POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

POPNASv2:
Efficient Neural Architecture Search through Time-Accuracy Optimization

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Andrea Falanti**

Student ID: 944954
Advisor: Prof. Matteo Matteucci
Co-advisors: Ing. Eugenio Lomurno, Ing. Stefano Samele
Academic Year: 2021-22

Abstract

Automating the research for the best neural network model is a task that has gained more and more relevance in the last few years. In this context, Neural Architecture Search (NAS) represents the most effective technique whose results rival the state of the art hand-crafted architectures. However, this approach requires a lot of computational capabilities as well as research time, which make prohibitive its usage in many real-world scenarios. With its sequential model-based optimization strategy, Progressive Neural Architecture Search (PNAS) represents a possible step forward to face this resources issue. Despite the quality of the found network architectures, this technique is still limited in research time. Further improvements have been done by Pareto-Optimal Progressive Neural Architecture Search (POPNAS), which expand PNAS with a time predictor to enable a trade-off between search time and accuracy, considering a multi-objective optimization problem.

This thesis proposes a new version of the Pareto-Optimal Progressive Neural Architecture Search, called POPNASv2. This work enhances its first version, improving its performance regarding both search time and accuracy of the top networks. The search space has been expanded with new operator, while the quality of both predictors have been improved to build more accurate Pareto fronts. Moreover, this new search strategy introduces cell equivalence checks and an adaptive greedy exploration step. These changes allow POPNASv2 to achieve PNAS-like performance with an average 4x factor research time speed-up.

Keywords: neural architecture search, deep learning, supervised learning, image classification, Pareto-optimality, machine learning

Abstract in lingua italiana

Automatizzare la ricerca di architetture di reti neurali efficaci per un determinato dataset è diventato un ambito estremamente rilevante negli ultimi anni. La tecnica più efficace in questo contesto è la "Neural Architecture Search" (NAS), ovvero l'utilizzo di algoritmi in grado di ricercare reti neurali ottimali all'interno di uno spazio di ricerca ben definito. Le architetture trovate da questi algoritmi competono o superano l'accuratezza raggiunta da architetture progettate dall'uomo, però molto spesso il processo di ricerca richiede l'utilizzo di molta potenza computazionale per lunghi periodi di tempo. Con la sua ricerca sequenziale, Progressive Neural Architecture Search (PNAS) rappresenta un possibile passo avanti nel ridurre l'utilizzo di risorse necessarie per questi tipi di metodi. Nonostante l'alta qualità delle reti trovate e la maggiore efficienza rispetto ai metodi precedenti, questa tecnica richiede ancora considerevoli risorse per essere attuata. La prima versione di Pareto-Optimal Progressive Neural Architecture Search (POPNAS) migliora ulteriormente l'efficienza di PNAS, espandendolo con un predittore dei tempi di training, permettendo di considerare la ricerca come un problema di ottimizzazione combinato di tempi e accuratezze.

Questa tesi propone una nuova versione di Pareto-Optimal Progressive Neural Architecture Search, chiamata POPNASv2, migliorandone sia i tempi di ricerca che l'accuratezza raggiunta dalle reti migliori. Lo spazio di ricerca è stato esteso con nuovi operatori, mentre la qualità dei predittori è stata migliorata al fine di costruire fronti di Pareto più accurati. Inoltre, questo nuova strategia di ricerca introduce controlli sulla equivalenza delle celle e uno step di esplorazione adattivo. Questi cambiamenti permettono a POPNASv2 di raggiungere accuratezze simili a PNAS, riducendo i tempi di ricerca di un fattore 4x.

Parole chiave: neural architecture search, deep learning, apprendimento supervisionato, classificazione di immagini, ottimo paretiano, machine learning

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

2 Background knowledge **5**
2.1 Machine learning 5
2.2 Deep learning 8
2.2.1 Artificial neural networks 10
2.2.2 Artificial neural networks fundamental topologies 13

3 State of the art **19**
3.1 NAS components 20
3.1.1 Search space 21
3.1.2 Search strategy 23
3.1.3 Performance estimation strategy 27
3.2 NAS benchmarks 31
3.2.1 NAS-Bench-101 31
3.2.2 NAS-bench-201 33
3.3 PNAS 34
3.3.1 Search space 34
3.3.2 Search strategy 36
3.3.3 Performance estimation strategy 36
3.4 POPNAS 37
3.4.1 Search space 37
3.4.2 Search strategy 38

3.4.3 Performance estimation strategy . . . 39

4 Method: POPNASv2 43
4.1 Search space . . . 43
4.1.1 Blocks equivalence . . . 44
4.1.2 Cells equivalence . . . 45
4.1.3 Search space cardinality . . . 47
4.2 From cell specification to actual CNN . . . 48
4.2.1 Generating the cell . . . 50
4.3 Search strategy . . . 51
4.3.1 Expansion step . . . 52
4.3.2 Exploration step . . . 54
4.4 Performance estimation strategy . . . 56
4.4.1 Accuracy predictor . . . 56
4.4.2 Time predictor . . . 58

5 Ablation studies 61
5.1 Equivalence check . . . 61
5.2 Exploration step . . . 62
5.3 Time predictors features . . . 63
5.3.1 POPNAS feature set . . . 63
5.3.2 Tentative feature set . . . 64
5.3.3 Definitive feature set . . . 68
5.4 Time predictors . . . 72

6 Experiments and results 73
6.1 Experiments setting . . . 73
6.2 Predictors results . . . 75
6.3 POPNASv2 vs PNAS . . . 76

7 Conclusions and future developments 81

Bibliography 83

A Appendix A 87
A.1 Top1 cells comparisons . . . 87

A.2 Inputs and operators usage comparison 90

B Appendix B **99**

B.1 Software implementation 99

B.2 User guide 100

List of Figures **107**

List of Tables **109**

1 | Introduction

Nowadays, deep neural networks achieves state-of-the-art performances in many tasks, for example image classification, object recognition and text processing. The main strength of these models is that they can extract features directly from the data, automatizing the feature engineering step which is instead fundamental for classical machine learning models. This property allows deep neural networks to perform better than other techniques when the feature set is hard to design, for example in computer vision tasks. It was evident when AlexNet [16] achieved in 2012 a top-5 error of 15.3% on ImageNet [31], a challenging dataset for image classification, composed by more than a million images, divided in 1000 classes. AlexNet top-5 error was more than 10.8 percentage points lower than that of the runner-up, that used instead machine learning techniques for computer vision.

Still, artificial neural networks are very complex models and therefore their training process is far more complicate and computationally demanding than most other artificial intelligence techniques. Even if the theory related to artificial neural network models dates back to '40, these architectures become popular only in the last decade, thanks to the progress in computational capacity, the exploitation of GPU acceleration and the availability of huge amounts of data.

With the rising adoption of deep neural networks, much effort have been spent on architecture engineering, i.e. the process in which a neural network model is designed, with the goal of optimizing its performance on a given task and dataset. Since deep neural networks are very complex models, whose internal results are difficultly interpretable by humans, it is challenging to design how a neural network should extract the features from the available data. Some empirical rules and common knowledge have been gained through works in the literature, still, there are no precise rules on how to define good architectures. Architecture engineering is therefore a tedious and time-consuming task.

The automation of the architecture engineering step is a core research direction, addressed in neural architecture search (NAS) field. NAS algorithms can search autonomously a set of architectures that achieves good accuracy results for the problem considered. NAS

methods rely on three main components: a *search space* for all the possible architectures, a *search strategy* to explore it, aided by a *performance estimation strategy* to identify the most promising networks to explore.

Despite the high quality of the neural networks obtained by exploiting the NAS technique, the enormous amount of time and computational resources required do not allow a widespread use of this approach. Progressive Neural Architecture Search (PNAS) [18] came as an effective solution to this issue. With its sequential model-based optimization search strategy, this algorithm iteratively increases the researched neural networks complexity. The progressive exploration starts from the simplest cells of the search space, and then progressively expands them with new blocks, based on their quality estimated by an auxiliary model called predictor.

Pareto-Optimal Progressive Neural Architecture Search (POPNAS) [20] introduces a new time regressor to further improve the search speed. Between cells with the same accuracy, those generating slow networks are removed from the exploration step. POPNAS doubles the speed of the search compared to PNAS, at the cost of a 8% reduced accuracy on the top-1 network found.

This thesis presents a new version of Pareto-optimal Progressive Neural Architecture Search (POPNASv2), which solves the tradeoff introduced by POPNAS, reducing the total search and training time without any loss in accuracy. POPNASv2 best models rival the ones found by PNAS, but they are produced with almost 4 times less the amount of GPU hours.

The main differences between the two versions can be summarized in 4 points:

- a more refined and complex trained procedure, designed to reach very good performances on few epochs, while preserving a good time efficiency.
- the usage of a more advanced time predictor, which feature set has been redesigned from scratch compared to the first version.
- an equivalence check, used to remove equivalent models from the search space and therefore avoiding to waste unnecessary resources on the training of these networks.
- the introduction of an adaptive greedy exploration step, which leads to a better and less biased exploration of the search space.

The thesis is structured as follows:

Chapter 2 presents a brief summary of the theoretical knowledge required for understanding the concepts later explained in the thesis. In particular, it describes the machine

learning and deep learning fields, discussing the innovation provided by their models but also the drawbacks to address for using them correctly.

Chapter 3 introduces in details the Neural Architecture Search field, which algorithms are based on three components: search space, search method and performance evaluation strategy. All these components are accurately described, listing the different implementation choices and the most relevant works in the literature. It also presents the NAS algorithms on which POPNASv2 is based upon.

Chapter 4 illustrates how POPNASv2 implements the three NAS components, giving a complete self-contained overview of the algorithm. Particular focus is given to the refinements and techniques introduced by the new version.

Chapter 5 describes the ablation studies conducted for the new characteristics introduced by POPNASv2. Each test performed is correlated with a discussion of the results, presenting benefits and drawbacks.

Chapter 6 shows how the experiments have been performed and their final results. A comparison between PNAS and POPNASv2 has been performed to benchmark the search strategy, illustrating the benefits reached by this work through the introduced refinements.

Chapter 7 draws the main conclusions of the work and possible future developments that could lead to further improvements.

Appendix A contains images and plots related to the results obtained during the final experiments.

Appendix B gives more details on the actual software implementation of POPNASv2 and provides a guide on how to install the dependencies and run the algorithm.

2 | Background knowledge

Artificial intelligence (AI) is a computer science field studying how machines can act intelligently and rationally. The goal of AI is to build smart agents, capable of performing tasks which usually require human intelligence.

The AI research field was born at 1956, but it initially struggled due to the limited computational power available at the time. Still, researchers proposed many theories and mathematical models capable of solving complex tasks by exploiting a multitude of different techniques, such as efficiently exploring vast search spaces, learning patterns from data, using inductive logic programming, exploiting probabilistic methods for uncertain reasoning, and many others. Artificial intelligence is significantly thrived in the last 3 decades, differentiating into multiple subfields which specialize in different groups of similar artificial intelligence techniques.

2.1. Machine learning

Machine learning is actually one of the most popular area of artificial intelligence. A popular definition of the machine learning field is proposed by Mitchell [26]:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

The main innovation of machine learning techniques is the change of paradigm compared to traditional programming. Computer programs define an algorithm to process the data (input) and produce an output, instead machine learning models receives the data and the output, producing an algorithm. The name of this AI subfield derives from this behavior, since the models learn how to address the task directly from the data, improving progressively their performance during multiple training iterations.

Considering for example a supervised learning problem, the training dataset $D = \{x, t\}$ contains tuples formed by an input x and its output t (target), which are samples of an unknown function f. The goal of a machine learning model is to learn the function

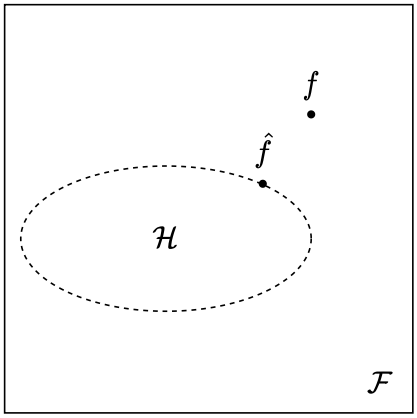


Figure 2.1: A visual example of the hypothesis space $\mathcal{H}$ spanned by a machine model. $\mathcal{H}$ is a subset of the space of all possible functions $\mathcal{F}$. The model goal is to learn a function $\hat{f}$ as close as possible to the true function f.

$\hat{f}$ among the *hypothesis space*, i.e. the hyperspace containing all the functions that can be expressed by the model, which most closely resembles the true function f. A visual example of this concept is provided in Fig. 2.1.

For any task, a loss function must be defined to measure the distance between the prediction $\hat{t}$ and the actual target t. Considering a regression problem, i.e. a task of predicting a continuous value, a popular example of loss function is the mean squared error (MSE):

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(t_i - \hat{t}_i)^2$$

where n is the cardinality of the considered samples.

The parameters of a machine learning model, also referred to as *weights*, are automatically adapted while processing the data. Techniques like gradient descent optimization can minimize the loss function by computing its first derivative and updating the weights to move towards the opposite direction of the gradient. Weight optimization grants to the model the ability to learn a function suited for the addressed task.

Machine learning models are extremely powerful for solving tasks where it is difficult to design an algorithm "by hand", but they require a large amount of data for their training. These techniques are indeed able to make informed and accurate predictions on unseen data, if the training data spans a significant part of the input domain and the models are able to generalize on new data. To do so, a machine learning model requires advanced training techniques and evaluation procedures, where designers optimize the model performances with an iterative workflow, aimed to tune different aspects of it in

each step.

Feature engineering is the first preliminary step required by any machine learning technique, where a contained set of numerical attributes, called features, is extracted from the data. The data can be non-tabular, like when processing images, which require a preprocessing step for extrapolating meaningful features for the task. Even when the data is already tabular, it could contain lots of features not correlated with the target output. Feature preprocessing is essential to optimize the model performance; without good features the models would not be able to learn how to correctly solve the assigned problem.

Furthermore, models must generalize over unseen data, therefore they should not be trained only to minimize the errors on the training data. A small set of the training data is held back to estimate the results of the model on unseen data, since it is easy for a model to just fit the training data with a bad approximation of the true function. This set of data is named *validation set* and it can be used to tune the performance and the generalization capability of the model. The condition in which the model suffers from bad generalization on new data is commonly referred to as *overfitting*, identifiable when the gap between the training loss and the validation loss is significantly high, i.e. the gap between the loss metric applied respectively on the training set and validation set. This behavior makes the model unusable in real-world context, since the results would be inaccurate on new data.

The training loss is an estimation of the bias of the model, while the difference between validation and training losses instead estimates the variance in the results. A good training procedure addresses the problem as a bias-variance trade-off, trying to reduce the gap between training and validation losses with a minimal increase in the prediction bias.

The main way to reduce the bias is to increase the model complexity. This is usually done by adding more weights to the model, which results in expanding its hypothesis space. The bias is reduced since there is an increased probability that the true function f, or a function $\hat{f}$ with a lower loss, is included in the hypothesis space. The drawback is that it is also more probable for the model to learn a function different from f, using a complex model to fit the training points but having a large variance on unseen data.

To reduce the variance, different techniques can be exploited. The most common one is weight regularization, which adds a penalty term to the loss function, proportional to the sum of the norm of the weights. Usually, the L1 norm (Lasso [34]) or L2 norm (Ridge [12]) are used to define the regularization term. Using for example the Ridge regression, the

MSE loss function become:

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(t_i - \hat{t}_i)^2 + \frac{\lambda}{2}||w||_2^2$$

where λ is the regularization factor and w is the vector of weights.

The penalty is proportional to the absolute value of the weights, so the model would converge to smaller weight values after training, since it tries to minimize the loss function. Small weight provides better generalization and lower variance, since the function $\hat{f}$ expressed by the model is smoother and therefore slight variations of the inputs produce small variations also on the outputs. This behavior is common in many physical and mathematical rules that characterize our world, making regularization a valid technique to reduce the variance of the predictions. Constraining the weights has also the effect of increasing the bias, since the hypothesis space restricts, making difficult for the model to accurately fit the training samples for high regularization values. The trade-off can be balanced through the λ parameter.

Both the models and the employed training techniques expose some hyperparameters, which are design variables not learned by the model, unlike the weights. Hyperparameter values control the model structure and its learning behavior, making possible to find the best fit for the considered task. A previously made example of hyperparameter is λ in weight regularization.

Validation loss and other metrics computed on validation set are useful to tune the hyperparameters, performing the so-called *model selection*. By exploiting previous knowledge of the task and performing some trial and error, it is possible to find an optimal set of hyperparameters for the model, used for training the final model structure.

The model is then evaluated on a test dataset, which contains data not previously used in the training and validation set. The results on the test dataset are indicative of the true performance of the model and represent the final step of a complete training procedure. All these step represent a common machine learning workflow for addressing model training, given a specific dataset and task to address.

2.2. Deep learning

Deep learning is a subfield of machine learning that specializes in models based on artificial neurons, mathematical models inspired on the structure of biological neurons. Deep learning models shares many similarities with standard machine learning techniques, but

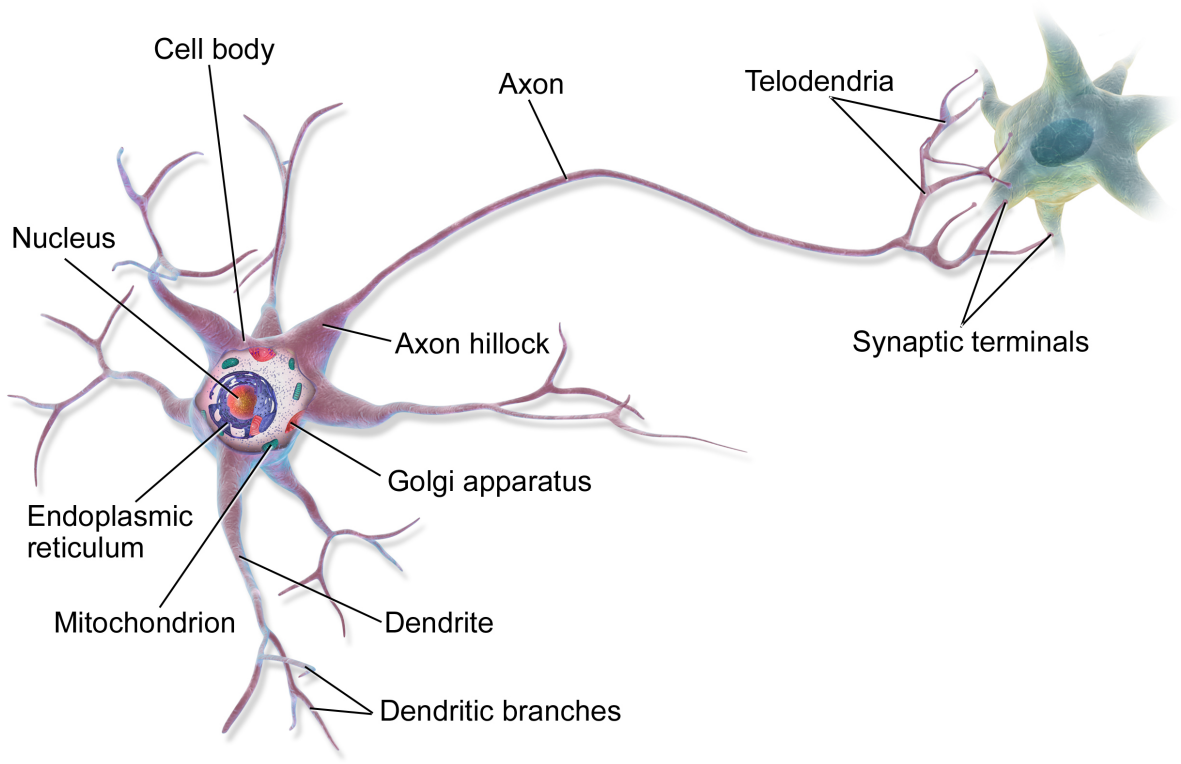


Figure 2.2: The high level structure of a biological neuron, accompanied by the names of its main parts. Source: BruceBlaus, Multipolar Neuron, 2013, Wikimedia Commons. `https://commons.wikimedia.org/w/index.php?curid=28761830`.

they can automatically extract relevant features from the training data, exploiting a single end-to-end process. Automatizing the feature engineering step allows these techniques to produce state-of-the-art results in tasks where the feature definition is innately complex, like image and text processing. This capability comes with the drawback of requiring much more complex models, which have orders of magnitude more weights than other machine learning methods, making them also more computationally demanding and data hungry.

Deep learning thrived in the last decade, thanks to the progress in computational power and the access to large amounts of data, produced by the billions of smart devices nowadays populating the world. The advent of Internet of Things, big data and the interest in optimizing parallel architectures and GPU-accelerated applications have indeed made possible for deep learning to flourish, becoming a stable and popular topic in the AI community.

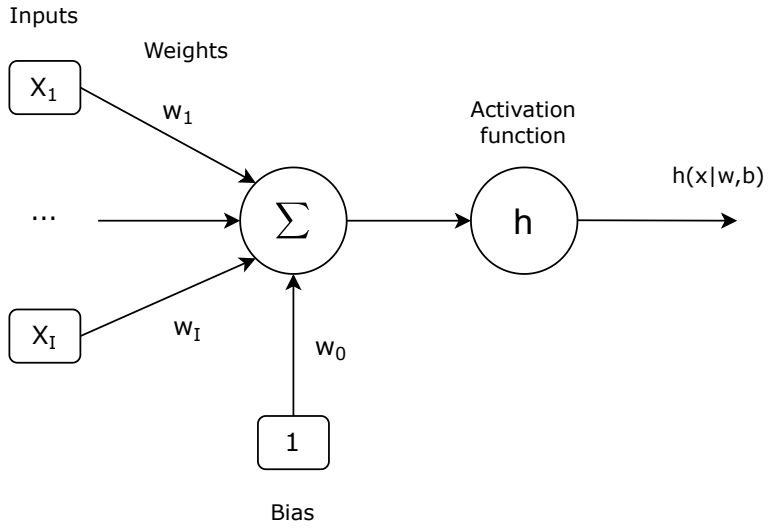


Figure 2.3: The structure of the perceptron unit, an artificial neuron whose mathematical model mimics actual biological neurons.

2.2.1. Artificial neural networks

Deep learning methods are based on artificial neural networks (ANNs), complex models built through the aggregation of multiple artificial neurons, resembling the structure of an organic brain.

