Speech, and Language Processing, vol. 22, no. 4, pp. 778-784, (2014) (cit. on p. 14).

[6] Harris, «Many-core GPU computing with NVIDIA CUDA». In: Int. Conf. Supercomputing, (2008) (cit. on p. 14).

[7] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. «Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations». (2016) (cit. on p. 15).

[8] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. «Degree-quant: Quantization-aware training for graph neural networks». In: International Conference on Learning Representations. (2021) (cit. on p. 15).

[9] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. «Learning to prune deep neural networks via layer-wise optimal brain surgeon». arXiv preprint arXiv:1705.07565. (2017) (cit. on p. 16).

[10] Shaohui Lin, Rongrong Ji, Yuchao Li, Yongjian Wu, Feiyue Huang, and

Baochang Zhang. «Accelerating convolutional networks via global & dynamic filter pruning». In: IJCAI, pages 2425–2432. (2018) (cit. on p. 16).

[11] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. «MobileNets: Efficient convolutional neural networks for mobile vision applications». arXiv preprint arXiv:1704.04861, (2017) (cit. on p. 16).

[12] Yani Ioannou, Duncan Robertson, Roberto Cipolla, and Antonio Criminisi. «Deep roots: Improving cnn efficiency with hierarchical filter groups». In: Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1231–1240, (2017) (cit. on p. 16).

[13] Hanxiao Liu, Karen Simonyan, and Yiming Yang. «Darts: Differentiable architecture search». In: arXiv preprint arXiv:1806.09055 (2018) (cit. on p. 16).

[14] Alvin Wan et al. «Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions». In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 12965–12974. (2020) (cit. on p. 16, 39, 41, 42, 43)

[15] M. Risso et al., «Pruning in time (pit): A light-weight network architecture optimizer for temporal convolutional networks». In Proc. 58th DAC, pp. 1–6. (2021) (cit. on p. 16, 45)

[16] Srivastava, Nitish, et al. «Dropout: a simple way to prevent neural networks from overfitting». The journal of machine learning research 15.1 (2014): 1929-1958 (cit. on p. 27).

[17] Qin, Zhuwei, et al. «How convolutional neural network see the world-A survey of convolutional neural network visualization methods». arXiv preprint arXiv:1804.11191 (2018) (cit. on p. 29, 30, 31).

[18] Sergey Ioffe and Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». In: arXiv preprint arXiv:1502.03167 (2015) (cit. on p. 31).

[19] Bai, Shaojie, J. Zico Kolter, and Vladlen Koltun. «An empirical evaluation of generic convolutional and recurrent networks for sequence modeling». arXiv preprint arXiv:1803.01271 (2018) (cit. on p. 33).

[20] B. Zoph and Q. V. Le, «Neural architecture search with reinforcement learning». arXiv preprint arXiv:1611.01578, 2016. (cit. on p. 35, 36)

[21] Ronald J Williams. «Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning». Machine Learning, 8(3-4): 229–256, (1992) (cit. on p. 36)

[22] Zoph, Barret, et al. «Learning transferable architectures for scalable image recognition». Proceedings of the IEEE conference on computer vision

and pattern recognition. 2018 (cit. on p. 37).
[23] H. Liu et al., «Darts: Differentiable architecture search». arXiv:1806.09055, 2019 (cit. on p. 37).
[24] A. Gordon et al., «Morphnet: Fast & simple resource-constrained structure learning of deep networks». In: Proc. of the IEEE CVPR, pp. 1586–1595 (2018) (cit. on p. 39)
[25] M. Courbariaux et al., «Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1». arXiv preprint arXiv:1602.02830, 2016 (cit. on p. 47).
[26] Colby R. Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman et al., «MLPerf Tiny Benchmark». In: Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, (2021) (cit. on p. 57)
[27] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton van den Hengel. «Deep Learning for Anomaly Detection: A Review». ACM Comput. Surv. 1, 1, Article 1, (2020) (cit. on p. 59)
[28] Y. Koizumi, S. Saito, H. Uematsu, N. Harada, and K. Imoto. «Toyadmos: A dataset of miniature machine operating sounds for anomalous sound detection». In: IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA), pages 313–317. (2019) (cit. on p. 59)
[29] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. «ImageNet Large Scale Visual Recognition Challenge». International Journal of Computer Vision (IJCV), 115(3):211–252, (2015) (cit. on p. 61)
[30] Krizhevsky A, Sutskever I, Hinton GE «ImageNet Classification with Deep Convolutional Neural Networks». NIPS'12 Proc 25th Int Conf Neural Inf Process Syst 1:1097–1105 (2012) (cit. on p. 61, 63)
[31] A. Krizhevsky, V. Nair, and G. Hinton. «Cifar-10». (canadian institute for advanced research). (2009) URL: http://www.cs.toronto.edu/ kriz/cifar.html. (cit. on p. 61, 63)
[32] K. He, X. Zhang, S. Ren, and J. Sun. «Deep residual learning for image recognition». In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, (2016) (cit. on p. 62)
[33] A. Chowdhery, P. Warden, J. Shlens, A. Howard, and R. Rhodes. «Visual wake words dataset». CoRR, abs/1906.05721, 2019. URL http://arxiv.org/abs/1906.05721. (cit. on p. 62)
[34] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollar, and C Lawrence Zitnick. «Microsoft

coco: Common objects in context». In: European conference on computer vision, pages 740–755. Springer, (2014) (cit. on p. 63)

[35] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. «Mobilenets: Efficient convolutional neural networks for mobile vision applications». arXiv preprint arXiv:1704.04861, (2017) (cit. on p. 64)

[36] López-Espejo, Iván, et al. «Deep spoken keyword spotting: An overview». IEEE Access (2021) (cit. on p. 64)

[37] Warden, Pete. «Speech commands: A dataset for limited-vocabulary speech recognition». arXiv preprint arXiv:1804.03209 (2018) (cit. on p. 65)