Human brains can have about 10^{11} neurons, connected together with more than 10^{14} synapses. The computation model of a biological brain is therefore intrinsically parallel, distributed among simple non-linear units, providing a redundant and fault-tolerant system capable of adapting to any task.

Scientists developed the computational model of the artificial neurons by studying the biological ones. The structure of a biological neuron is presented in Fig. 2.2. Neurons exchange ionic chemical charge to move electric signals across the brain. Dendrites can be considered the inputs of a neuron, which receives the electrical signals from other neurons. The charge is accumulated in the axon hillock until it reaches a certain threshold, after which the neuron "fires" the signal through the axon. The signal propagate through the axon, which can be modeled as the output of the neuron, and then it reaches the synapses, where the dendrites of other neurons can receive the signal.

McCulloch and Pitts [25] (1943) proposed the first computational model of artificial neurons, which they called *threshold linear unit*. Frank Rosenblatt [30] (1958) further improved the threshold linear unit, developing the *perceptron*, illustrated in Fig. 2.3. The inputs x_i simulate the dendrites, while each weight w_i represents the strength of the

synapse between two neurons and if the signal is excitatory (positive weight) or inhibitory (negative weight). The bias is used to model the chemical charge threshold to "fire" the signal, assuming a negative weight w_0. The activation function h chosen to process the charge is the Heaviside step function, which express the non-linearity of the neuron by outputting 1 if the sum of the inputs is positive, 0 if it is negative. The output of an artificial neuron is produced by the formula:

$$h(x|w,b) = h(\sum_{i=0}^{I} w_i \cdot x_i)$$

This model resembles closely how a biological neuron works, passing the signal only if the threshold condition is met.

An artificial neural network is built through the aggregation of multiple perceptron units, organized in layers. Initially, these models were constituted by a single layer of neurons, trained with a technique called *Hebbian learning*, which allows to update the weights in case a target is not correctly predicted. The rules of Hebbian learning can be summarized as follows:

$$w_i^{k+1} = w_i^k + \Delta w_i^k \tag{2.1a}$$

$$\Delta w_i^k = \eta \cdot x_i^k \cdot t^k \tag{2.1b}$$

where η is a hyperparameter called learning rate, k is the current time step and t is the target output of the processed sample.

Hebbian learning can only be applied to single layer artificial neural networks, but these models are too simple for solving complex tasks, since they can only find a linear separation boundary in the inputs hyperspace. This limitation is addressed by feed-forward neural networks, which are composed by multiple layers, each one composed by multiple neurons. These networks are capable of finding arbitrary hyperspace regions, making them able to find an incredibly large set of functions.

To enable their training, LeCun *et al.* [17] designed a technique called *backpropagation*, which allows to update the weights of each layer with standard gradient methods already employed in machine learning field. The procedure is illustrated in Fig. 2.4. Gradient techniques, like stochastic gradient descent, compute deltas to apply on each weight to minimize the defined loss function. Since the output of any neuron depends only on the outputs of the previous layers, it is possible to store a partial derivative of the loss function right during the *forward pass*, the computation phase traversing the entire network from the input layer to the output layer. The *backward pass* instead goes from the output layer to the input layer and updates the weights by applying the derivation chain-rule on all

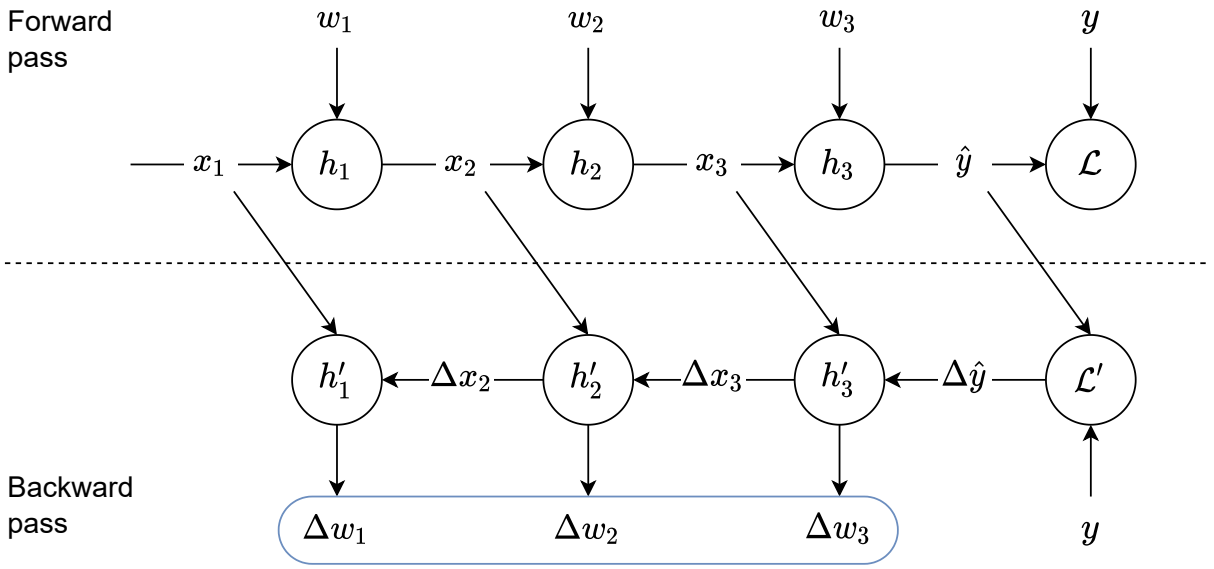


Figure 2.4: A data dependency graph of forward pass and backward pass in a 3-layers ANN. For each activation layer h_i, w_i are its weights, while x_i and h_{i+1} are respectively its inputs and outputs. The outputs of each activation layer must be saved, so that the backpropagation can use them in the chain-rule derivative and compute the weights update Δw_i.

partial derivatives.

Since the application of backpropagation requires the function computed by the artificial neural network to be derivable, the Heaviside step activation function must be replaced with a derivable non-linear function, e.g. sigmoid and hyperbolic tangent. Furthermore, no recurrent connections from a layer to a precedent one are allowed when exploiting backpropagation, because it otherwise would incur in an infinite chain of derivations. The combination of these two passes allows to train any feed-forward neural network, making possible to compose them with a custom amount of layers.

Artificial neural networks are trained for fixed number of training steps, called *epochs*, which involve a complete iteration of the dataset. The dataset is almost always processed in batches, enabling a better parallelization of the computation, which significantly speed-ups the training process. The training of neural networks architectures is a difficult process, since these models expose a significant amount of parameters and hyperparameters. As example, the amounts of neurons per layer, the number of layers, how the weights and the biases are initialized, the learning rate, and many more hyperparameters must be optimized during the model selection phase. Model selection is performed as an iterative trial-and-error process, similar to other machine learning techniques, evaluating which

hyperparameter sets perform better for the addressed task, by comparing their results on a validation set.

Furthermore, significantly deep neural networks can exhibit the so-called *vanishing gradient* problem when optimizing their weights with gradient descent techniques. This behavior is caused by partial derivatives potentially tending to 0, which affect the gradient computed by backpropagation chain-rule. As result, the layer parameters with small gradients would update slowly, stagnating the learning process. Vanishing gradient is addressed with the usage of activation functions that efficiently propagate the gradients, avoiding or reducing the range of input values associated to near zero (vanishing gradient) or much greater than 1 (exploding gradient) outputs in the derivative function. Popular examples of activation functions solving the vanishing gradient problem are the rectified linear unit (ReLU) and its variants, i.e. ELU and leaky ReLU. Weights initialization is also crucial to address the vanishing gradient, since the gradients are also scaled with the weight values. Xavier initialization [6] and He initialization [8] are common choices for initializing the weights of each layer, randomly selecting the value from a normal distribution with 0 mean and variance based on the amount of inputs and outputs of the considered neuron.

Given their complexity, artificial neural networks are also more prone to overfitting, achieving large gaps between the accuracy found on training and validation sets. Techniques frequently exploited to minimize overfitting are *early stopping*, *weight regularization* and *dropout*.

Early stopping simply terminates the training procedure after a set amount of consecutive epochs in which the validation loss does not improve. Weight regularization applies a penalty to the loss function, proportional to the absolute value of the weights. Learning smaller weights leads to less variance on unseen data, as explained in Section 2.1. Dropout technique can be applied as an additional network layer, which masks each received input with a probability p. Masking random neuron outputs in each batch behaves similarly to the training of an ensemble of weaker learners, whose outputs can be averaged to reduce the variance of the predictions on unseen data.

2.2.2. Artificial neural networks fundamental topologies

The previous subsection introduced the main concepts used in artificial neural networks and in their training. Going deeper in the details, artificial neural networks can be implemented with different topologies, based on the task to address.

Traditional feed-forward neural networks are built by stacking multiple densely connected

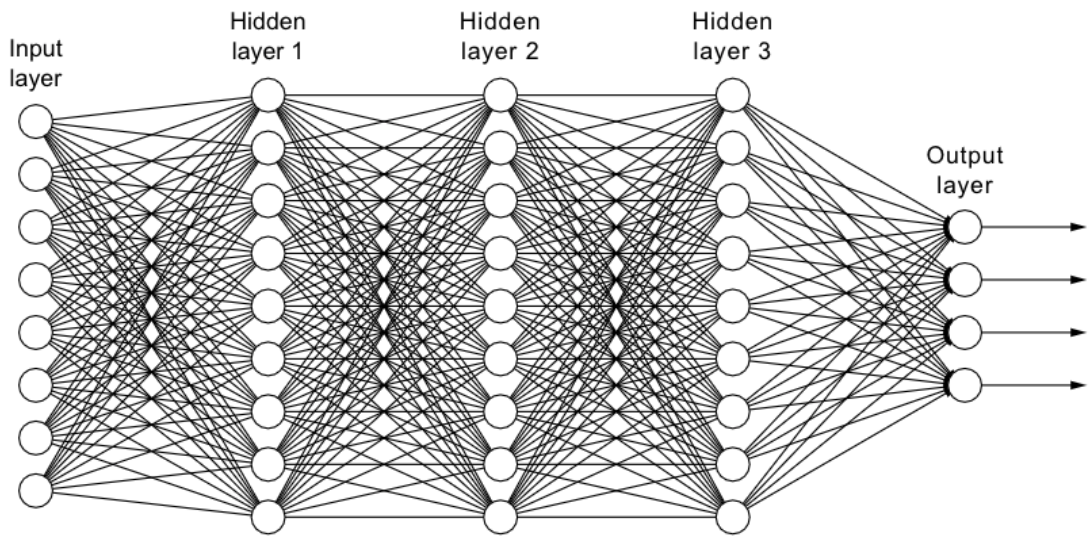


Figure 2.5: An example structure of a fully-connected neural network, composed by 5 layers densely connected between each other.

layers, where each neuron of a layer is connected to all neurons of the next layer. These models are called fully-connected neural networks (FCNN) and can extract the relevant features from the data thanks to the progressive processing made by the layers stack. An example of FCNN model is provided in Fig. 2.5. Hidden layers are simply neuron layers which outputs are only used internally to the network, exploiting the work made by the previous neurons to further process the information. These layers are responsible for the extraction of the features, which are used by the final output layer to produce the results.

Convolutional neural networks

Convolutional neural network (CNN) is another type of ANN, exploited mainly for tasks related to images and videos processing. These neural architectures are characterized by the usage of *2D convolution*, i.e. a linear transformation applying a *kernel* (also called *filter*) to a neighborhood of pixels, which allows them to efficiently extrapolate spatial dependencies. Fig. 2.6 shows an example of how the convolution operation is performed. Both the input and the filters used in convolutions are tensors of 3 dimensions, i.e width x height x channels, referred to as *volumes*. The convolution acts as an element-wise multiplication between a filter-sized patch of the input and the filter weights, which is then summed to return the output value associated to that neighborhood. The filter is applied multiple times, each one on a different section of the input, producing an output volume. The *stride* hyperparameter indicates the height and width movement between each application of the filter on the input volume. Multiple filters can be applied in each convolutional layer, and each one is responsible for producing a different channel

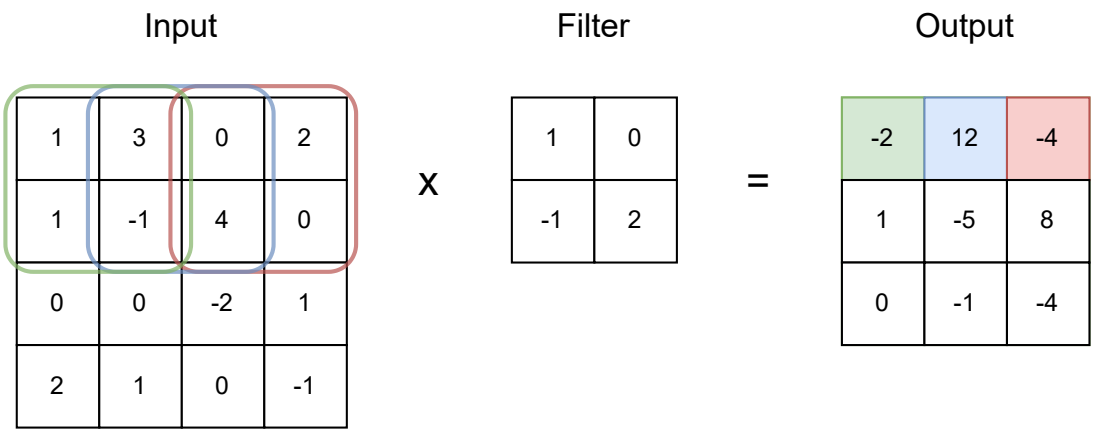


Figure 2.6: An example of convolution operation with stride (1, 1) using a 2x2x1 filter on a 4x4x1 input. The color boxes identify the pixel neighborhoods on which the filter is applied to produce the first row of the output volume.

of the convolution output; the only constraint is that they must have the same amount of channels of the input. All the input volume share the same filter weights, drastically reducing the amounts of parameters compared to FCNN models.

Convolution allows the ANN to study patterns among the pixel neighborhood, extracting relevant feature from the correlation of their values. Each value of the output volume is associated to a portion of the initial image, which size is commonly referred to as *receptive field*. The receptive field increases progressively through the network, allowing the CNN to extract low-level features in the starting layers, which are processed by following convolutional layers to correlate these feature in high-level ones, capturing details from a vast portion of the image.

A common pattern in CNN is to reduce periodically the spatial dimension of the volumes to increase the amount of filters, keeping a similar hidden state size while augmenting the depth of the network, as seen in Fig. 2.7. Spatial dimension is reduced by pooling layers, operations that can process neighborhood of pixels for each channel and output a single value. They are therefore similar to convolutions, but pooling operators actually have a deterministic behavior, since they do not have learnable parameters. Common implementations of pooling layers are max pooling, which outputs the maximum value of the considered neighborhood, and average pooling, which outputs the average of all pixels inside the neighborhood.

Considering a CNN for image classification, like the example provided in Fig. 2.7, the output is often computed by flattening the volume of the last convolution, processing the features with a few fully-connected layers before applying Softmax to compute the

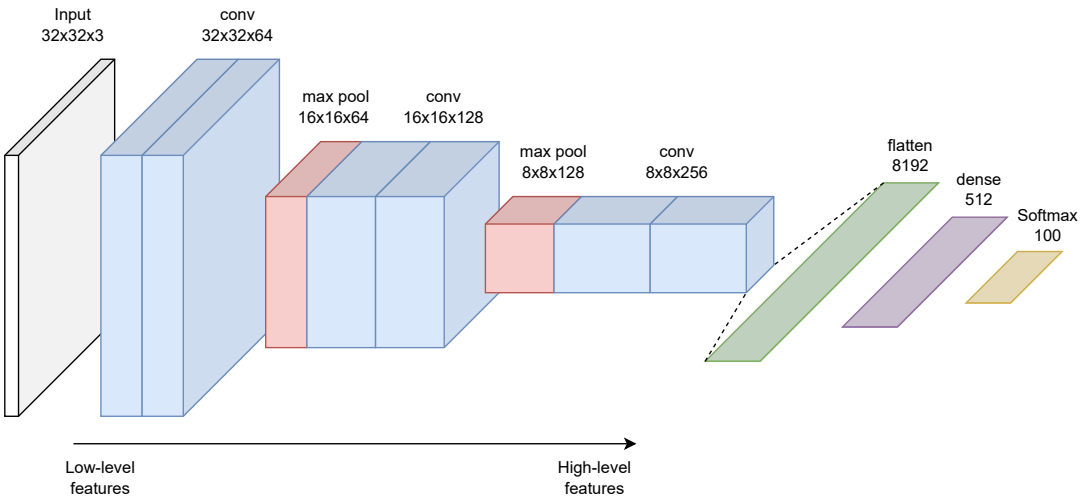


Figure 2.7: An example of CNN architecture for image classification, illustrated with the volume representation. The spatial dimension is reduced with max pooling, while the filters of the convolutions are progressively increased. The volume of the last convolution is flattened into a vector of features, that is processed by a dense layer and Softmax to produce the final output.

final output. Another popular technique is global average pooling (GAP), an operation inserted after the last convolution. This operator outputs the average of each channel of a volume, producing a flat tensor which can be followed directly by the Softmax layer. GAP reduces drastically the number of parameters of the network, but usually does not decrease the final accuracy.

Recurrent neural networks

Previously presented ANN topologies considered "static" datasets, where the output depends only on the provided input, without temporal dependencies among the samples. Other tasks, like text processing and time series forecasting, requires an ANN to address sequences of inputs and outputs, whose values are correlated and must be processed considering their order.

Sequence modeling is often addressed with recurrent neural networks (RNN), models capable of retaining a *memory* of previous inputs, which is exploited for learning temporal patterns. Adding the memory to the hidden state extrapolated from the current processed input enable complex non-linear dynamics, which have been proved to be Turing complete.

The recurrent connections unfortunately introduce new training problems, because they violate the conditions to apply backpropagation and would cause an infinite chain of

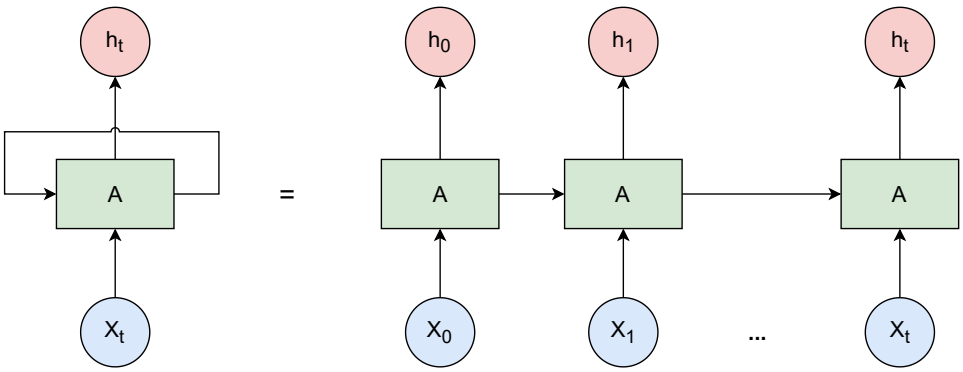


Figure 2.8: Illustration of how a recurrent neural network can be unrolled through a finite amount of time steps, enabling the use of backpropagation techniques.

partial derivatives. The solution is to unroll the network for a finite amount of steps (see Fig. 2.8), enabling the training algorithm to accumulate the errors at each time step and update the weights with a technique called *backpropagation through time* [35].

Long short-term memory [11] (LSTM) is a popular architecture introduced for building RNN. Each LSTM unit is similar to a memory cell composed by multiple logical gates. These cells can be stacked to build a RNN model, where each unit represent a time step. The LSTM structure has been designed to allow an efficient propagation of the gradient through the units stack, minimizing the vanishing gradient effect that could occur when considering a significant amount of time steps.

3 | State of the art

Neural architecture search (NAS) is a subfield of automated machine learning (AutoML), which studies techniques for automating the design of artificial neural network architectures. Architecture engineering is a complex and time-consuming activity, requiring many trial-and-error iterations to define a promising network structure, making it valuable for the considered task.

The previous chapter discussed how artificial neural networks outclass other machine learning methods for tasks in which the features are hard to define, thanks to their capability of extracting patterns and information automatically with a data-driven process Similarly, automating the architecture engineering step can provide significant improvements over hand-designed models, especially in tasks where the accuracy is still not optimal for real-world applications.

Deep learning literature provides architectural patterns which empirically perform well in certain conditions, but it is usually hard to define mathematical rules that fully explain these behaviors. Given those considerations, it is hard for human designers to guide how ANNs should learn patterns from data, by defining a complex structure of nested layers and operations.

NAS algorithms can provide the solution to inefficiencies of human-designed networks, by efficiently searching optimal architectures in a defined search space. These methods are not biased by human knowledge, if the designer does not significantly limit its search space, and can provide additional knowledge on the optimization of neural network models.

NAS techniques have recently gained lots of attention, thanks to the work of Zoph *et al.* [41] (2017), which achieved state-of-the-art results on CIFAR-10, a reference dataset for image classification problems, and the Penn Treebank dataset, used frequently as benchmark for text processing tasks.

The research community recognized the potentiality of NAS [5]. Many other works have been focused on NAS, applying a multitude of different techniques known in the literature of deep learning and machine learning, trying to further optimize the quality of the

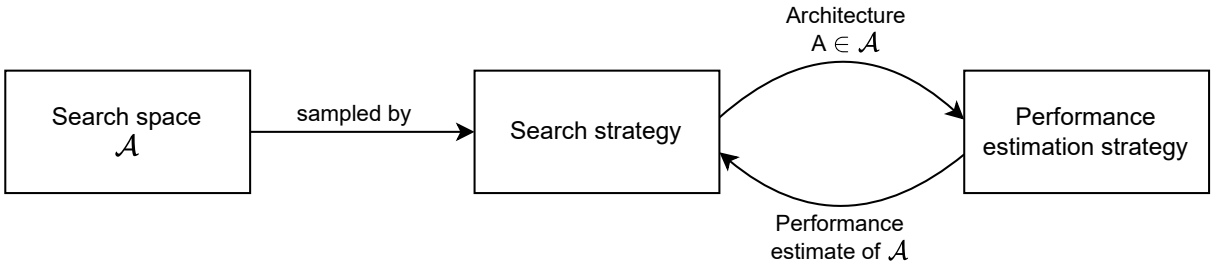


Figure 3.1: Overview of the workflow used in NAS methods. Search space $\mathcal{A}$ is defined by the designer. Search strategy select architectures from $\mathcal{A}$, that are then evaluated with a performance estimation strategy, estimating their accuracy on unseen data.

networks found and to reduce the search time. The most focus is on image classification problems, while only a few papers also focus on other problems, like recurrent neural networks for text analysis or time series.

Recently, NAS algorithms started to consider the architecture search no more as a single-objective optimization problem, focused only on obtaining the best accuracy possible, but as a multi-objective optimization problem. Some works focused their effort in reducing as much as possible the search time required to produce the architectures, while others adapted the search strategy based on various additional constraints, like the latency and the memory required by the final architectures.

3.1. NAS components

NAS algorithms can be extremely heterogeneous, but they are all characterized by three main components:

- *search space*, which defines the architectures that can be searched by the algorithm.
- *search strategy*, which provides the algorithm used to explore the search space.
- *performance estimation strategy*, used to forecast the quality of the networks, to guide the exploration towards the most proficient architectures.

These three dimensions are interconnected to perform the search, forming a common workflow shown in Fig. 3.1. The next subsections describe how each of these components can be implemented.

3.1.1. Search space

The search space defines the space of all the possible architectures that can be searched by the NAS algorithm. This is commonly defined by the algorithm designer, through a set of operators that is considered well-suited for the task addressed by the output architectures, plus the possible choices to combine them. Defining a priori a set of operators reduce the size of the search space, simplifying the search, but also introduce a human bias in the algorithm. Embedding knowledge in the search space is beneficial, because operations chosen by designers are often extensively used in already existing neural network architectures that achieve top performances. Anyway, this can also be seen as a limitation, since the algorithms can not find new building block not currently discovered in the literature, especially for problems where existing architectures still do not perform well enough.

Hand-crafted architectures usually find a performing combination of operators and inputs, forming a "building block" that is then repeated multiple times across the network to build the actual architectures. This building block is usually referred in the literature as *motif*. Recent NAS techniques embrace the motif repetition pattern, focusing on the search of good *cells* to be repeated inside the architecture, instead of searching for the architecture specification as a whole.

The search space can be divided in two categories: the *micro-architecture*, defining the possible choices for cell specifications, and the *macro-architecture*, defining how the cells can be combined to build the final architecture. Fig. 3.2 illustrates how the final architectures can be built after defining the motifs and the rules to stack them.

Cells are composed of multiple units called *blocks*. A block is specified as a 5-elements tuple $(i_1, o_1, i_2, o_2, o_{join})$, where i_1 is the input of the operator o_1 and i_2 is the input of the operator o_2. The outputs of o_1 and o_2 are then joined with o_{join}, producing the final output of the block. The set of allowed operators and inputs are defined a-priori by the algorithm designer.

A cell can be considered as a directed acyclic graph (DAG), where blocks are interconnected in a multi-branch fashion, receiving inputs from previous cells or other blocks. Many NAS methods define two different types of cells: normal cells, that preserve the dimensionality of the input, and reduction cells, which reduce the spatial dimension of the output compared to the input received, but also increase the number of filters.

Defined the cell structures, a common approach to build the final architecture is to connect them in a single path chain-structured network, where multiple normal cells and

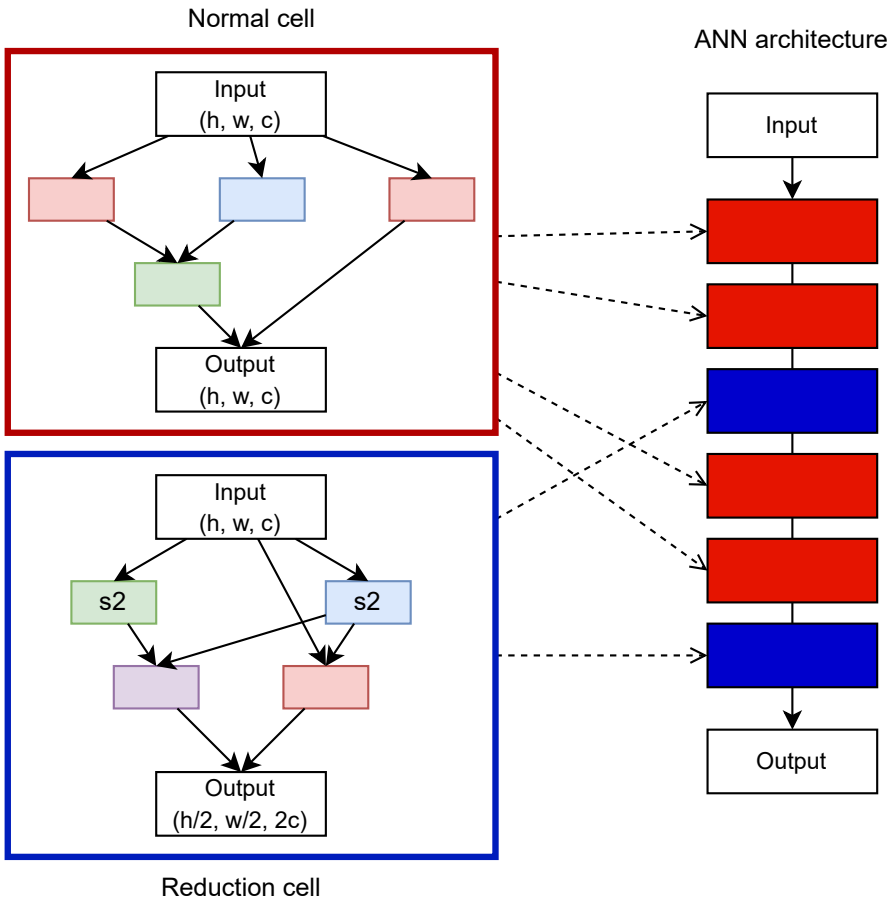


Figure 3.2: On the left are represented a normal cell (top) and a reduction cell (bottom). Micro-search focus on finding the best structure for these two cells. These cells are then stacked multiple times to define the neural network architecture, represented on the right, according to macro-search results.

single reduction cells are interleaved. Recent NAS works also incorporate modern design elements known from hand-crafted architectures such as skip connections, which allow to build complex, multi-branch networks.

Searching for cell motifs effectively restrict the search space cardinality, compared to searching for a whole large architecture composed of potentially hundreds of layers. Moreover, searching a cell allows to tune the architecture effectively also for similar tasks and datasets. In fact, cells can be transferred more easily to other datasets by adapting the number of cells stacked within a model, along with the number of filters used by operators. This concept has been demonstrated by the experiments performed in NASNet [42] and PNAS [18] works, where cells found on CIFAR10 have been ported to ImageNet, achieving state-of-the-art performances on both datasets.

Given these benefits, most of the recent NAS works define their architecture through the usage of cell motifs. Anyway, the search space remains too large to be completely explored, since often NAS works define search spaces that includes more than 10^{10} motifs. The main challenge of any NAS algorithm is therefore to effectively search into huge search spaces, finding top quality architectures with the least amount of GPU hours. *Search strategy* and *performance evaluation strategy* define how the algorithm explore the search space and can be considered the most important components of NAS, while search space is adaptable for the task.

3.1.2. Search strategy

The search strategy defines the algorithmic steps for exploring the search space. Zoph *et al.* [41] used reinforcement learning as search strategy, but many solutions have been employed in other NAS works. Most of the search strategies described in the literature can be organized in the following categories:

- Random search
- Reinforcement learning (RL)
- Evolutionary algorithms (EA)
- Gradient-based
- Sequential model-based optimization (SMBO)
- Bayesian optimization (BO)

Since randomly exploring vast search spaces is inconsistent for finding top architectures, random search is just used as a benchmark baseline to estimate the quality of more complex search strategies. Good search strategies are expected to adapt to the defined search space and to the tasks provided, converging to the space of optimal architecture. The whole process is aided by the performance estimation strategy, which is described in Section 3.1.3.

Reinforcement learning

NAS task can be reformulated as a standard reinforcement learning (RL) problem. The action space is equivalent to the search space, so the choice of an architecture is the agent's action. The reward function is the estimated accuracy on unseen data of the chosen architecture. The main differences in works exploiting reinforcement learning regards the agent used and the policy to train it.

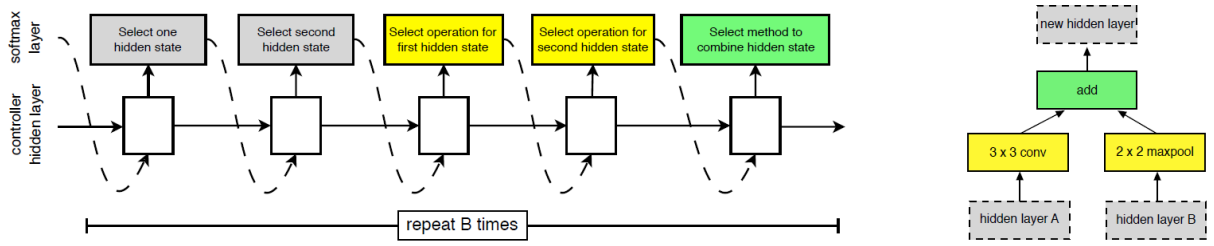


Figure 3.3: The LSTM controller model used by NASNet. Each prediction made by the LSTM represent an element of a specific block, chosen as the maximum argument of a Softmax layer. By repeating the predictions, multiple blocks can be defined to form both the normal cell and the reduction cell structures. On the right, it is illustrated an example of block structure defined through the LSTM predictions.

NASNet [42] is the most popular example of NAS work adopting reinforcement learning. NASNet searches for separate structures for normal and reduction cells, which are produced by the agent of the RL problem. The agent deployed is a recurrent neural network based on LSTM [11], referred as controller by the authors, which is used to output the block specifications for the cells. The controller weights are updated through Proximal Policy Optimization [32], scaling the gradients with the validation accuracy of the trained network.

The cells searched in NASNet are composed by 5 blocks, where each block is defined through a 5-elements tuple. The controller makes $2 \cdot 5B$ prediction, where the first 5 blocks refer to the normal cell, while the last 5 blocks refer to the reduction cell. Each prediction made by the controller is actually a Softmax layer; the argument corresponding to the maximum output of Softmax define the element used in the block specification. The Softmax output of each step is used as input for the next prediction, which of course also considers the hidden state of the LSTM. The mechanism is illustrated in Fig. 3.3.

Evolutionary algorithms

Evolutionary algorithms use neuro-evolutionary approaches for searching optimal neural network architectures. Even if old works in the literature tried neuro-evolutionary techniques for adapting the weights, nowadays gradient-based methods outclassed them for weight optimization. Evolutionary algorithms therefore exploit neuro-evolutionary methods only for manipulating the motifs (or networks) specifications, starting from a random population.

In each step, the current population of networks is altered through mutations and crossover operations, generating an offspring. Mutations are local operations that modify a motif

encoding, for example by adding or removing layers, adding skip connections and changing hyperparameters. The possible mutations are defined by the algorithm and they are applied in a stochastic way to the selected encodings. Crossover instead consider different encodings and mix them together to produce new architectures. The resulting offspring is trained and tested on a validation set, then the population is updated with the most promising individuals.

An example of evolutionary algorithm is given by AmoebaNet [29]. The search space is based on NASNet one, and the algorithm searches for normal and reduction cell specifications. Cells are represented as binary encodings, where bits can refer to pairwise inputs or operators, with 0 and 1 representing the two possible values. Since each bit represent a choice between only two elements defined by the authors, the search space is actually a subset of NASNet.

The population is initialized with random architectures encodings, which are trained and evaluated on data without further modifications. The algorithm then proceeds for C cycles. In each step, a subset of networks is randomly sampled from the current population and the model with the highest accuracy is selected as parent. The parent is mutated, changing random bits in the encoding to generate a single child. The child architecture is trained and evaluated, then it is inserted in the population, replacing the oldest element of the population. This process is referred in the literature as tournament selection [7], using aging evolution since the population is updated through age criteria.

Sequential model-based optimization

In Sequential model-based optimization (SMBO) [13], NAS is addressed as a sequential process. Starting from the simplest models of the search space, the motifs are progressively expanded with new layers, until a target level of complexity is reached. At each step, the expanded motifs are evaluated on a validation set and are then used as baseline for further expansions. SMBO algorithms usually select the most promising expansions through the estimations made by auxiliary models.

PNAS [18] is the most cited algorithm employing SMBO as search strategy. PNAS search space is inspired by NASNet, defining normal and reduction cells as aggregation of multiple blocks. In this work, normal and reduction cells share the same structure, using a (2, 2) stride in some reduction cell operators to reduce spatial dimensionality.

Initially, all cell specifications composed of one block are trained and evaluated. The algorithm is composed of B steps, where B is the target amount of blocks that final cells will contain. The cells are progressively expanded by adding one block in each step.

A surrogate model, referred as predictor, is trained on the cell results and it is used to predict the quality of the possible expansions. Only the most promising cell expansions are selected for training, while the others are discarded.

Gradient-based

The search space defined by NAS works is commonly discrete, since it embeds a finite (huge) number of cell or network specifications to be searched. Gradient-based methods relax the constraints of the search space to make it continuous and differentiable, allowing the application of gradient-based techniques to optimize the architecture structure. This type of search strategy makes therefore possible to optimize both architecture weights and structure using similar techniques, simplifying the NAS algorithm. In fact, compared to other search strategies, gradient-based optimization does not require additional surrogate models, making it easier to adapt for different tasks.

Gradient-based search strategy was introduced by DARTS [19] and it has been a very popular choice for NAS algorithms since its introduction. DARTS search space is based on cell motifs, organized as direct acyclic graphs of N nodes, representing hidden layers of the neural architecture. Between each pair of nodes there are multiple edges, each one representing a different operation of the search space plus the none operator. Continuous values are attributed to each edge, using Softmax formula. The k-top operators are chosen to discretize the search space and build the final architectures. Using this relaxation, DARTS train both the weights and the network architecture using gradient descent. The algorithm use as metric the validation loss for optimizing the architecture structure and the training loss to optimize the weights, addressing a bilevel optimization problem.

Bayesian optimization

Bayesian optimization [33] is a popular technique exploited by hyperparameter tuning tools for neural networks. Anyway, this method has not been applied frequently in NAS algorithms, because NAS search space is high-dimensional and discontinuous, while this technique is more suited for low-dimensional continuous optimization problems.

Bayesian optimization methods starts from a prior Gaussian distribution of the objective function f, which is updated at each iteration in a new Gaussian distribution, called posterior. The distribution update is based on the evaluation of the sample extracted from the distribution, also keeping track of the confidence interval of the estimations to enable an exploration-exploitation trade-off. Bayesian optimization is mainly used in cases where the knowledge about the true objective function f is limited, or it is simply

significantly expensive to compute. Bayesian methods are characterized by two main elements:

- a surrogate model $\hat{f}$, used to approximate f.
- an acquisition function computed from the surrogate model, exploited for sampling significant evaluation points at each iteration.

The iterative approach evaluates N samples, making possible to fit a valid surrogate model $\hat{f}$ to approximate f. The surrogate model can be implemented with different techniques, such as polynomial interpolation, support vector machines and Gaussian processes. The acquisition function can also be defined in multiple ways, like Maximum Probability of Improvement (MPI), Expected Improvement (EI) and Upper Confidence Bound (UCB).

In NAS methods, Bayesian optimization aims to find the architecture belonging to the studied search space that has the minimal validation error. Considering $\mathcal{A}$, i.e. the NAS search space, and any architecture $a \in \mathcal{A}$ belonging to the search space, the goal is to find the best architecture $a^* = \text{argmin}_{a \in A} f(a)$, where $f(a)$ is the validation error reached by an architecture a after training. The training results of each architecture sampled from the search space are used to update the surrogate model $\hat{f}(a)$.

BANANAS [36] is a recent promising NAS work using Bayesian optimization search strategy. It implements the surrogate model as an ensemble of feedforward neural networks, using the multiple predictions to model the mean and variance of any architecture of the search space. The chosen acquisition function is independent Thompson sampling (ITS). The search is initialized by uniformly sampling a fixed amount of architectures from the search space, training them on the datasets. The predictor ensemble is trained on the results. The best architectures previously sampled are then mutated and the new candidate architectures are analyzed with the acquisition function, selecting a single network to train and evaluate for the next step. The process repeats for N steps, afterwards the algorithm outputs $a^* = \text{argmin}_{n=0,\dots,N} f(a_n)$ as best architecture.

3.1.3. Performance estimation strategy

Performance estimation is a necessary step to guide the search strategy into finding the most promising architectures in the considered search space. In fact, neural architectures selected by the search strategy can not be trained to convergence to provide an accurate validation error, because it would require an immense GPU time. Considering that the search spaces cardinality of the NAS algorithms can be in order of 10^{10} architectures or more, is common to sample at least hundreds of networks to reliably find good architec-

tures.

The goal of the estimation phase is to evaluate the quality of a selected architecture, without the need of a complete training. This step enables the search strategy to rank the architectures quality, converging to search space areas which contains good motifs, discarding instead suboptimal ones.

Various techniques exist for implementing a valid performance estimation strategy, providing shallower results in a shorter amount of time compared to a training to convergence. The most common strategies used in NAS works are:

- Low-fidelity estimation
- Learning curve extrapolation
- Weight inheritance
- One-shot model (weight sharing)

Low-fidelity estimates

Some strategies simply focus on providing lower fidelity estimates, compared to an extensive network training on the entire data available. Speedups are achievable by using different technique during training, for example:

- reducing the resolution of the images in training set
- training on a subset of the original dataset
- training for a small fixed amount of epochs
- reducing the number of filters used in each layer
- reducing the actual number of motifs stacked in the network architecture

Even if the computational cost is drastically reduced, this approach introduces bias towards smaller and simpler architectures, because they are easier to train and therefore provide good results even with a short training. There is study evidence in the literature that the ranking of the architectures can change significantly because of this intrinsic bias (see Zela *et al.* [40]), making this method not well-suited for general use. The ranking bias is proportional to the difference of epochs between the lower fidelity estimate training and the actual epochs required for a training to convergence. Given this consideration, it is important to balance how much shallower the performance estimate should be, reaching a trade-off between fidelity and time speedup.

Learning curve extrapolation

Learning curve extrapolation methods study the initial learning curve shown after training a neural network for few epochs. These techniques allow the search algorithm to terminate the training of the architectures which are predicted to perform poorly, with a negligible time cost, speeding up the search.

A simple way to implement learning curve extrapolation is fitting a curve with the accuracies found during a limited amount of epochs, providing a noisy forecast of the accuracy that the neural network can reach in future epochs. Another way to perform this strategy is to analyze the gradients with mathematical techniques, estimating the most promising architectures from the information extracted.

Weight inheritance

Weight inheritance speed-ups the performance estimation of an architecture by initializing its weights with the ones of another compatible architecture that was previously trained. Warm starting the weights of the networks, when possible, allows them to reach near convergence accuracy with fewer training steps, speeding up the NAS algorithm.

This technique is mainly employed in evolutionary search strategies. Offspring networks share a similar structure with their parents, since they are generated through mutations of their encodings. Some mutations do not change layers and their shape, making possible to inherit the weights of the parent without further manipulation. For cases where mutations change the layers of the parents, weight inheritance could still be possible but requires some additional processing, like padding or reshaping of the parent weights.

One-shot model

Using a *one-shot model*, also referred to as *supernet*, is a very efficient approach for performing the performance estimation strategy. After defining the search space, it is possible to build and train a single huge neural network (the one-shot model), which embeds all architectures of the search space. Each architecture is a single path of the supernet, taken from input to output. When the NAS algorithm samples an architecture of the search space, the paths of the supernet not involved in the chosen architecture are pruned, making possible to use directly the weights learned by the supernet to evaluate the candidate quality.

After the initial resource investment for training and storing the supernet, using a one-shot model makes possible to perform the search process very efficiently thanks to the

weight sharing mechanism, which makes the performance estimation step trivial, enabling major speedups in the search time. Training a supernet is costly, but the one-shot model size scales linearly with the choices introduced in the search space, while the search space cardinality is instead exponential to them. This property, in the long run, makes training a supernet less costly than training from scratch the multiple networks sampled by the search strategy.

Even if this performance evaluation strategy is extremely efficient, it is by far the most complex performance evaluation strategy to use in NAS works. "Understanding and Simplifying One-Shot Architecture Search" [1] work provides a comprehensive analysis of one-shot architecture and a baseline on how to train them properly.

The main problem is related to co-adaptation of model weights. Training naively the supernet leads to weights co-adapting during training, causing high accuracy drops when considering a single path during the model predictions. This behavior is due to the paths learning weights that works well only when all paths are used together, but actually perform poorly when considered individually. To discover good architectures during the search, zeroing out non-performing operations must have a minor impact, instead dropping well performing operations should lead to significant drops in accuracy. Incorporating path dropout during training mitigates this issue, allowing weights to train more independently, making well-suited operators perform better without involving other subpaths.

Another problem is the high amount of memory required to perform this technique. The whole supernet must reside on the GPU memory for training and for evaluating the subpaths during the search, therefore the search space size achievable with one-shot models is limited by the available GPU memory.

Moreover, the results reached by evaluating the subgraphs of the one-shot model are significantly lower than the accuracy reachable after an exhaustive training. The performance gap is caused by the heavy use of dropout and regularization technique, which is required to avoid weight co-adaptation but also ruins the general performance. Even the quality ranking of the architectures is not guaranteed to be reliable and could be subject to noise, as demonstrated by Yu *et al.* [39]. Training the architectures for a few epochs is a possible solution for improving the accuracy of the estimations, initializing the weights with the ones contained in the supernet. Still, this additional step partially negates the advantages provided by the one-shot model approach, making it undesirable.

3.2. NAS benchmarks

Neural architecture search demands a tremendous amount of computational resources, which makes it difficult to reproduce experiments and imposes a barrier-to-entry to researchers without access to large-scale computation assets. In fact, oldest NAS algorithms require thousands of hours of GPU time to perform the search space. Furthermore, NAS methods are usually not comparable to each other due to different training procedures and different search spaces, which make it difficult to attribute the success of each method to the search algorithm itself.

The goal of NAS benchmarks is to provide a standard way for testing different NAS algorithms and compare them, without the necessity of using huge amount of resources. This is made possible by reducing the search space to an acceptable amount of architecture, train all of them with the same procedure and hyperparameter set, and finally store the accuracy results and relevant training metrics into a database. Dataset records can be queried to immediately get the required characteristic for the comparisons, instead of performing a train and evaluate procedure, which is the most costly step of all NAS methods. This enables researchers to test a search method using the data contained in the NAS benchmarks, eliminating the necessity of GPU usage.

The limits of NAS benchmarks are clearly the extremely limited achievable size for a search space, since all networks included must be trained and evaluated. Since each choice added to the search space increasing exponentially the number of architectures included in it, the GPU hours required for the complete training of the search space also increase exponentially. This drawback makes tabular NAS benchmark unfeasible for the search spaces commonly used in literature works, which can contain over 10^{10} architectures. Moreover, large search spaces make difficult to consider also different datasets, since training the architectures on multiple datasets further increase the cumulative GPU hours required to build the NAS benchmark.

3.2.1. NAS-Bench-101

The first work in this field is NAS-Bench-101 [38]. NAS-Bench-101 provides an exhaustive evaluation of all architectures contained in a cell-structured space, trained only on CIFAR-10 dataset. Most NAS approaches to date have trained models on the CIFAR-10 classification set because its small images allow relatively fast neural network training. Furthermore, models which perform well on CIFAR-10 tend to perform well on harder benchmarks, such as ImageNet, as demonstrated in some relevant NAS works [18, 42].

Table 3.1: Hyperparameter set used in NAS-bench-101

batch size	256
initial convolution filters	128
learning rate schedule	cosine decay
initial learning rate	0.2
ending learning rate	0.0
optimizer	RMSProp
momentum	0.9
L2 weight decay	$1e^{-4}$
batch normalization momentum	0.997
batch normalization epsilon	$1e^{-5}$
accelerator	TPU v2 chip

Each cell is represented as a directed acyclic graph, where the nodes represent operation choices and the edges simply indicates information flow through the neural network. To limit the number of architectures in the search space, the authors used the following constraints on the cell:

- choice between 3 operations: 3x3 convolution, 1x1 convolution, 3x3 max-pool. The operations are followed by batch normalization and ReLU activation.
- at most 7 nodes (this includes input and output node, therefore at most 5 operation nodes)
- at most 9 edges

Removing equivalent cell encodings, the search space contains about 423k unique architectures.

The architectures are built by composing multiple motifs. A 3x3 convolution *stem* is applied to the input to initialize the network. *Stacks* are compositions of 3 cells, while downsample layers are max pooling layers halving spatial dimension and doubling the filters. A complete architecture is constituted by the stem, followed by stack + downsample repeated two times, a final stack, which is finally connected to a global average pooling and dense layer to produce the final output.

A single, fixed set of hyperparameters is used to train all NAS-Bench-101 models. Standard data augmentation techniques are also employed (see He *et al.* [9]). This set of hyperparameters, illustrated in Tab. 3.1, was chosen to be robust across different architectures.

The training and evaluation of all architectures is repeated 3 times to obtain a measure of

variance, using 4 increasing epoch budgets $E_{stop} = \{4, 12, 36, 108\}$. As result, the obtained dataset is a mapping from $(A, E_{stop}, trial_{\#})$ to the following quantities:

- training accuracy
- validation accuracy
- testing accuracy
- training time in seconds
- number of trainable model parameters

Over 100 TPU years of computation times have been used to train all these architectures.

3.2.2. NAS-bench-201

NAS-bench-201 [4] is a more recent work, that consider itself as an extension to the NAS-bench-101 dataset. NAS-bench-201 architectures has been trained on 3 different datasets, namely: CIFAR-10, CIFAR-100 and ImageNet-16-120 [3]. The goal of NAS-bench-201 is to be directly applicable to almost any existent NAS algorithm, as some of them could not work properly on NAS-bench-101, for example methods using parameter sharing (one-shot) or network morphism. NAS-Bench-201 solves this problem by further reducing the search space size, but including all possible edges in cells to be algorithm-agnostic and providing extra diagnostic information, required for generalizing on different performance evaluation strategies.

The search space includes all possible architectures generated by 4 nodes and 5 associated operation options, without additional constraints, which results in a total of 15.625 neural cells candidates. The set of operations chosen by NAS-Bench-201 is composed by: zeroize, skip-connection, 1x1 convolution, 3x3 convolution and 3x3 average pooling layer.

Regarding the training information, NAS-Bench-201 provides more fine-grained accuracy and loss values compared to NAS-Bench-101, computing them after each epoch. Moreover, the trained weights are also saved for each network, making possible to apply weight inheritance techniques. The number of parameters, FLOPS and the latency are also provided, allowing algorithms to consider resource constraint and optimize the search for multi-objective optimization problems. A comparison between the two NAS benchmarks is provided in Tab. 3.2, providing also a summary of the supported NAS techniques.

Table 3.2: Summary of the main differences between NAS-Bench-101 and NAS-Bench-201, regarding search space and supported NAS techniques (RL = reinforcement learning, ES = evolutionary strategy, GB = gradient-based, HPO = hyperparameter optimization).

Bench method	# architecture	# datasets	operator set size	Supported NAS algorithms			
				RL	ES	GB	HPO
NAS-Bench-101	423K	1	3	partial	partial	none	most
NAS-Bench-201	15.6K	3	5	all	all	all	most

3.3. PNAS

Progressive Neural Architecture Search (PNAS) [18] is an efficient search algorithm, based on sequential model optimization (SMBO) strategy. Its search strategy is capable of finding state-of-the-art architectures while drastically reducing the search time required, compared to previous methods.

Contrary to many other works, PNAS works by sequentially building the models, progressively expanding the models trained with new blocks, until the desired number of blocks per cell is reached. The expansion process is guided by a surrogate model, a neural network model that predicts the accuracy of unexplored solutions. Using the accuracy estimations, the search algorithm can choose the most promising cells among all possible expansions. The selected architectures are built and trained in the next step, afterwards they are expanded again, repeating the process until the cells are expanded to the target amount of blocks. This approach has been proved to provide state-of-the-art networks on CIFAR10 and ImageNet [31], with a reduction of the search time of a factor 8 compared to NASNet [42], the reinforcement learning NAS method on which PNAS is based upon.

3.3.1. Search space

The networks generated are convolutional neural networks (CNN) built by stacking multiple cells together, concluding the network with a GAP followed by Softmax to get the final results. Each cell is composed by multiple blocks. Each block is identified as 4-tuple (i_1, i_2, o_1, o_2), since addition is the only join operator defined by PNAS and it is therefore implicit in the block encoding. The operator space $\mathcal{O}$ is the following:

- identity
- 3x3 depthwise-separable convolution
- 5x5 depthwise-separable convolution
- 7x7 depthwise-separable convolution
- 1x7 followed by 7x1 convolution
- 3x3 dilated convolution
- 3x3 max pooling
- 3x3 average pooling

This operator space is inspired by the most used operators in the state-of-the-art handcrafted models and by the operator search space of NASNet, exploiting its results to restrict the search space to the most relevant operators, boosting the search time efficiency.

The input space varies based on the search step the algorithm is currently performing. Initially, when the algorithm is exploring the cells having only one single block, only the lookback inputs can be chosen. The lookback inputs are the outputs of previous cells (or the initial image) and in PNAS are limited to the set $\mathcal{I}_1 = \{-1, -2\}$, where -1 is the output of the previous cell and -2 is instead the output of the second to last cell (skip connection). When the number of blocks is greater than 1, the n-th block of a cell can use as input the output of any previous block of the cell specification. The blocks inside a cell are enumerated with an index starting by 0. The input space values ≥ 0 specifies that the block of that index is used as hidden layer in another block (e.g. block 0 output is associated to input value 0, which can be used in any block with index > 0). The set of inputs grows therefore with the number of blocks b and has cardinality $|\mathcal{I}_b| = 2 + b - 1$.

There are two types of cells in each CNN: normal cells and reduction cells. In PNAS they have the same structure, simplifying the whole search procedure, since only one cell specification is searched instead of finding good combinations of normal and reduction cells for each network. Here, the only structural difference between normal and reduction cells is that the reduction cells use a stride (2, 2) in the operations using any lookback as input. Applying a stride 2 allows the network to halve the spatial dimensions of the tensors, so that the filters can be doubled without increasing the hidden layers size. This pattern is frequently used in many handcrafted CNN models.

Each model trained during the search phase is a convolutional neural network composed by a stack of 8 cells: 6 normal cells and 2 reduction cells, followed by a GAP and Softmax to get the final results. All models are trained with the same hyperparameter set for a fixed amount of epochs E, which in the paper experiments was set to 20.

The total size of the search space, considering cells composed by 1 to 5 blocks (the experimental search run performed by PNAS) is of about 10^{12} different cells. Even if this size is smaller than other NAS methods like NASNet, it is still significantly large to explore with random search and therefore PNAS exploits a surrogate model to guide the cell expansions.

3.3.2. Search strategy

PNAS method is based on a sequential model optimization (SMBO) search strategy. An initial training phase is performed on the simplest models of the search space, which are the cells composed by a single block. After the training phase, a surrogate model, referred to as predictor, is trained on the validation accuracy obtained by each cell during the training phase. When the predictor is ready, an expansion phase occurs, where the predictor helps in choosing the most promising models for the next step, since it can estimate the accuracy reachable by each candidate. The models considered in the expansion phase are built by adding another block to the cell specifications selected in the previous training phase. The amount of networks picked for the next training phase is limited to the top-K most promising architecture regarding the obtainable accuracy, according to the estimations made by the predictor. K is a parameter of the PNAS method, which is needed to limit the search space to a handful of architecture in each training step, since there are no other constraints for the selection of the networks. This process is repeated until the target amount of blocks inside a cell is reached (5 in PNAS experiments).

3.3.3. Performance estimation strategy

The predictor is trained on the validation accuracy results obtained by each cell during the training phase, so that it can make an informed decision for selecting the most promising expansions for the training step with $b + 1$ blocks.

PNAS authors listed 3 fundamental properties that a predictor should have to be integrated in the search algorithm and produce good results for driving the expansion phase:

- *handle variable-sized inputs*: since the amount of blocks contained in a cell varies during the search, the predictor must be able to handle different input sizes or pad them properly.
- *correlated with true performances*: the most important thing for the predictor is to correctly rank the architectures, and not to predict the exact value of the accuracy.
- *sample efficiency*: since PNAS wants to be as efficient as possible regarding search time, the amount of networks trained is small. This means that the predictor must be designed to learn efficiently from a limited amount of samples.

In PNAS, the predictor receives as input the categorical encoding of both inputs and operators of the cell, returning as output the expected validation accuracy that the cell would reach after the training process. Two different type of models have been tested for

the predictor: an LSTM and a multi-layer perceptron (MLP). Both models use separate embeddings for the input and operator encodings, then the hidden state goes either in the LSTM or in the MLP, both followed by a sigmoid activation for producing the output. The predictor is trained after the first training phase and then progressively fine-tuned with the results obtained in the following steps. This surrogate model is then used to predict all possible expansions of the models trained for the step with b blocks, so that the search algorithm can select the best ones to carry on for future steps, pruning the others.

3.4. POPNAS

POPNAS [20], acronym of Pareto-Optimal Progressive Neural Architecture Search, is a NAS method which enhances the search efficiency of PNAS by considering a time-accuracy Pareto optimization problem. POPNAS adds a new surrogate model, referred to as time predictor, to the progressive approach, making possible to carry out a joint prediction of time and accuracy for each candidate neural network architecture. This change allows POPNAS to limit the search on the Pareto front of the defined multi-objective optimization problem, reaching a trade-off between accuracy and training time. The goal of this search strategy is identifying neural network architectures with competitive accuracy but drastically reduced training time.

3.4.1. Search space

POPNAS search space follows very similar concepts to the one defined by PNAS. A block is still specified by the 4-elements tuple (i_1, o_1, i_2, o_2), but using concatenation as the join operator of the two operations of the block. The considered operator space $\mathcal{O}$ contains the same 8 PNAS operations (listed in Section 3.3.1), but, in POPNAS, the order in which they appear becomes relevant. Thus, an index from 1 up to 8 is associated to each operator, according to the time required to perform it in increasing order:

1. 3x3 average pooling
2. 3x3 max pooling
3. identity
4. 3x3 dilated convolution
5. 3x3 depthwise separable convolution
6. 5x5 depthwise separable convolution
7. 7x7 depthwise separable convolution
8. 1x7 - 7x1 convolutions

Note that identity is transformed into a pointwise convolution inside reduction cells, since the tensor shape must be adapted to the format required by following operators; this is

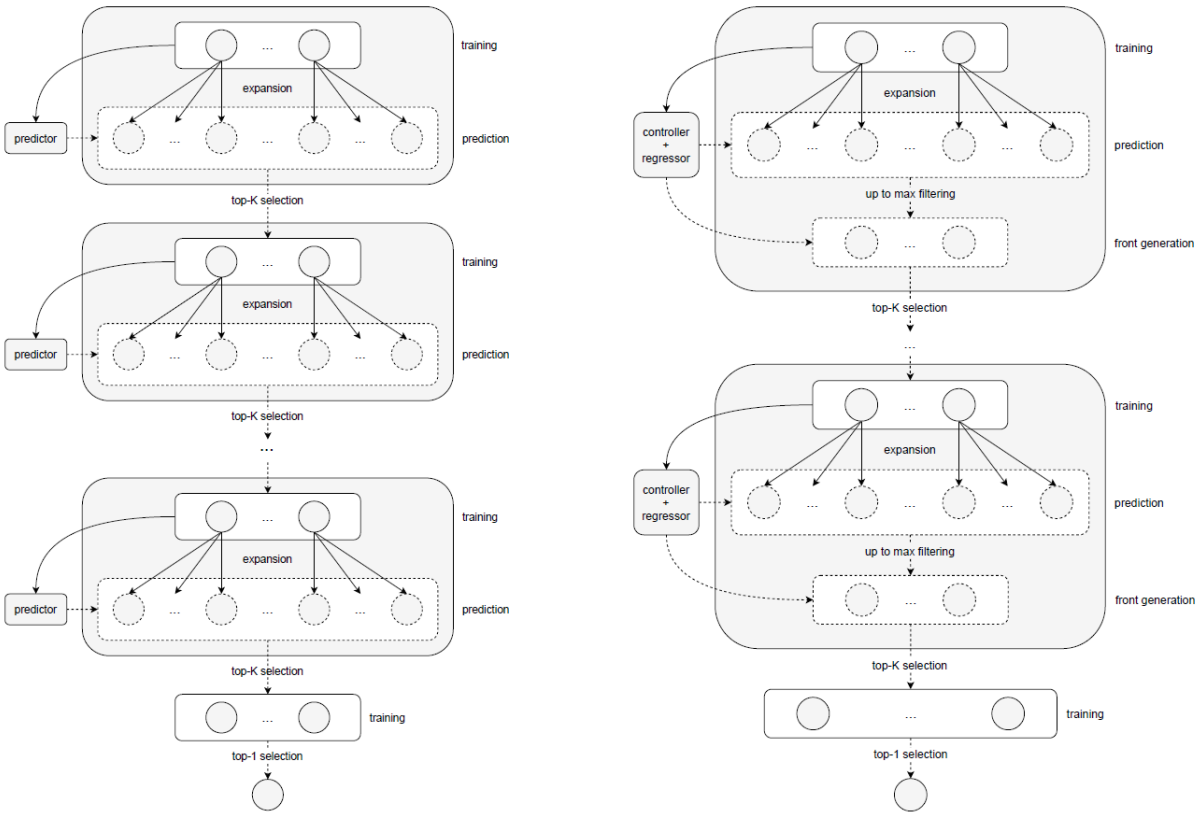


Figure 3.4: A comparison of PNAS (on the left) and POPNAS (on the right) search procedures. POPNAS search procedure expands the one defined by PNAS, adding a new surrogate model, called regressor, to estimate the training times of the cell expansions, enabling this NAS algorithm to build a time-accuracy Pareto front.

why identity appear slower than pooling operators. The input space has been simplified to only the -1 lookback, building flat cells where all blocks are performed in parallel on the input of the previous cell.

3.4.2. Search strategy

The goal of POPNAS search strategy is to find for the most accurate cell structures among those with the lowest training time, pruning out the cells that require more training time but have the same accuracy. Limiting the search on the architectures belonging to the time-accuracy Pareto front further increases the efficiency of the search algorithm, reducing the total amount of GPU hours. Furthermore, it is possible to set a maximum time limit L for the training time of the children networks, so that the algorithm can exclude from the Pareto front selection the cells which are predicted to violate this constraint.

The search strategy resembles the progressive one used by PNAS, expanding it with

additional steps to convert the search into a Pareto-optimized one. A visual comparison of the two search procedures is provided in Fig. 3.4. The search starts with the training of the empty cell, which results are exploited for training the predictors. Then, the algorithm proceeds by training all the architectures of the search space having a single block for cell.

Each CNN generated from a cell specification is built as a stack of 8 cells (6 normal cells and 2 reduction cells), followed by GAP and Softmax. All networks are trained for E epochs to get indicative values on the validation accuracy and the time required for training the network. Each training phase up to cells with $B-1$ blocks is followed by the so-called expansion phase. This phase starts with the training of the two surrogate models used by POPNAS: the *controller* and the *regressor*.

The controller, is trained on the validation accuracies, with the goal of predicting the quality of the cell expansions, similarly to what is done in PNAS. The regressor is instead introduced by POPNAS and it is instead trained on the training time of the sampled architectures, for estimating the total time required to train their expansions on E epochs.

For each possible cell expansion, both the validation accuracy $\hat{a}$ and the training time $\hat{t}$ required for E epochs are predicted. The time-accuracy Pareto front can be built from the estimations of the two surrogate models, allowing POPNAS search strategy to prune the dominated cells. The Pareto optimality increases the algorithm search efficiency, by dropping time-consuming networks which perform worse than others already selected for training.

The Pareto front is built in the following manner. Initially, if the maximum time parameter L is set, the cells having a predicted time $\hat{t} > L$ are discarded, applying the time limit. The highest predicted accuracy cell is inserted as the Pareto front head to initialize the process, then a simple domination rule is applied to build the Pareto front: cell 1, having predicted accuracy $\hat{a_1}$ and predicted training time $\hat{t_1}$, dominates cell 2, having predicted accuracy $\hat{a_2}$ and predicted training time $\hat{t_2}$, if $\hat{t_1} \leq \hat{t_2}$ and $\hat{a_1} > \hat{a_2}$.

The Pareto front is constituted by all the dominant cells among the set of all possible expansions. The cell specifications inserted in the Pareto front are limited up to K elements, which are the cells selected for the next training step. Training and expansion steps are repeated in cycles, until the target number of blocks B is reached.

3.4.3. Performance estimation strategy

Similar to PNAS, the models are trained on a small amount of epochs E, but in this case both the reached validation accuracy and the total training time are relevant metrics,

used for selecting the forthcoming set of cells to train.

The controller is trained to predict the accuracy $\hat{a}$ of the expanded cells. In POPNAS, the controller is an LSTM model, that receives the separate encodings of the inputs and operators of each block composing the cell specification, forming a temporal series. Inputs and operators are embedded separately and then given as input to the actual LSTM, which is followed by a single sigmoid unit to produce the result.

The time regressor is instead exclusive to POPNAS. The regressor is trained to predict the training time $\hat{t}$ required by the expanded cells. The model chosen for the regressor is based on machine learning techniques, which perform well for regression tasks on limited amount of data, like in the case of this search strategy. In particular, Ridge, NNLS and XGBoost have been tested on the data of multiple runs, where NNLS resulted the most accurate technique for the regressor.

The features used by the time regressor are extracted directly from the encoded cell specification. In details, the features used are:

- the inputs, as 1-indexed categorical encoding
- the operators, as static or dynamic index values
- the number of blocks contained by the cell

Operators indexing

Since the operators are associated to categorical values in the cell encoding, they do not provide direct information about their impact on the training time required by the cell. It becomes essential to find an explicit way to encode the training complexity of the operators included in a cell, since the operators are the main driver of time variations between models.

In POPNAS, the operators set is indexed from 1 to $|\mathcal{O}|$, ranked in base of the time impact of the operators. This feature encoding is referred as *static-indexing*, which is a simplistic way to address the problem, since the categorical values provides the time ranking, but they do not exhibit weighted information about the time distance between the operators.

A more complex and adaptive way to encode the operator values is instead based on the actual training time taken by cells during the run, which considers the difference between the operators required time. This feature encoding is referred to as *dynamic reindex* and it is performed by processing the training times of the cells with a single block, where both inputs are -1 and both operators are the same (symmetric flat cell).

The formula of the dynamic reindex for each operator $o \in \mathcal{O}$, considering T the set of training times required by symmetric cells, and $t_o \in T$ the time taken to train the symmetric flat cell with encoding [(-1, -1, o, o)], is the following:

$$index_o = \frac{t_o}{max(T)} * |\mathcal{O}|$$

The dynamic reindex formula maps each operator to a float value between 0 and $|\mathcal{O}|$. This technique provides a weighted estimate of the training time impact, since the distance between two operators is proportional to the actual training time difference evaluated during runtime. Using dynamic reindex has been proved to be effective in improving the regressor accuracy, compared to just using the static-indexing.

4 | Method: POPNASv2

In this section it is described in details how the POPNASv2 algorithm works, the main innovations introduced in this new version and how all elements of the search method are interconnected.

4.1. Search space

POPNASv2 search space is based on the cell-space approach proposed by NASNet. The operator space has been expanded compared to PNAS, since limiting the search to the Pareto front networks prunes time-consuming operators which do not perform well for the task, making possible to efficiently explore more options. Extending the operator space has a considerable time impact only on the first training step (cells with a single block), since all the possible unique blocks combinations must be trained.

The operator set $\mathcal{O}$ chosen for POPNASv2 consists of the 12 following operators:

- identity
- 3x3 depthwise separable convolution
- 5x5 depthwise separable convolution
- 7x7 depthwise separable convolution
- 1x3-3x1 convolutions
- 1x5-5x1 convolutions
- 1x7-7x1 convolutions
- 1x1 convolution
- 3x3 convolution
- 5x5 convolution
- 2x2 max pooling
- 2x2 average pooling

The input set $\mathcal{I}_b$ changes based on the actual blocks used in the training step. Initially, only the inputs values related to previous cells outputs, referred to as *lookbacks*, are available. POPNASv2 use as lookbacks the last two cells currently built. The input set in the initial training step is therefore $I_1 = \{-2, -1\}$, where the input value for lookbacks can be seen as the cell distance. Enumerating the cells and considering the cell with index j, the input -1 is a sequential connection from cell with index $j - 1$, while -2 represent a

skip connection, since the input is associated to cell $j-2$, jumping a cell.

The input set progressively expands with the amount of blocks b, since the blocks contained in a cell can potentially use as hidden layers the block previously specified. In this case, the input values are associated to the index of each block after enumerating them with 0-indexed integer numbers. Consider as an example a cell composed by 3 blocks: (b_0, b_1, b_2). Regarding b_2, both b_0 and b_1 can be used as hidden layers, so $\mathcal{I}_3 = \{-2, -1, 0, 1\}$, since 0 and 1 are the input values associated respectively to b_0 and b_1. Considering instead b_1, only b_0 can be used as hidden layer, since b_2 is defined later in the encoding and therefore is added on the already existing two blocks through the progressive expansion process.

Each block can be specified as a 4-elements tuple (i_1, o_1, i_2, o_2), with $i_1, i_2 \in \mathcal{I}_b$ and $o_1, o_2 \in \mathcal{O}$. Input i_1 is the hidden layer used in operator o_1, while input i_2 is the hidden layer used in operator o_2. The join operator that process the outputs of o_1 and o_2 is always the addition, so it is not specified in the block encoding, which otherwise should be defined as a 5-elements tuple.

Other search space parameters that can be set in POPNASv2 are K, the maximum amount of networks trained in each step, and Ex, a new parameter which set the maximum amount of networks trained in the exploration step. More information about the exploration step will be given in section 4.3.2.

Similarly to other NAS works using cell motifs, two types of cells are defined: normal cells and reduction cells. Reduction cells have the same structure of normal cells but operators using lookbacks as inputs perform a stride 2, reducing the spatial dimension of the tensors, and also double the amount of filters. PNAS proved that searching for a single common cell specification for both normal and reduction cells is a valid strategy to drastically reduce the search time without a significant impact on the accuracy reached by the networks. POPNASv2 follows the same principle, given that search efficiency is one of the main goals of the search method.

4.1.1. Blocks equivalence

The block structure is usually referred in the literature as a 5-tuple $(i_1, o_1, i_2, o_2, o_{join})$. It can also be seen as a very simple DAG, with two separate branches that are merged together with the join operator. We can refer these two branches as left(L) and right(R), as shown in Fig. 4.1.

Swapping the left and right branches does not change the actual structure of the block if

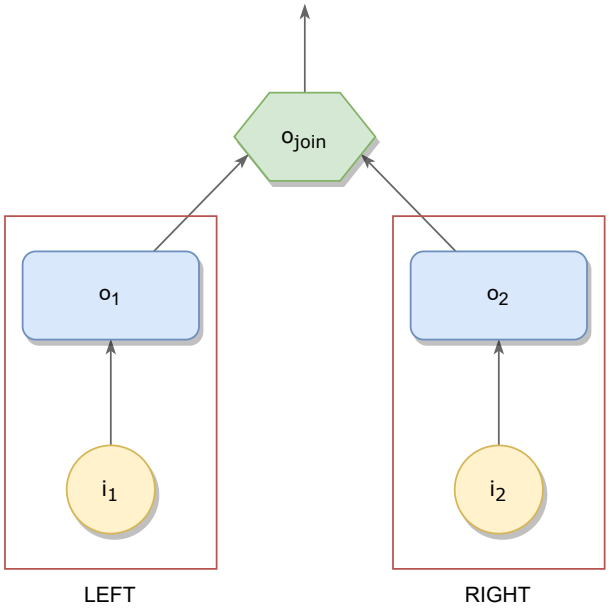


Figure 4.1: Visualization of a generic block DAG. Left and right branches can be swapped without changing the results, if o_{join} is commutative.

o_{join} is a commutative operator (which is usually the case in NAS algorithms), therefore all symmetric blocks are actually equivalent. Since POPNASv2 always use addition as o_{join}, only one block of a symmetric pair should be generated during cell expansions, to avoid producing equivalent models. POPNASv2 filters the blocks using a simple condition check, the only requisite for applying it is to enumerate the input and operator values with integer numbers (categorical label encoding). Considering a block with enumerate values $(i_{en1}, o_{en1}, i_{en2}, o_{en2})$, the block is generated only if the following condition is satisfied:

$$i_{en2} \geq i_{en1} \wedge (i_{en2} \neq i_{en1} \vee o_{en2} \geq o_{en1})$$

4.1.2. Cells equivalence

While block symmetries are explicitly mentioned in PNAS work, there are other cases involving cells that can cause different cell specifications to generate equivalent neural network models. POPNASv2 investigates also these cases, checking them during the expansion step to avoid the training of multiple equivalent cells. For instance, it is possible to consider as an example these two different cell specifications:

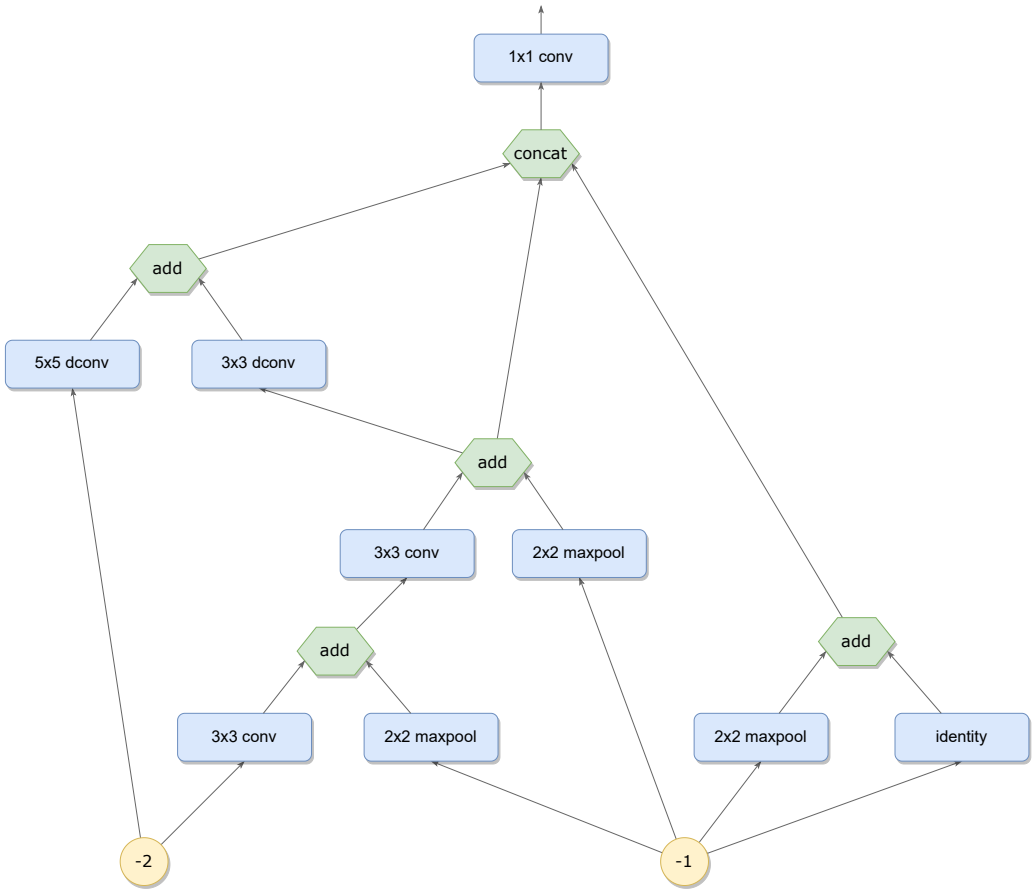


Figure 4.2: An example of cell equivalence. The two given cell specifications c and c', albeit different, produce the same cell DAG illustrated here.

$$\text{c} = \begin{array}{l}[(\text{-2, '3x3 conv', -1, '2x2 maxpool'}),\\ (\text{0, '3x3 conv', -1, '2x2 maxpool'}),\\ (\text{-2, '5x5 dconv', 1, '3x3 dconv'}),\\ (\text{-1, '2x2 maxpool', -1, 'identity'})]\end{array} \qquad \text{c'} = \begin{array}{l}[(\text{-1, '2x2 maxpool', -1, 'identity'}),\\ (\text{-2, '3x3 conv', -1, '2x2 maxpool'}),\\ (\text{1, '3x3 conv', -1, '2x2 maxpool'}),\\ (\text{-2, '5x5 dconv', 2, '3x3 dconv'})]\end{array}$$

c and c' have the same exact cell representation in the actual neural network model, represented in the Fig. 4.2.

Considering two cells $c = [b_1, b_2, ...]$ and $c' = [b'_1, b'_2, ...]$, where each block is the usual 4-elements tuple $b = (i_1, o_1, i_2, o_2)$, the two cells are equivalent if:

$$|c| = |c'| \wedge c' = [\sigma(b_1), \sigma(b_2), ...]$$

where σ is a permutation. If the sequence of block of c' is a permutation of the blocks of c, it means that each block of c' is equivalent to a different block of c, therefore it is also needed to define the block equivalence.

Considering the block as two branches (L, R), as explained in section 4.1.1, it is possible to define the block equivalence as an equivalence of the two branches. A branch $br = (i, o)$ is equivalent to $br' = (i', o')$ if:

$$i = i' \wedge o = o'$$

A particular case is when i or i' is a hidden layer produced by another block. In this case, the input is not a simple integer value but the block object itself, so the block equivalence is performed instead of the trivial value equivalence check.

After defining the branch equivalence, it is finally possible to define the block equivalence formula, that is the last rule needed to define the cell equivalence. Considering two blocks $b = (L, R)$ and $b' = (L', R')$, the two blocks are equivalent if:

$$(L = L' \wedge R = R') \vee (L = R' \wedge R = L')$$

The block equivalence uses the branch equivalence rule, but in case non-lookback input values are used inside a branch, the input equivalence is resolved using the block equivalence rule. The block rule resolution can therefore be recursive. Anyway, this is not a problem since the cell specification is guaranteed to be a DAG and therefore the recursion can be performed at max $B - 1$ times, afterwards it is guaranteed to be a simple value equivalence between lookback inputs.

4.1.3. Search space cardinality

The amount of unique blocks that can be generated in each training step depends only on the cardinality of the input and operator sets. Since the input set cardinality increases with the number of blocks, also the amount of generable blocks increases at each step.

It is possible to derive the total amount of unique blocks $|\mathcal{B}_b|$ that can expand a cell with $b - 1$ blocks. To do so, it is necessary to compute first the total amount of generable blocks including the equivalences ($|\mathcal{B}_{eqv,b}|$) and the amount of symmetric blocks ($|\mathcal{B}_{sym,b}|$). The equations are given below:

$$|\mathcal{B}_{eqv,b}| = |\mathcal{I}_b|^2 * |\mathcal{O}|^2$$

$$|\mathcal{B}_{sym,b}| = |\mathcal{I}_b| * |\mathcal{O}|$$

Block specifications with same inputs and operators (i, o, i, o) have no equivalent specifications, because they are already symmetric. Other blocks instead have always another

specular representation which is equivalent to them, as also explained in Section 4.1.1. The amount of unique blocks can be found with this formula:

$$|\mathcal{B}_b| = \frac{|\mathcal{B}_{eqv,b}| - |\mathcal{B}_{sym,b}|}{2} + |\mathcal{B}_{sym,b}| = \frac{|\mathcal{B}_{eqv,b}| + |\mathcal{B}_{sym,b}|}{2}$$

Using this formula, it is possible to find the amount of unique "monoblock" cells trained by POPNASv2 in the first training step, performed after the training of the empty cell. This amount of blocks is particularly important since POPNASv2 needs to train all possible cells with $b = 1$. Changing either the operator set $\mathcal{O}$ or the lookback inputs $\mathcal{I}_1$ cardinality alters the amount of generable blocks, therefore they directly impact the time required to perform the first training step. The amount of cells trained in other steps is instead capped by K and Ex parameters, so expanding $\mathcal{O}$ and $\mathcal{I}_b$ will not directly alter these limits.

Since in the standard POPNASv2 configuration $|\mathcal{O}| = 12$ and $|\mathcal{I}_1| = 2$, the total amount of unique monoblock cells is 300. The search space cardinality $|\mathcal{C}|$ is the number of possible cells that can be generated by POPNASv2 algorithm during the entire search process. An upper bound of the search space cardinality can be found as:

$$|\mathcal{C}| < \prod_{b=1}^{B} \mathcal{B}_b$$

This formula does not consider the equivalent cell specifications, which are actually pruned by POPNASv2 as they are redundant. Using the default POPNASv2 configuration, this means that the upper bound of the search space cardinality for $B = 5$ is $|\mathcal{C}| < 300 * 666 * 1176 * 1830 * 2628 = 1.13 * 10^{15}$. The main challenge of POPNASv2 is to efficiently search through this vast search space, training meanwhile neural networks models which are quite variegate in the results, since it aims to find a Pareto front of the time-accuracy optimization problem.

4.2. From cell specification to actual CNN

During the algorithm run, all the cells selected for the current training step are stored in an encoded form, from which the CNN can be built. The encoded representation of each cell is simply a list of the encoded blocks, so a list of 4-elements tuples. When a cell is selected for training, the model generator builds the network from scratch, using the cell specification info. All the networks are built with the same rules, but the model generator behavior can be customized through the configuration file provided at the start

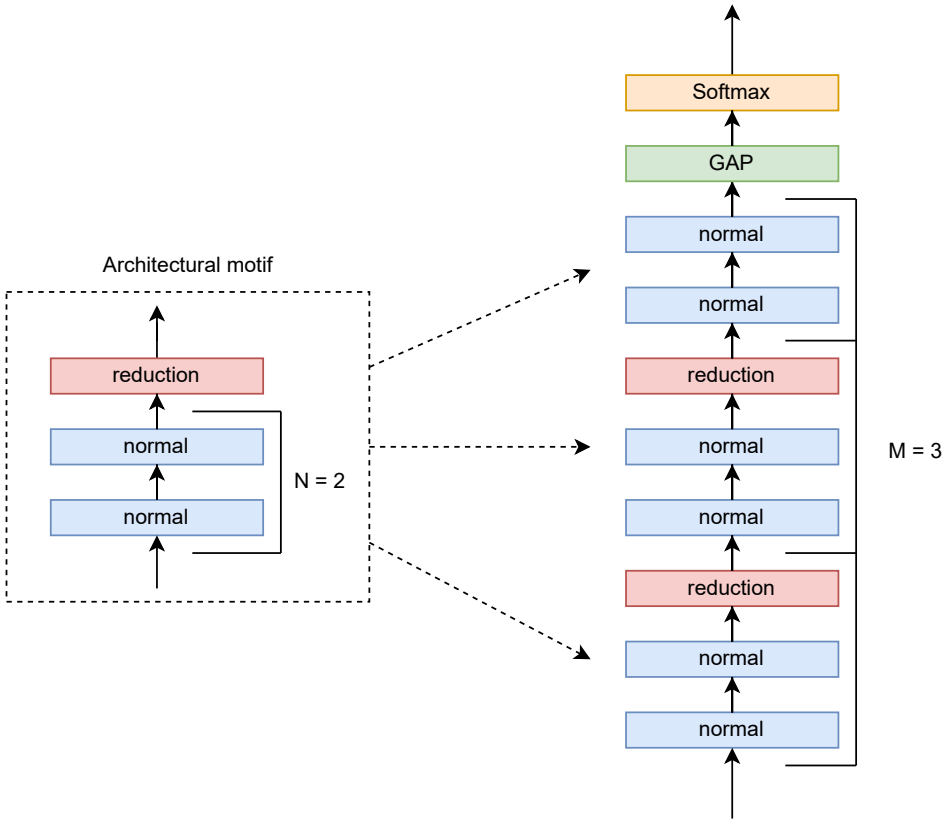


Figure 4.3: An example of architecture definable with POPNASv2 search space and architecture parameters. N indicates the number of normal cells in each architectural motif, while M is the number of motifs stacked to build the neural network architecture. The last reduction cell is replaced with GAP and Softmax.

of the run.

In general, the CNN is built as a stack of cells, as usual in cell-based NAS algorithms. POPNASv2 additionally defines an architectural motif with a higher level of abstraction than the cell, constituted as a stack of N normal cells followed by a single reduction cell. The final model produced is the stack of M architectural motifs, but the last motif has actually no reduction cell and instead is followed by GAP and Softmax layers to compute the final output. A visual example is provided in Fig. 4.3. The last reduction cell is cut from the model since reducing the tensor dimensions right before the output computation, without further processing, will not be much beneficial, compared to the increase of parameters and training time due to the presence of this additional cell.

4.2.1. Generating the cell

The neural network architecture is a composition of M architectural motifs, which are themselves compositions of the two types of cell used in POPNASv2. The main logic of the model generator is formed by the set of rules on how to build the cell implementation. At start, all lookback inputs simply refer to the input image (or more precisely the batch of input images), since there are no cell outputs available. Every time a cell is generated, its output tensor is referred as the -1 input for the next cell, while the other lookback input scale of a position, becoming associated to the -2 input value. The blocks contained by the considered cell are enumerated and produced in the encoding order, since the other following blocks could use the previous ones as hidden layer. To address these cases, the output tensor of each generated block is inserted in the pool of available inputs for the current generated cell. The inputs related to the blocks are associated to the position in which they appear in the cell encoding. After a cell of the architecture has been generated, the input value set shrinks back to the set of the lookback values, properly updated with the new cell output. The inputs related to the blocks are instead discarded, since they cannot be directly used in the following cells.

When the cells are composed of multiple blocks, more than one block output could remain unused. In this case, the cell output is computed through a concatenation of all the unused block outputs, followed by a pointwise convolution to reduce the output filters to the target for the current cell. Similar to PNAS, POPNASv2 opts for doubling the filters only in the first level of reduction cells, so the pointwise convolution is indeed needed to maintain the desired tensor depth across the neural network. There are other cases which require attention regarding the tensor shape, since otherwise it would be difficult to combine the blocks and the cells together in a single neural network model. Some general rules can be identified for blocks and cells; any operator or stack of operators that satisfy the following requisites are guaranteed to work in POPNASv2 model generator.

Regarding the blocks, all the ones belonging to the same cell must have the same output shape. To achieve this result, all operators involved in the cell must have exactly the same output shape. This constraint allows to perform the addition of the results of the two block operators, making also the output shape of a block equal to the output shape of the operators. Since reduction cells alter both spatial and depth dimensions, POPNASv2 operators are required to adapt the tensor shape when necessary. Convolutional operators can easily achieve the shape adaptation through the stride and filters parameters, instead pooling operators are followed by a pointwise convolution in reduction cells since they cannot adapt the tensor depth. This is also the case for identity operator, that become a

pointwise convolution inside reduction cells.

Regarding the cells, the output shapes must have a specific shape compared to the input shape. Given an input shape (H, W, F), the output shape of normal cells is exactly equal to the input shape, while for the reduction cells the output shape is $(\frac{H}{2}, \frac{W}{2}, 2F)$. Additionally, all lookback inputs must have the same tensor shape. When using different lookback values, this requirement is broken every time the -1 input is the output of a reduction cell, since -2 would be the output of a normal cell and so they would have different shapes for the point explained above. To address the shape difference in these cases, all lookback inputs that come from normal cells are processed by a (2, 2) strided pointwise convolution, that adjust both spatial and depth dimensions to the one of the reduction cell. In this way the next cell can process the inputs without further adjustments.

4.3. Search strategy

POPNASv2 search strategy is composed of multiple phases:

- the initial thrust, which is the training of the empty cell.
- the training step, in which the selected cells are built and trained.
- the expansion step, which is aided by two predictors and estimates the quality of all possible expansions of the cells trained in the training step. A Pareto front is built from these estimates, solving the time-accuracy optimization problem.
- the exploration step, a conditional phase with the goal of exploring input and operator values underused in the current Pareto front.

POPNASv2 starts the training procedure by training the empty cell, which produce a very simple neural network model composed only by a GAP layer followed by Softmax. This step is referred as "initial thrust" and it is necessary to set up the features used by the predictors. In fact, the training time t_0 and accuracy a_0 reached by this network can be considered a common bias of all neural networks trained, because they are impacted by elements included in all architectures generated by the algorithm. Considering the difference between a cell results (t, a) and the bias (t_0, a_0) gives a better estimate on the impact of inputs and operators specified in the cell, on both time and accuracy metrics, improving the predictors accuracy.

After completing the empty cell training, all unique cells with a single block are trained on the target number of epochs E. POPNASv2 saves the training time t and the best validation accuracy a reached by each cell for further analysis and for training the predic-

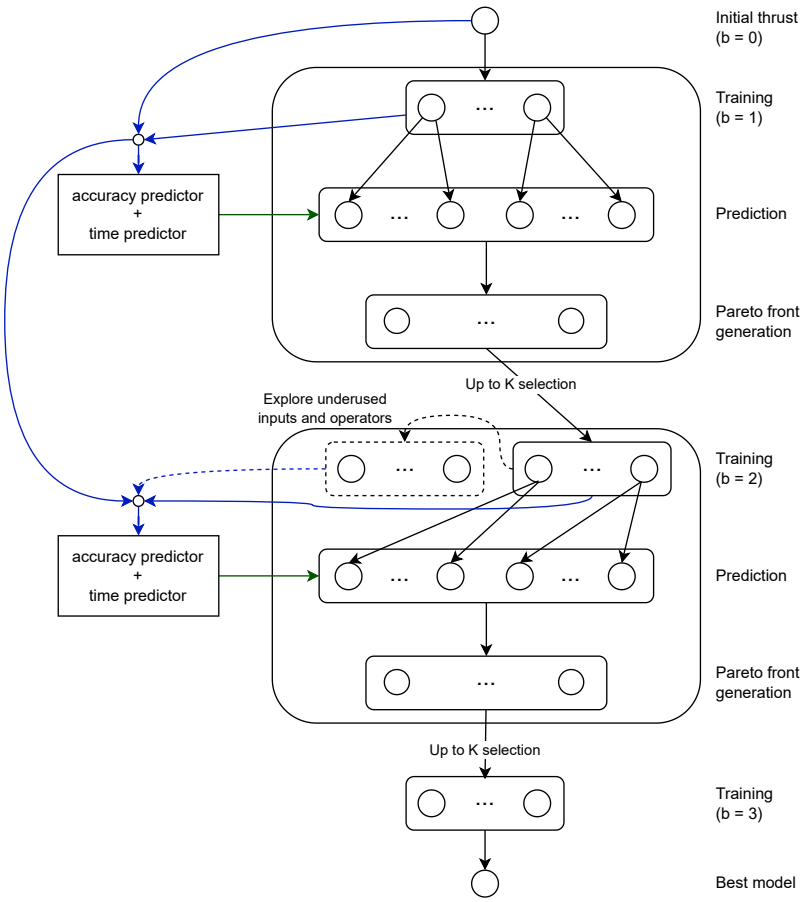


Figure 4.4: Illustration of POPNASv2 search procedure on $B = 3$. The process starts with the initial thrust and then all generable unique monoblocks are trained. For steps with $1 < b < B$, the exploration Pareto front can be built to explore elements underused in the actual Pareto front, to help the predictors to reconsider them if they actually perform well in a given step.

tors. This concludes the training step of the cells with $b = 1$. For all following training steps ($b > 1$), the search process will repeat cyclically until the target $b = B$ is reached. An overview of the search procedure is given in Fig. 4.4 and Algorithm 4.1.

Both time and accuracy predictors are trained on the data of all networks trained during the search, after each training step except the final one. More details on the predictors are given in the Section 4.4.

4.3.1. Expansion step

After the predictors have been fully trained, they are used to estimate the training time $\hat{t}$ and the validation accuracy $\hat{a}$ of all the possible expansions of the cells trained in the

Algorithm 4.1 POPNASv2 search strategy

Require: B (max num blocks), E (epochs), K (beam size), J (exploration beam size), T (time constraint), *CNN-hp* (networks hyperparameters), *dataset*.

$S_0 =$ empty cell
$\mathcal{A}_0, \mathcal{T}_0 =$ train-cells$(S_0, \text{CNN-hp}, \text{dataset})$
$S_1 = B_1$
$\mathcal{A}_1, \mathcal{T}_1 =$ train-cells$(S_1, \text{CNN-hp}, \text{dataset})$
dynamic-reindex = initialize-reindex$(\mathcal{A}_0, \mathcal{T}_0, \mathcal{A}_1, \mathcal{T}_1)$
for $b = 2 : B$ **do**
  $\mathcal{P}_{acc} =$ fit$(\mathcal{A}_{0\to b-1}, S_{0\to b-1})$
  $\mathcal{F}_{b-1} =$ extract-features(S_{b-1})
  $\mathcal{P}_{time} =$ fit$(\mathcal{T}_{0\to b-1}, \mathcal{F}_{0\to b-1})$
  $S'_b =$ expand-cells(S_{b-1})
  $\mathcal{A}'_b =$ predict$(S'_b, \mathcal{P}_{acc})$
  $\mathcal{T}'_b =$ predict$(S'_b, \mathcal{P}_{time})$
  $S''_b, \mathcal{A}''_b, \mathcal{T}''_b =$ apply-time-constraint$(S'_b, \mathcal{A}'_b, \mathcal{T}'_b, T)$
  $S_b =$ build-pareto-front$(S''_b, \mathcal{A}''_b, \mathcal{T}''_b, K)$
  $\tilde{\mathcal{O}}, \tilde{\mathcal{I}}_b =$ build-exploration-sets(S_b)
  if $|\tilde{\mathcal{O}}| > 0 \vee |\tilde{\mathcal{I}}_b| > 0$ **then**
    $S_{b,exp} =$ build-epf$(S''_b, \mathcal{A}'_b, \mathcal{T}'_b, S_b, J)$
  else
    $S_{b,exp} = \{\}$
  end if
  $\mathcal{A}_b, \mathcal{T}_b =$ train-cells$(S_b \cup S_{b,exp}, \text{CNN-hp}, \text{dataset})$
end for

previous step. An expansion of a cell is simply the addition of any block to an already existing cell. The amount of unique blocks which can expand a cell to $B = b$ blocks can be found with the formula presented in Section 4.1.3.

POPNASv2 searches the best cells in the so-called expansion step, by identifying the Pareto front of the time-accuracy optimization problem, using $\hat{a}$ and $\hat{t}$ estimated by the predictors.

The way in which the Pareto front is built is the same of the first version of POPNAS. As a first step, if the parameter T is provided, POPNASv2 discards all the expanded cells that have $\hat{t} > T$. T is therefore a time constraint that can be optionally configured to restrict the search on less demanding network, which can be beneficial if these neural networks architectures are designed to run on low performance hardware.

After this optional preprocessing step, the Pareto front is built. The cell with the highest predicted accuracy $\hat{a}$ is inserted as the Pareto front first element to initialize the process, then a domination rule is applied to build the rest of the Pareto front.

The domination rule compare two different cells and it is defined as follows: Cell 1, having predicted accuracy $\hat{a_1}$ and predicted training time $\hat{t_1}$, dominates cell 2, having predicted accuracy $\hat{a_2}$ and predicted training time $\hat{t_2}$, if $\hat{t_1} \leq \hat{t_2}$ and $\hat{a_1} > \hat{a_2}$. All cell expansions are iteratively selected and compared with the last element inserted in the Pareto front; if the considered cell is not dominated, then the cell is insert in the Pareto front.

The Pareto frontier is therefore the set of all non-dominated cells which enable the trade-off between the considered objectives: time and accuracy. The Pareto front cardinality is limited up to K elements, but it could actually be less than K if the dominant cells are very sparse, providing additional search speedups when some architectures are considerably better than others.

4.3.2. Exploration step

POPNASv2 algorithm introduce a new conditional component called *exploration step*. Exploration-exploitation dilemma is a common theme in many machine learning techniques, since exploiting the information gathered during the training can lead to premature convergence to suboptimal regions of the search space. Regarding NAS field, exploiting too much the information gathered while sampling the search space can bias the search towards suboptimal architectural patterns. In fact, after the search method discovers good combination of input and operators, it is more incentivized to reuse these combinations with minimal alterations, which is an optimal behavior only if the algorithm actually converged to the subspace of optimal neural network architectures.

The goal of the exploration step is to train a small supplementary set of architectures that contains different characteristics from the ones prevalent in the Pareto front. The Pareto front exploits heavily the information about the metrics retrieved during the previous training steps, but it is possible that the expanded cells could benefit more from operators and inputs that did not perform well in previous iterations. The complexity of the models progressively increase with the number of blocks, therefore the search method is exploring a new subset of the search space in each step. Considering this behavior, testing previously suboptimal operators and inputs has a non-negligible potential of producing different results.

The exploration step also helps in avoiding a potential problem caused by the input set. As explained in the Section 4.1, the input set increase in cardinality at each expansion step, since the previously last block of the cell could now be used as hidden layer in the block inserted during the cell expansion. This new input value is therefore never observed in the previous architectures trained, so the prediction of these architectures are more

influenced by noise. The exploration step can help in exploring this new input value, in case it is rarely inserted in the Pareto front.

The exploration step is performed right after the expansion step (except for final expansion step) and basically finds a small set of significant architectures to train, exploiting again the Pareto front technique. Differently from the expansion step, here the architectures inserted in the *exploration Pareto front* (EPF) not only need to satisfy the domination rule of the multi-objective optimization problem, but also to satisfy a special score metric defined appositely for the exploration step.

Initially, POPNASv2 computes the set of inputs and operators to explore. The exploration sets contain inputs and operators having a percentage of utilization in the Pareto front (i_{perc} and o_{perc}) lower than a given threshold. Consider as example a Pareto front composed by 10 cells of 3 blocks, since each block contains 2 operators and 2 inputs, there are a total of $2 * 3 * 10 = 60$ inputs and operators inside this Pareto front. Considering an operator $o \in \mathcal{O}$, if o appears 6 times inside these cells, then its percentage of utilization is $o_{perc} = \frac{6}{30} = 0.2$.

An operator $o \in \mathcal{O}$ is inserted in the operator exploration set $\tilde{\mathcal{O}}$ if $o_{perc} < \frac{1}{5|\mathcal{O}|}$, similarly an input $i \in \mathcal{I}_b$ is inserted in the input exploration set $\tilde{\mathcal{I}}_b$ if $i_{perc} < \frac{1}{5|\mathcal{I}_b|}$.

After defining both $\tilde{\mathcal{O}}$ and $\tilde{\mathcal{I}}_b$, the EPF can be built by processing the forecasted values $(\hat{a}, \hat{t})$ of each cell expansion. A trivial case is when both $\tilde{\mathcal{O}}$ and $\tilde{\mathcal{I}}_b$ are empty, in this case the exploration step is skipped since the usage of both inputs and operators is quite balanced. If at least one of the exploration sets is instead not empty, the EPF is computed.

To build the EPF, each evaluated cell processed is associated to an exploration score, that changes dynamically over time while the exploration Pareto front is built. The reason for dynamically changing the score is to balance the exploration of both inputs and operators sets, also trying to encourage the usage of all the values of an exploration set, if $\hat{t}$ and $\hat{a}$ are good enough to fit the Pareto front. To adapt the score system, the algorithm counts the total usage of exploration inputs and exploration operators in the exploration Pareto front, referred as $|i_{exp}|$ and $|o_{exp}|$, and the individual usage of each exploration input and exploration operator as a percentage of respectively $|i_{exp}|$ and $|o_{exp}|$, referred as i_{perc} and o_{perc}.

The rules that attribute exploration points to a cell are described as follows:

- $+1$ for each input $i \in \tilde{\mathcal{I}}_b$, with bonus:
  - $+2$ if $i_{perc} \leq \frac{1}{|\tilde{\mathcal{I}}_b|}$

  - +1 if $|i_{exp}| \leq |o_{exp}|$

- +1 for each operator $o \in \tilde{\mathcal{O}}$, with bonus:
  - +2 if $o_{perc} \leq \frac{1}{|\tilde{\mathcal{O}}|}$
  - +1 if $|i_{exp}| \geq |o_{exp}|$

If the score is > 0, further points can be assigned based on the difference between the predicted time and predicted accuracy of the considered cell and the ones of the last element of the EPF. In particular, one point is added for each 4% of relative accuracy difference and for each 10% of relative time difference.

To be accepted for the insertion in the EPF, a cell must have a score ≥ 8 if both $\tilde{\mathcal{O}}$ and $\tilde{\mathcal{I}}_b$ sets are populated, ≥ 4 instead if one of them is empty. Since initially the EPF is empty, the additional rules considering time and accuracy difference are not used, but they are activated for the evaluation of the second element.

The exploration set generates an exploration Pareto front of at most Ex elements, a parameter given in the run configuration When the exploration step completes, both the standard Pareto front and the EPF are processed by the training step, in this order. The training step is followed by the expansion step, and the process repeats cyclically until the target amount of blocks B is reached.

4.4. Performance estimation strategy

Given the enormous size of the search, it is fundamental to predict as accurately as possible the variables to optimize in the multi-objective optimization problem. POPNASv2 implements two additional surrogate models, referred as *time predictor* and *accuracy predictor*, to forecast the training time $\hat{t}$ and accuracy $\hat{a}$ of each individual cells during expansions. The predictors are fundamental to enable the Pareto front technique and their results can significantly alter the search. The two predictors are trained after each training step with $1 \leq b < B$ blocks, retraining them each time with the results of all the cells trained during the search, and not just the cells trained in the most recent training step.

4.4.1. Accuracy predictor

The accuracy predictor is inspired by PNAS, which utilized in its definitive experiments an ensemble of 5 LSTM models. POPNASv2 uses an ensemble of 5 LSTM models, but they are built with a slightly different structure and hyperparameters compared to PNAS.

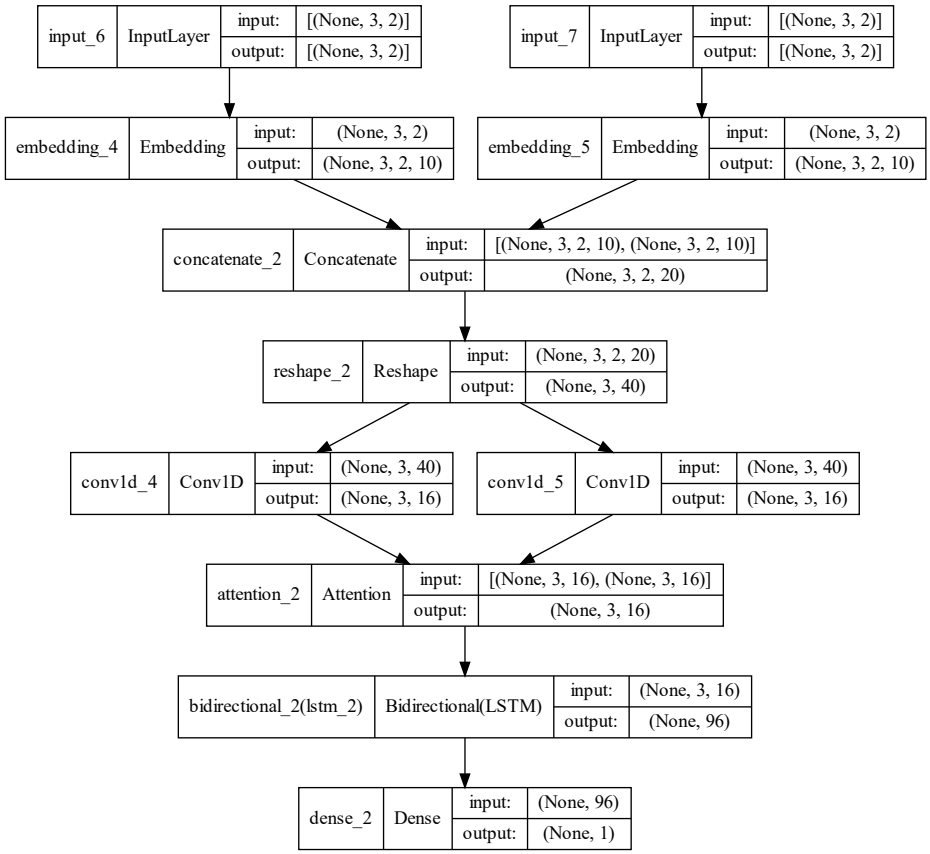


Figure 4.5: The neural network structure of an LSTM used by the accuracy predictor. All LSTM used in the ensemble share the same structure. The first dimension (None) is the batch size. The second dimension is $B = 5$, the maximum amount of blocks per cell.

The LSTM model receives two inputs, based on the cell encoding: a tensor with the blocks' inputs, grouped per block, and a tensor with the blocks' operators, also grouped per block. Both tensors have dimension $(B, 2)$, since each block has 2 inputs and 2 operators and the maximum amount of blocks present in a cell is equal to B, the target amount of blocks for the search run. Both input and operator values are encoded as 1-indexed categorical. If the considered cell has fewer blocks than B, then both input tensors are padded with (0, 0) for each missing block. The two tensors are then processed separately with an embedding layer and finally concatenated together. This new tensor is then processed by two different conv1D layers, which produce the Q and K tensors of the Attention layer [24] that follows.

The output of the Attention is finally used as input for a bidirectional LSTM, which produces the final hidden layer used by the sigmoid unit to predict the final value, the

estimated accuracy $\hat{a}$ of the cell. The final accuracy predictor is composed by 5 LSTM models with the same exact structure. Each LSTM is trained on 4 folds of a 5-folds training, so on the results of 80% of the cells trained at that point. When the ensemble is used to predict the accuracy $\hat{a}$ of a given cell specification, the output is simply the mean of all models' predictions.

4.4.2. Time predictor

The time predictor is the agent responsible for estimating the training time $\hat{t}$ that a cell would require to train on E epochs. Contrary to the accuracy predictor, which is based on a neural network model, the time predictors tested in POPNASv2 are based on machine learning techniques. In particular, the default time predictor employed in the experiments is based on CatBoost [27], which is an implementation of gradient boosted decision trees.

Unlike neural network models, that can extract features directly from data, it is very important to have a significant feature set to maximize the accuracy of a regressor based on standard machine learning techniques. Using the categorical values of inputs and operators as features does not lead to good results, therefore a more sophisticated feature set is extracted directly from the structure of the cell DAG used in the actual neural network model. Since the features are extracted from the DAG, which is still an abstraction of the actual model, it should generalize on any neural network framework chosen for the actual implementation. More details are given in Section 5.3.3.

The features set used to train the time predictor is described below:

- number of blocks
- number of cells
- the sum of the dynamic reindex value of each cell block operator (OP score)
- number of concatenated tensors in cell output
- usage of multiple lookbacks (boolean)
- cell DAG depth (in blocks)
- number of block dependencies
- % of the total cell OP score related to the heaviest cell path
- % of the total cell OP score related to the blocks using lookbacks as input.

The dynamic reindex is a metric introduced in POPNAS to evaluate and rank single

operators impact with respect to their training time. In order to achieve this goal, a preliminary evaluation is carried out on cells composed of single blocks containing two identical operators.

POPNASv2 implements a revised version of dynamic reindex which takes into account time biases due to neural network components present in all the generated configurations. In detail, the time t_0 is due solely to GAP and Softmax layers and to the data augmentation process, which are common to all architectures, therefore excluding t_0 provides a more fair estimation of the impact of the operators on the training time. For each operator $o \in \mathcal{O}$, considering $t_o \in T$ the time taken to train the symmetric flat cell with encoding [(-1, o, -1, o)], the corresponding dynamic reindex value is computed as:

$$index_o = \frac{t_o - t_0}{max(T) - t_0}$$

This small change has contributed to significantly boost the accuracy of the time predictor. The time predictor is trained using random search for hyperparameter tuning. The random search performs multiple training iterations on k-folds, using early stopping to preventively terminate the runs using bad hyperparameters, making the process more time efficient. After the random search completes, the best set of hyperparameters is used to train the final CatBoost model, using all the samples available. This final model is responsible for estimating the training time $\hat{t}$ of each cell expansion.

5 | Ablation studies

In this chapter it is presented a set of tests about the new features introduced by POPNASv2. These tests demonstrate the impact of these modification on the final results. Most of the ablation studies have been conducted on multiple image classification datasets, i.e., CIFAR10, CIFAR100 [15], Fashion MNIST [37] and EuroSAT [10]. These datasets are also used in the final experiments, which are discussed in Chapter 6. All runs use the same set of hyperparameters and differ only for the input dataset.

5.1. Equivalence check

The equivalence check purpose is to increase the efficiency of the search method by avoiding the training of the same architecture multiple times, since many architectures do not have a unique cell encoding.

Table 5.1: Total equivalent cells detected by the equivalence check, for each tested dataset.

Dataset	POPNASv2	PNAS
CIFAR10	2	211
CIFAR100	0	236
Fashion MNIST	7	305
EuroSAT	6	232

Tab. 5.1 shows the number of cells detected by the cell equivalence check in both POPNASv2 and PNAS. These numbers do not include the equivalent blocks case, since the check discussed in Section 4.1.1 prunes equivalent block encodings a priori. POPNASv2 actually does not prune lots of cells since the Pareto front mechanism drastically reduce the chances that multiple equivalent cell encodings are selected for training. Anyway, if the detected equivalent cell encodings are not pruned in the early steps of the algorithm, these encodings could lead to a potential multiplication of these equivalences during the search procedure expansions. With this consideration, the cell equivalence check avoids potential harmful alterations of the Pareto front and improves the general efficiency of the algorithm, since the check is almost instant to perform.

Table 5.2: Total amount of cells which are expansions of a cell introduced by the exploration step, for each tested dataset.

CIFAR10	CIFAR100	Fashion MNIST	EuroSAT
2	1	29	42

PNAS instead benefits significantly from this change, since it just selects the most performing networks. This means that, if the predictor is able to generalize $\hat{a}$ of the multiple equivalent cell encodings, top networks would appear in the selection with multiple encodings, since all of them would have similar $\hat{a}$ values. This behavior explains the significantly higher numbers related to PNAS, shown in Tab. 5.1.

5.2. Exploration step

The exploration step goal is to question if the predictors behavior is biased from the results of the simplest architectures, disregarding input and operator values which did not perform well in the previous training steps.

Tab. 5.2 shows the amount of cells introduced in the search thanks to the exploration step, for each tested dataset. These amounts consider all the cell specifications which are direct expansions of the cells trained in the exploration step. This means that, in the expansion step that follows the training of the EPF, some expansions of the EPF cells are actually inserted the Pareto front since they are non-dominated by others.

Search runs on FashionMNIST and EuroSAT datasets have benefited a lot from the exploration step, since a high amount of networks derives directly from cells introduced by the exploration step. Runs on CIFAR10 and CIFAR100 instead do not exhibit this behavior, but the exploration step still has significant benefits since it reduces the uncertainty of the results of less utilized inputs and operators. Furthermore, predictors could still reintroduce these explored values in the search, even without directly expanding the cells trained in the exploration step, therefore the impact of the exploration step should not be underestimated even when EPF cells are discarded.

The goal of the adaptive greedy exploration is to correct the potential convergence to suboptimal solutions, but this extra search effort not always translate to significantly better top networks. The exploration step mainly stabilize the search results and can help in increasing the Pareto variety, which is beneficial for adapting the search method to different tasks.

5.3. Time predictors features

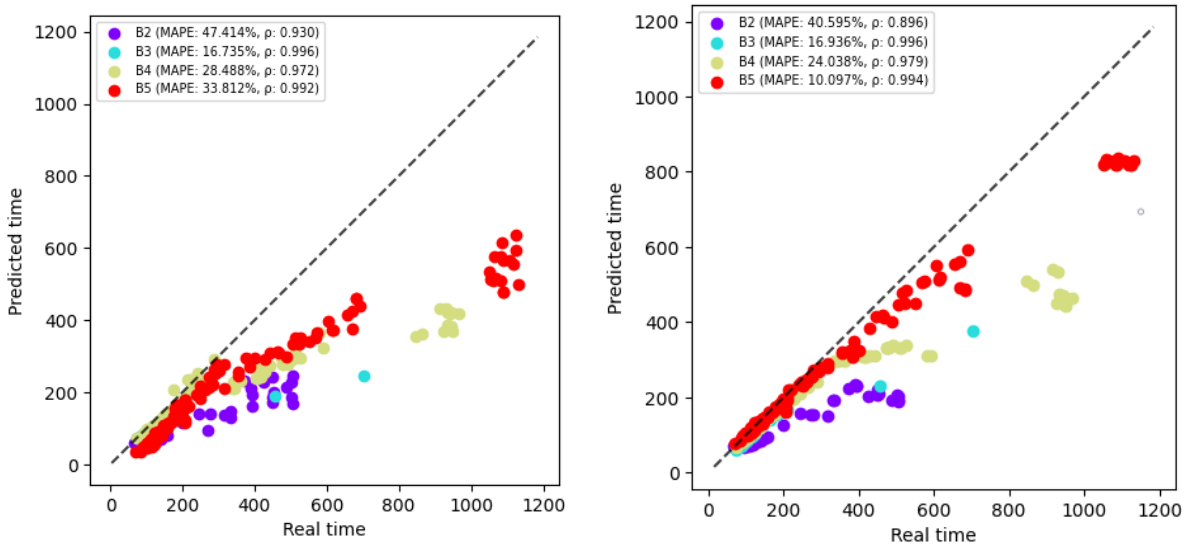
5.3.1. POPNAS feature set

The feature set used the first version of POPNAS to train time predictors is based on the abstract cell encoding stored during the search method. For each cell specification, the features extracted are the following:

- the number of blocks
- a feature for each operation, produced by the dynamic reindex
- a feature for each input, as 1-indexed categorical

This feature set have some intrinsic problems that lead to poor time forecasts on the expanded CNN structures. In particular:

- inputs are categorical features, but among all tested time predictor models only CatBoost can properly handle them. Other models would treat them as integers, considering a ranking among them, which is totally wrong in this case (e.g. input 3 would have 3 times the impact of input 1 on final result).
- time values used to build the dynamic reindex includes the bias t_0 of the empty cell. As explained in Section 4.4, t_0 embeds characteristics common to all neural network architectures trained, so it should be subtracted from the times considered by the dynamic reindex. This change should provide a more fair comparison of the operators impact on the training time.
- the number of features produced depends on the max number of blocks B set in the search algorithm, which is something not desirable.
- when the blocks included in a cell is $b < B$, the features of the blocks not present are set to 0, producing a sparse feature set. Also, since these features are positional, multiple records must be added with sliding block mechanism to make the predictor generalize better on the positional features, trying to give them similar importance.
- equivalent cell specifications produce different features, therefore the variance in their training time estimates depends on how well the predictor generalize on the equivalent encodings.



(a) original feature set (b) new tentative feature set

Figure 5.1: CatBoost predictions comparison between POPNASv1 feature set and the new tentative feature set.

5.3.2. Tentative feature set

A new tentative feature set has been developed, to address the first two problems highlighted for POPNAS feature set, which are the most critical. This new feature set avoids the usage of categorical inputs, providing instead a set of alternative numeric features which also give more information about the considered cell structure. In details, the input features have been substituted with:

- usage of each lookback input inside the cell
- usage of each lookback inside a specific block ($|\mathcal{I}_1| \cdot B$ boolean features)
- incidence matrix of the blocks usage in other blocks, since a block can use as input the output of a previous block (lower triangular matrix, boolean features)

Boolean values are encoded as $\{0, 1\}$ to be interpretable by the regressors. Dynamic reindex have been changed to remove the bias (the time t_0 of the empty cell network) and to normalize its output into interval $[0, 1]$, as explained in Section 4.4.2. Also, the number of cells stacked inside the CNN model has been added as an additional feature, since the actual number of cells used depends on the lookbacks usage and therefore is not the same for all networks.

If input -1 is used, the model contains the target amount of cells, otherwise it contains a number of cells equal to $\lceil \frac{\text{target cells}}{\text{min lookback distance}} \rceil$. For example, considering the default POP-

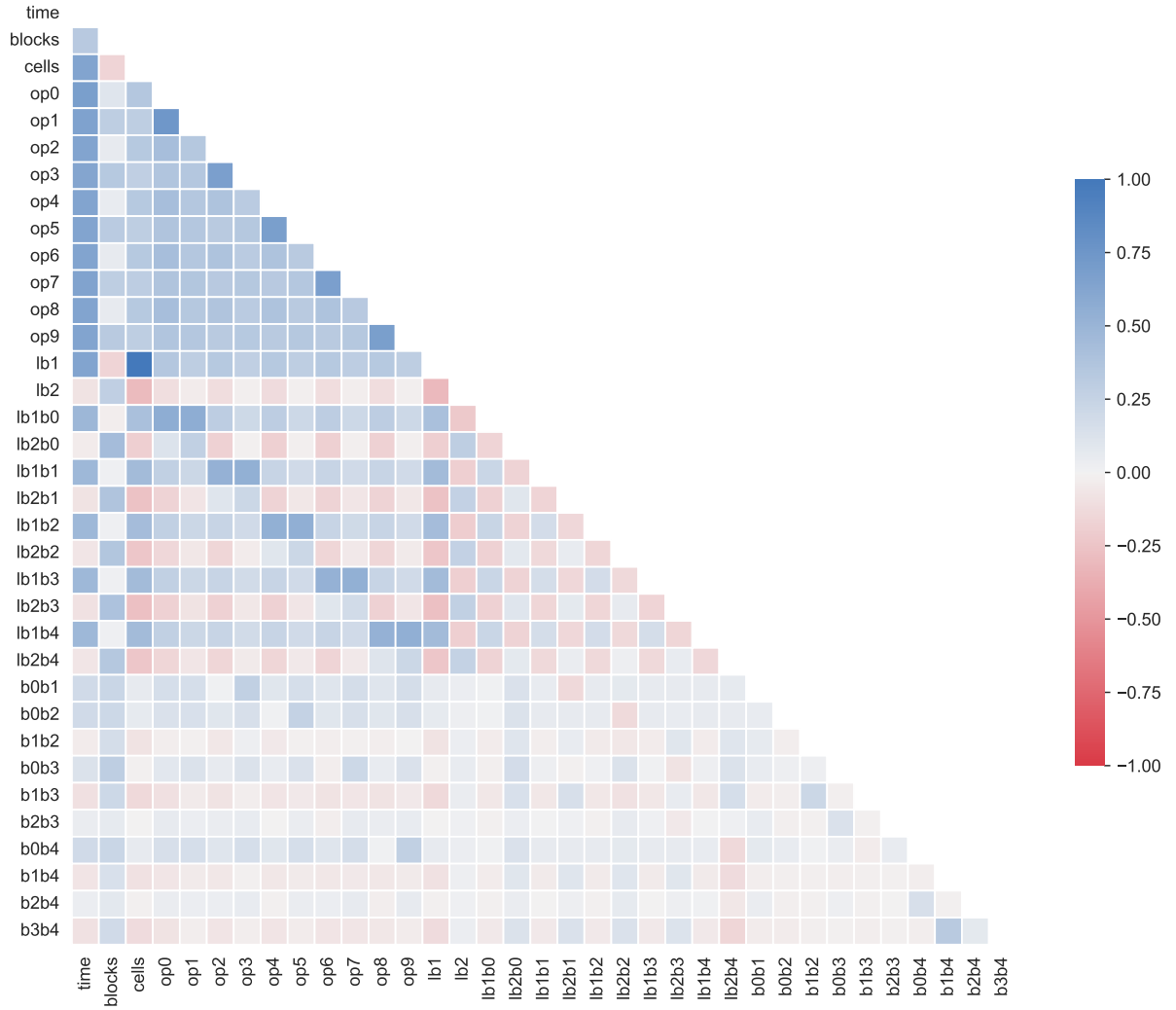


Figure 5.2: Feature correlation heatmap of the tentative feature set, produced on the data of a CIFAR10 run. Many features exhibit a low correlation with the output (time), highlighting a problem with the feature choice.

NASv2 configuration that stacks 8 cells to build each network, and a cell specification using only skip connections as lookbacks (-2 input value), the actual amount of cells stacked for that configuration is $\lceil \frac{8}{2} \rceil = 4$. The training time required scales almost linearly with the number of cells, so it is really important to explicit this feature.

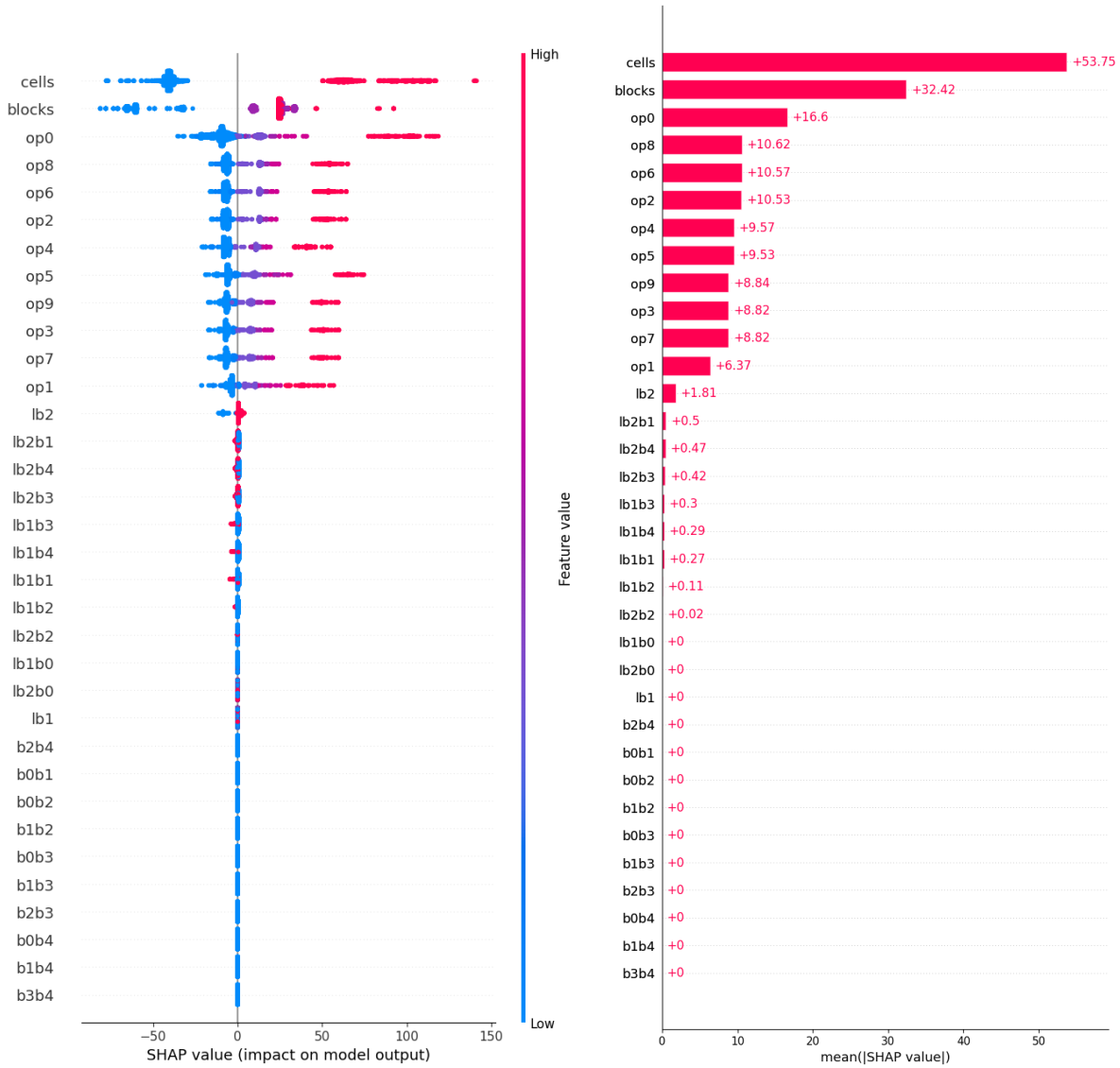
To test the efficiency of the new feature set, CatBoost has been trained on the results of a run on CIFAR10 dataset, using the old and the new feature sets. The predictor, before forecasting values for cells with b blocks, is trained on all data of cells with $b-1$ blocks, simulating the behavior of an actual POPNASv2 run. As seen in Fig. 5.1, the new feature set achieves a much lower mean average percentage error (MAPE) and a better Spearman rank coefficient (ρ), except for $b = 2$. It has been proved to also improve the average error

on other runs. Even after these changes, some problems pointed out for old feature set persists in the new one, like the fact that the number of features scales with B and that the algorithm needs to generate multiple records for each architecture to generalize on the features, since they are still positional and correlated together. Furthermore, the mean average percentage error is still significantly high, causing inaccuracies in the application of the optional time constraint T.

By performing some feature analysis techniques, it is possible to investigate in details the causes of the errors in the predictions. From the features correlation heatmap, shown in Fig. 5.2, it is possible to notice that the new features set is heavily bloated with features not important for the output, since they have low correlation with the training time. Also, having a large amount of features is not beneficial since it is easier for the model to learn an incorrect behavior.

SHAP [23] tool have been used to further analyze the feature importance, generating additional metrics useful for identifying the quality of these features. The data of a completed run on CIFAR10 dataset has been used for the analysis, retraining the CatBoost model with the SHAP tool attached to extract relevant information. By plotting a beeswarm plot and importance plot with SHAP (Fig. 5.3), it is possible to further investigate the issues. The information shown is relative to the last CatBoost training, when the largest amount of data is available, to minimize the uncertainty of predictions due to eventual lack of data.

Positional features not only introduce a difficult to learn correlation, but also fails to generalize properly since the model tends to not give a similar importance to them. In fact, at least the operator features should have an almost identical impact on the results, since the complexity of the operator does not change with its position in the cell encoding. From SHAP results instead is evident that the operator importance varies a lot between each feature, ranging from 6.37 to 16.6. Furthermore, the majority of features related to the input incidence matrices have a negligible impact on the output, confirming that the sparse feature set is not beneficial for improving the results.



(a) SHAP feature impact on output (beeswarm)

(b) SHAP feature importance

Figure 5.3: SHAP graphs about the importance of each feature of the new tentative set. CatBoost have been run on the data of a previously completed search run on CIFAR10, training it on all cell specification with $b \leq 4$. This setup simulates the last training of CatBoost during the actual run, using all available data prior to last training.

5.3.3. Definitive feature set

After the feature analysis had been performed, it was clear that the high errors seen in training time estimations were due to the problems identified in the tentative feature set and not due to the difficulty of the problem. A new feature set have been engineered from scratch, trying to satisfy these important conditions:

- extract the features directly from the DAG structure generated from the cell specification, instead of the abstract cell specification. This change allow to implicitly generalize on the equivalent cells, without the need of data augmentation, since equivalent cell encodings would produce the same features. This approach also highlighted the architectural changes done by the model generator to adapt the cell for the final neural network structure, that justified the actual time gap between similar cell encodings.
- generate a limited fixed amount of features. Making the size of the feature set independent of the POPNAS search space configuration makes it more robust to different experiment setups.
- avoid positional features, merging the ones that should have the same impact on the output. This change address in particular the disparity in operators importance seen in Fig. 5.3.

The new feature set, referred to as *definitive set*, is composed by the following features:

- number of blocks
- number of cells
- the sum of the dynamic reindex value (OP score) of each cell block operator
- number of concatenated tensors in cell output
- usage of multiple lookbacks (boolean)
- cell DAG depth (in blocks)
- number of block dependencies
- % of the total cell OP score related to the heaviest cell path
- % of the total cell OP score related to the blocks using lookbacks as input.

To explain better how these features are extracted from the cell it is useful to visualize the DAG of a real cell specification, for example a cell with 3 blocks:

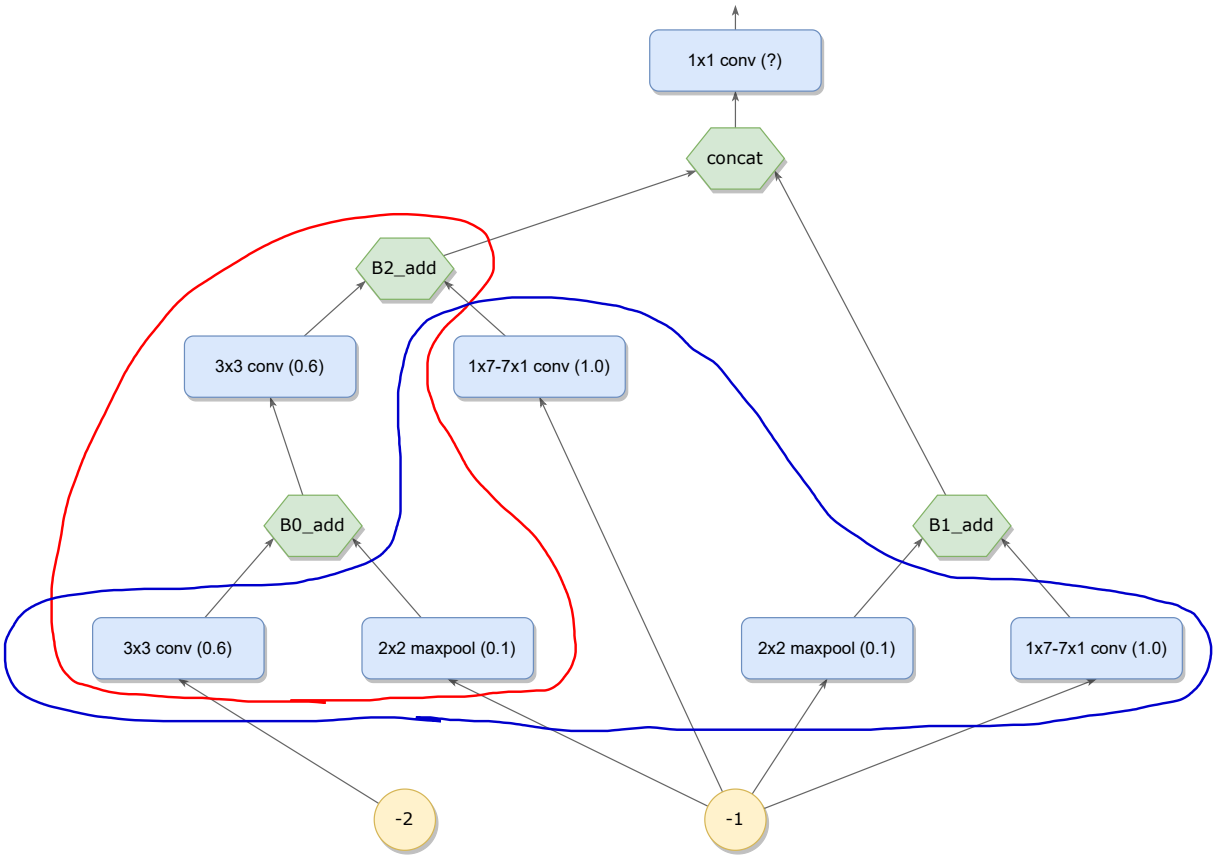


Figure 5.4: Cell DAG of the example cell specification c, used as example to describe how the features are extracted from the actual cell structure. The OP score associated to each operator is written in the round brackets. The heaviest cell path is highlighted in red, while the set of operators using lookback values is highlighted in blue.

$$c = \begin{matrix}[(\text{-2, 3x3 conv, -1, 2x2 maxpool}),\\ (\text{-1, 2x2 maxpool, -1, 1x7-7x1 conv}),\\ (\text{0, 3x3 conv, -1, 1x7-7x1 conv})].\end{matrix}$$

The DAG is shown in figure 5.4.

The OP score is a property exclusive to the operators, depicted as blue boxes in the graph. The OP score is assigned to each $o \in \mathcal{O}$ through the dynamic reindex formula. The yellow circles, -2 and -1, are the utilized lookback inputs, while the green hexagons are the join layers: addition for blocks and concatenation for the cell output.

The first important thing to notice is that the concat operation is followed by a pointwise convolution, used to reduce the filters to the value expected by the next cell. The number of filters of this convolution scales with the amount of concat inputs. Since the time required to perform a convolution is proportional to the amount of filters, this is a key

Table 5.3: Time features extracted from DAG of cell example c.

blocks	cells	OP score	concat inputs	multiple lookbacks	DAG depth	block deps	% OP score in heaviest path	% OP score using lookbacks
3	8	3.4	2	1	2	1	$1.3/3.4 = 0.38$	$1.8/3.4 = 0.53$

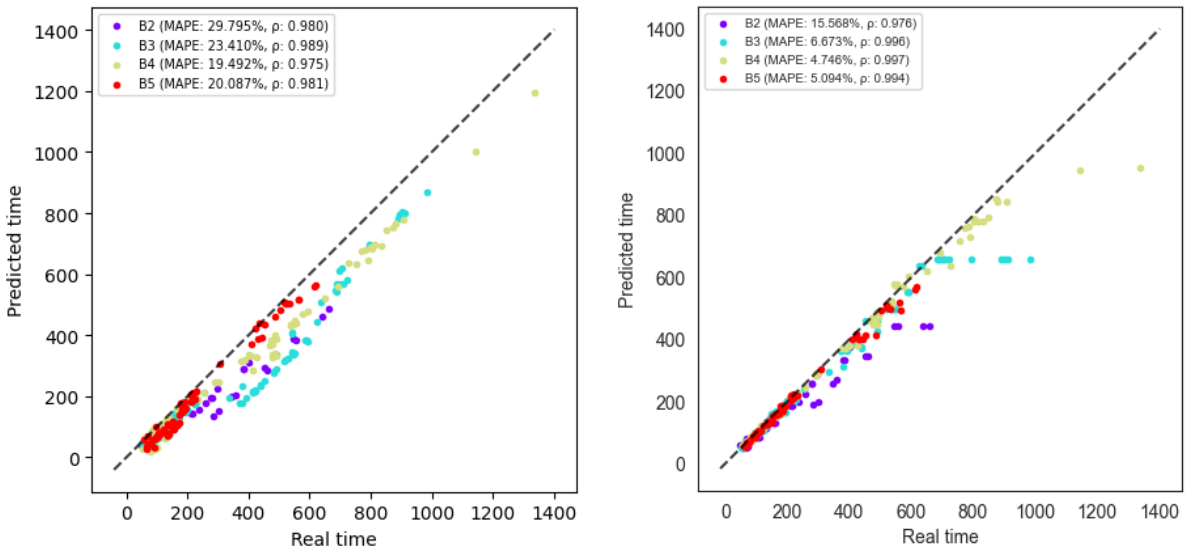
feature for estimating the training time, missing in previous feature sets. If there is only one block output not utilized inside the DAG, then the concat + pointwise convolution is entirely skipped, reducing the training time required; in this case the concat inputs feature is set to 0, since it is not present.

The "multiple lookbacks" feature is set to 1 if different lookbacks are used inside the cell specification. If this is the case, a normalization of the inputs must be performed when one lookback is the output of a reduction cell, since the other would come from a normal cell and therefore they would have different shapes. The normalization is performed with a pointwise convolution, that adds a small overhead to training time.

Regarding the cumulative OP score, it is simply computed as the sum of the OP score of all operators in the actual cell specification. The pointwise convolution after the concatenation is not considered in the OP score, using instead the number of concatenated tensors in cell output to estimate its time impact.

The heaviest path of the cell DAG is the path leading to an add converging in concat (or the final cell output) with the most op score involved. It is highlighted in red in Fig. 5.4. The score includes all the predecessor operators of the last one in the path. The feature value is the percentage of the cumulative OP score contained in this path. This feature can be important to give a tentative metric on how much efficient is the parallelization of multiple operators.

Similar thing is done for the operators using lookbacks as inputs, highlighted in blue in Fig. 5.4. In reduction cells the operators using lookback inputs must perform a (2,2) stride with also a duplication of filters on the output. This feature captures this behavior, which can lead to minimal time discrepancies due to the fact that these operators perform fewer FLOPS than the others using blocks as hidden layers. In fact, due to the stride, the filters are applied the same amount of times on the tensor even if the spatial resolution is double compared to the output tensor used in other operators, but the amount of filters is halved, therefore these computations require half the time compared to performing them on the internal hidden layers of the cell. Block dependencies are simply the number of times an input of a block is produced from another block. The other features are quite intuitive to understand. The features extracted from the example case are shown in Tab. 5.3.



(a) precedent feature set (b) definitive feature set

Figure 5.5: Comparison of CatBoost results, trained using the tentative and definitive feature sets.

To verify the quality of the new feature set, a comparison test has been performed between the prediction values provided by CatBoost trained on the precedent tentative feature set and the definitive one. CatBoost have been trained on the data gathered on a single search run on CIFAR10. The comparison is shown in Fig. 5.5. The new feature set has more accurate results, reducing the MAPE in all search steps and further boosting the ranking accuracy. The results confirm that the issues highlighted in the feature analysis were the cause of the poor quality of the time predictor.

Furthermore, the time required to train a CatBoost model is at least one order of magnitude faster than previous feature configuration, since both the amount of samples and the number of features have been drastically reduced. Indeed, the new feature set does not require the data augmentation of the samples, because it implicitly generalizes on the equivalent cell encodings and has no positional features, thus making useless the sliding block mechanism introduced in the first version of POPNAS. The definitive feature set, given its significantly better results compared to other tested solutions, also across different search runs, is the default time feature set used in all future test and in the experiments described in later sections.

Dataset	MAPE(%)				Spearman(ρ)			
	b=2	b=3	b=4	b=5	b=2	b=3	b=4	b=5
NNLS	24.525	16.168	12.712	13.531	0.776	0.984	0.990	0.989
Ridge	26.852	23.648	7.793	6.326	0.827	0.993	0.991	0.987
XGBoost	24.419	9.073	9.797	5.554	0.977	0.996	0.953	0.986
CatBoost	23.571	6.298	7.668	2.852	0.988	0.996	0.990	0.992

Table 5.4: Time regressor comparison on CIFAR10 results, using the new feature set.

5.4. Time predictors

In POPNASv2, the NNLS time predictor has been replaced with a CatBoost regressor. The introduction of the new feature set, extrapolated from the comparative analysis of the network architectures and their results, has further improved the results, as explained in Section 5.3.3. Changing the feature set is a major change that could lead to interesting and improved results also for other regressor models. To make sure that CatBoost was still the best time predictor for POPNASv2 search strategy, all the regressor models considered in the first version of POPNAS have been tested on the same data, using the definitive feature set.

Tab. 5.4 shows the results of the time predictors experiment. CatBoost is still the best model regarding both accuracy and ranking, with XGBoost [2] following closely. NNLS has a quite good ranking except for $b = 2$ step, but suffers also from a significantly average error. This test confirms that CatBoost is still the most suited model to forecast the required training time required for training the cell specifications selected by the search strategy.

6 | Experiments and results

This section presents the experiments conducted to evaluate POPNASv2. The results are compared with PNAS, since it is the baseline of the work. Since there is no open-source version of PNAS search method, the tested version is derived by stripping the time predictor, Pareto front generation and exploration step from POPNASv2 method. In order to achieve a fair performance comparison, both algorithms have been set up with the same search space. The control step for pruning equivalent models is instead maintained for both algorithms. The hyperparameters common to both POPNASv2 and PNAS have the same values. The experiments have been carried out on a NVIDIA A100 GPU, using MIG 3g.20gb profile.

6.1. Experiments setting

The experiments conducted to validate the generalization capabilities of POPNASv2 efficiency improvements have been carried out on four different datasets for image classification, i.e., CIFAR10, CIFAR100 [15], Fashion MNIST [37] and EuroSAT [10]. These datasets differ in the number of classes, the number of channels, and the images dimensions, providing a robust benchmark for the generalization capabilities of the search strategy. A summary of the dataset characteristics is provided in Tab. 6.1.

All the datasets share the same settings for preprocessing and data augmentation. In the preprocessing step, all input channels are normalized in [0, 1] range, and the samples are split into training-validation sets, respectively containing 90% and 10% of the total training samples. The default batch size is 128. We use random horizontal flip and random translation on both height and width for data augmentation, with a range of 0.125 as the actual input size.

The configuration of the search algorithm is the same for all these datasets. The architectures are trained on the training set, measuring the total training time t and keeping the validation set to gather the validation accuracy a, used in the performance estimation strategy. The architectures are trained on $E = 21$ epochs, using AdamW [22] with cosine

Table 6.1: Summary of relevant characteristics of the datasets chosen for the tests.

Dataset	# classes	Spatial dim	# channels	Training samples
CIFAR10	10	32x32	3 (RGB)	50k
CIFAR100	100	32x32	3 (RGB)	50k
Fashion MNIST	10	28x28	1 (grayscale)	27k
EuroSAT	10	64x64	3 (RGB)	60k

decay restart [21], starting learning = 0.01, starting weight decay = $5e^{-4}$, $T_0 = 3$ and $T_{mul} = 2$, achieving a total of 3 complete cosine decay restart periods.

Swish [28] activation function is used instead of ReLU since it provided an accuracy boost to the trained architectures both in POPNASv2 and PNAS. Each training step after $b = 1$ trains at max $K = 128$ architectures. The maximum number of extra architectures selected in exploration step Ex is set to 16. The neural networks produced starting from each cell are composed as a stack of 3 architectural motifs, with $N = 2$, for a total of 8 stacked cells. This is the same structure also used in PNAS and NASNet.

The accuracy predictor is built as an ensemble of 5 LSTM using Attention. Each LSTM is trained for 30 epochs on $\frac{4}{5}$ of the available data using Adam [14] optimizer, with $lr = 4e^{-3}$ and L2 weight regularization with factor $1e^{-5}$. The embedding size used is composed of 10 units, the Conv1D filters are 16, and the cells used in each LSTM of the bidirectional are 48.

For each dataset, the time predictor is implemented with a Catboost Regressor model, tuned via the random search hyperparameters optimization already provided in its library. CatBoost hyperparameter space is defined as follows:

$$\begin{cases} \text{learning\_rate:} & \text{uniform(0.02, 0.2)} \\ \text{depth:} & \text{randint(3, 7)} \\ \text{l2\_leaf\_reg:} & \text{uniform(0.1, 5)} \\ \text{random\_strength:} & \text{uniform(0.3, 3)} \\ \text{bagging\_temperature:} & \text{uniform(0.3, 3)} \end{cases}$$

Each Catboost model is trained for 2500 iterations in a 5-fold fashion, using early stopping with patience set to 50 iterations. The final model is retrained from scratch on the entire dataset, using the best hyperparameters configuration found.

Table 6.2: The results of POPNASv2 accuracy predictor for each evaluated dataset.

Dataset	MAPE(%)				Spearman(ρ)			
	b=2	b=3	b=4	b=5	b=2	b=3	b=4	b=5
CIFAR10	2.886	3.505	1.374	1.704	0.678	0.95	0.889	0.915
CIFAR100	13.997	3.645	3.382	5.409	0.509	0.75	0.814	0.934
Fashion MNIST	2.296	0.891	1.067	0.861	0.798	0.875	0.84	0.928
EuroSAT	0.446	0.636	0.552	0.564	0.739	0.688	0.823	0.922

Table 6.3: The results of POPNASv2 time predictor for each evaluated dataset.

Dataset	MAPE(%)				Spearman(ρ)			
	b=2	b=3	b=4	b=5	b=2	b=3	b=4	b=5
CIFAR10	23.571	6.298	7.668	2.852	0.988	0.996	0.99	0.992
CIFAR100	21.661	9.202	6.881	3.709	0.964	0.975	0.951	0.989
Fashion MNIST	23.45	11.916	4.884	8.706	0.919	0.983	0.99	0.982
EuroSAT	25.721	9.031	9.913	5.937	0.963	0.974	0.987	0.97

6.2. Predictors results

POPNASv2 relies on the time and accuracy predictors to build an effective Pareto front. The domination rule compares the estimated accuracy and training time of two cells; therefore, it is extremely important to rank precisely the estimated values. Spearman's rank correlation coefficient (ρ) is used to measure the quality of the ranking of both predictors. The accuracy of the predictors (measured with MAPE) is less relevant than high-quality ranking. However, it is preferred to enable the time constraint T and minimize the pruning errors given by this constraint.

The results of POPNASv2 predictors over the four datasets are summarized in Tab. 6.2 and Tab. 6.3. MAPE and Spearman values are computed on the unseen data, e.g., the columns $b = 4$ refer to the values forecasted after the predictors have been trained on the information related to cells with $b < 4$ blocks.

In general, both predictors have less accurate results on $b = 2$ since the data available is fewer and the cells with $b = 1$ do not exhibit behaviors proper of multi-blocks cells. In particular, cells constituted of multiple blocks could perform a concatenation operation followed by pointwise convolution at the end of the cell, used to join the outputs in case there are multiple unused block outputs. The presence of this extra pointwise convolution has a significant impact on the training time, leading to the error bias seen in the time predictor for $b = 2$. Moreover, some blocks could use other blocks as hidden layers to build a multi-level cell DAG, which is not possible in $b = 1$. These structural changes can

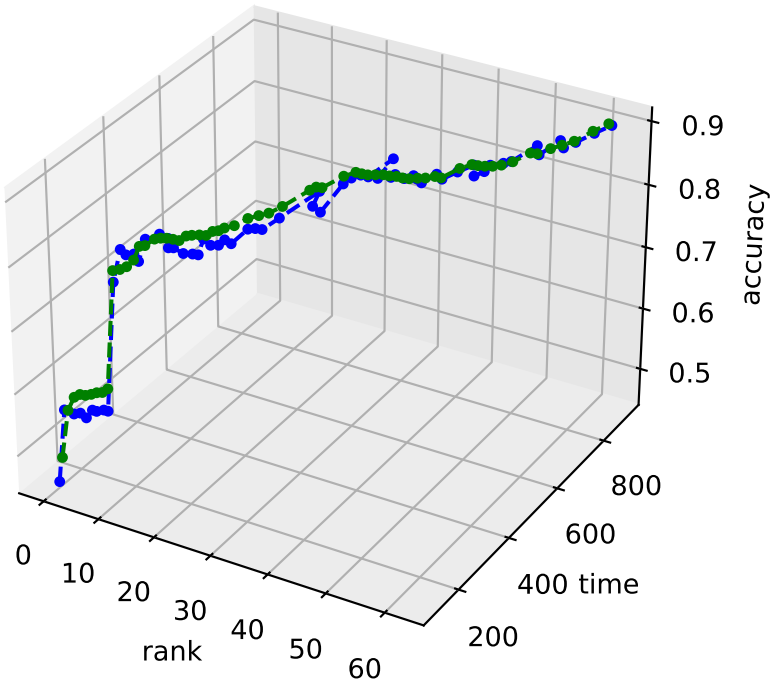


Figure 6.1: The Pareto front trained for $b = 5$ in CIFAR10 dataset. The predicted values $\hat{a}, \hat{t}$ are plotted in green, while the blue points represent the actual a, t retrieved after training. The rank values represent the ordering of these points in the Pareto front.

alter the trend exhibited by time and accuracy metrics in single block cells, making $b = 2$ the most challenging step for both predictors. With the increase of b, the average errors of the predictors tend to decrease, also improving the ranking quality.

The Spearman coefficients of both predictors tend to 1, so the predictors are ranking correctly the architectures, which is beneficial for POPNASv2 Pareto front methodology. From Fig. 6.1, which represents the Pareto front for $b = 5$ over CIFAR10 dataset, it is possible to notice the quality of predicted performance with respect to the real ones. It is possible to argue that the features selected for the predictors successfully allow POPNASv2 to build accurate multi-objective rankings, despite the low correlation between training time and accuracy commonly expressed by neural network models.

6.3. POPNASv2 vs PNAS

In this section is presented a comparison of the results of both POPNASv2 and PNAS search methods, considering the total time required to complete the search and the results of the top networks found. Tab. 6.4 shows a summary of the search results. The results

Table 6.4: The comparison between POPNASv2 and PNAS search performance over the evaluated datasets.

Dataset	Method	# Networks	Top1 Accuracy	Top5 Accuracy	Search Time
CIFAR10	POPNASv2	**543**	0.912	0.911	**49h24m**
	PNAS	814	**0.922**	**0.92**	176h30m
CIFAR100	POPNASv2	**548**	**0.685**	**0.684**	**53h37m**
	PNAS	814	0.68	0.679	161h47m
Fashion MNIST	POPNASv2	**537**	**0.947**	**0.946**	**49h50m**
	PNAS	814	0.946	0.945	174h53m
EuroSAT	POPNASv2	**549**	0.973	0.971	**69h35m**
	PNAS	814	**0.974**	**0.973**	380h23m

indicate that POPNASv2 is much more efficient than PNAS: with a 4.02x average time speed-up to complete the search, this new algorithm can achieve the same average top-1 and top-5 accuracy performance concerning datasets differing in the number of classes, in the number of channels and in the dimension of the images.

Contrarily to PNAS that always train K architectures in training steps with $b > 1$, POPNASv2 exploits the Pareto front pruning to drastically reduce the total number of trained networks up to the 33.14%. Furthermore, the time optimization performed by POPNASv2 significantly lower the average training time of the considered networks, leading to a significant time speed-up also in the case both algorithms would train an equal amount of networks.

Surprisingly, for CIFAR100 and Fashion MNIST datasets POPNASv2 can also produce architectures achieving slightly better accuracy compared with the best ones produced by PNAS. In general, the accuracy difference is at worse under one percentage point, confirming that POPNASv2 method can obtain remarkable gains from the accuracy-time trade-off. The cell structures found by the two methods have some significant differences. The figures shown in Appendix A.1 provides a visual comparison among the top-1 cells found over all the datasets tested.

POPNASv2 tend to organize cells into graphs composed of multiple levels, while PNAS cells are generally flat or almost flat. This behavior is imputable to the exploration step, which guarantees that the input values related to blocks are used in at least a small amount of networks, in each training step. The predictors can then adapt their results in following expansion steps, considering more these inputs values if they lead to good networks. PNAS instead does not guarantee the exploration of these input values, therefore the accuracy predictor tends to highly prefer inputs coming from other cells

Table 6.5: The comparison between POPNASv2 and PNAS top-1 networks performance over the evaluated datasets.

Dataset	Method	Params	B	Accuracy	Training Time
CIFAR10	POPNASv2	2.87M	4	0.929	**1h52m**
	PNAS	**2.36M**	5	**0.936**	4h6m
CIFAR100	POPNASv2	**2.27M**	5	**0.718**	**1h55m**
	PNAS	3.99M	5	0.711	3h19m
Fashion MNIST	POPNASv2	1.68M	4	**0.951**	**1h56m**
	PNAS	**1.41M**	4	0.95	3h4m
EuroSAT	POPNASv2	1.54M	4	**0.979**	**2h47m**
	PNAS	**1.47M**	5	**0.979**	11h34m

rather than from inner blocks.

Another difference is in the choice of the operators. POPNASv2 tends to discard the operators that heavily impact the training time on the device hardware, except when they have the best performance among the entire operator set. Separable convolutions give a quite counter-intuitive example of this behavior. Even if they have fewer parameters and FLOPS than normal convolutions, they impacted a lot more on the training time in all performed experiments. The inefficiency in parallelizing these operators, considering that they are built as a stack of two convolutional operations that must be performed sequentially, could be the cause of the observed behavior. Since PNAS is targeted in finding top accuracy networks without compromises and constraints, it tends to choose complex and time-consuming operators, even when they have negligible accuracy boosts over simpler and more performing operators. Appendix A.2 provides a visual comparison of the percentage of use of the values of both the inputs and the operators defined in the search space, for all the datasets considered, on step $b = 5$.

These results validate POPNASv2 strategy of forecasting and optimizing the training time of the selected architectures. Simply using metrics like FLOPS and network parameters, which can be directly computed from the neural network model without the necessity of a predictor, is not indicative of the training time required by a neural network model, when considering different operators. Forecasting time with surrogate model instead provides accurate results, making the algorithm hardware-aware, at the cost of the added complexity to perform these estimations.

The comparison between POPNASv2 and PNAS approaches has been further investigated by running an extended training session for each of their top-1 architecture, training them until convergence. In detail, the best cells found by PNAS and POPNASv2 have been retrained from scratch, on the same dataset on which they were found, changing

only the number of epochs E to 254 and reducing cosine decay restart T_0 to 2. This experiment provides accurate results on the final performance of the networks, using the same hyperparameter set used for the search method.

The results are summarized in Tab. 6.5. Even in this scenario, POPNASv2 is able to fill the average accuracy GAP with respect to PNAS. Concerning the top-1 networks training time, POPNASv2 achieved an average speed-up of 2.6x, despite the slightly higher parameters number. As discussed before, this behavior is caused by the time optimization, which favors the operators which can be performed efficiently on the used hardware.

Concerning the number of blocks, POPNASv2 tends to find top-1 cells composed by a fewer amount of blocks. A hypothesis of the cause of this behavior is that POPNASv2 cells tends to use more the internal blocks as hidden layers of other ones, since the exploration step can reliably introduce them during the search. Multi-level cell DAGs could be more effective than flat cells with multiple parallel paths, allowing POPNASv2 to reach competitive accuracies without the necessity of further increasing the neural network model complexity.

Summarizing the comparison, the experiment results assert that POPNASv2 can solve image classification problems by discovering simpler neural network architectures which achieve comparable accuracy performance with PNAS. The techniques introduced by this work guarantee remarkable searching and training time speed-ups, which is an important step forward to exploit NAS algorithms in real-world applications and datasets.

7 | Conclusions and future developments

Most of the state-of-the-art NAS algorithms are focused only on the search of top performance neural networks architectures, exploring incredibly huge search spaces that embeds the most suited operators for the analyzed task. Still, NAS works tends to neglect constraints and trade-offs usually imposed in real-world applications, like hardware requirements and economic costs to perform the neural architecture search. Addressing NAS as a multi-objective optimization problem allows the search strategy to be more adaptable to real world scenarios, making a step forward for the application of NAS technique outside the research field.

The thesis introduce POPNASv2, a sequential model-based optimization search strategy solving a multi-objective efficiency problem by building a time-accuracy Pareto front and exploiting its optimality properties. This work is an evolution of POPNAS, a previous student thesis of Politecnico di Milano, which exploited this methodology but achieved major accuracy drawbacks compared to other NAS methods. Two surrogate models are built to estimate the training time and accuracy of the cells while searching for expansions, allowing to prune suboptimal results. Implementing the time predictor as a CatBoost regressor model, paired with the time features re-engineerization, allows POPNASv2 to find effective architectures in a more efficient fashion, since these changes significantly boost the Pareto front accuracy. The adaptive greedy exploration step conditionally reintroduce operators and inputs discarded in previous training step, solving the potential convergence to suboptimal architecture structures due to early training bias. Comparing the results with PNAS, one of the most credited methods in the literature and baseline of this thesis work, POPNASv2 obtained almost a 4x speed-up in the search time and achieved similar accuracy in the top networks.

There are many possible directions to investigate for future developments. Regarding the search space, including Attention inside the operator set could lead to interesting results, since Transformer and convolution+Transformer networks have gained a lot of interest in

the latest years, achieving state-of-the-art result even in image classification tasks. It is worthwhile to try to further improve the predictors, since they are the fundamental for building accurate Pareto fronts, which are the core of this search strategy. Regarding the time predictor, CatBoost exhibits some outliers that could potentially alter the Pareto front. A different model or an ensemble could mitigate this issue. For the accuracy predictor, testing machine learning models could lead to competitive results and improve the ranking, but it could be complex to find a significant feature set to embed the operator importance on the results. Furthermore, the model builder can be expanded to generate architectures suited for tasks different from image classifications, e.g. image segmentation and text processing. More objectives could be integrated into the Pareto front, allowing the algorithm to find a set of networks suitable for different constraints, like memory utilization and inference time.

Bibliography

[1] G. M. Bender, P. jan Kindermans, B. Zoph, V. Vasudevan, and Q. Le. Understanding and simplifying one-shot architecture search. 2018. URL http://proceedings.mlr.press/v80/bender18a/bender18a.pdf.

[2] T. Chen and C. Guestrin. Xgboost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2016. doi: 10.1145/2939672.2939785. URL http://dx.doi.org/10.1145/2939672.2939785.

[3] P. Chrabaszcz, I. Loshchilov, and F. Hutter. A downsampled variant of imagenet as an alternative to the cifar datasets, 2017.

[4] X. Dong and Y. Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search, 2020.

[5] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey, 2019.

[6] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL https://proceedings.mlr.press/v9/glorot10a.html.

[7] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. volume 1 of *Foundations of Genetic Algorithms*, pages 69–93. Elsevier, 1991. doi: https://doi.org/10.1016/B978-0-08-050684-5.50008-2. URL https://www.sciencedirect.com/science/article/pii/B9780080506845500082.

[8] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. URL http://arxiv.org/abs/1502.01852.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[10] P. Helber, B. Bischke, A. Dengel, and D. Borth. Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification, 2019.

[11] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

[12] A. E. Hoerl and R. W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970. ISSN 00401706. URL `http://www.jstor.org/stable/1267351`.

[13] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In C. A. C. Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-25566-3.

[14] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

[15] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL `https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

[17] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

[18] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search, 2018.

[19] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search, 2019.

[20] E. Lomurno, S. Samele, M. Matteucci, and D. Ardagna. *Pareto-Optimal Progressive Neural Architecture Search*, page 1726–1734. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383516. URL `https://doi.org/10.1145/3449726.3463146`.

[21] I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts, 2017.

[22] I. Loshchilov and F. Hutter. Decoupled weight decay regularization, 2019.

[23] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017. URL `http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf`.

[24] M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[25] W. Mcculloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[26] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 978-0-07-042807-2.

[27] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin. Catboost: unbiased boosting with categorical features, 2019.

[28] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions, 2017.

[29] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search, 2019.

[30] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[31] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017.

[33] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016. doi: 10.1109/JPROC.2015.2494218.

[34] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. ISSN 00359246. URL `http://www.jstor.org/stable/2346178`.

[35] P. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. doi: 10.1109/5.58337.

[36] C. White, W. Neiswanger, and Y. Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search, 2020.

[37] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[38] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter. Nas-bench-101: Towards reproducible neural architecture search, 2019.

[39] K. Yu, C. Sciuto, M. Jaggi, C. Musat, and M. Salzmann. Evaluating the search phase of neural architecture search, 2019.

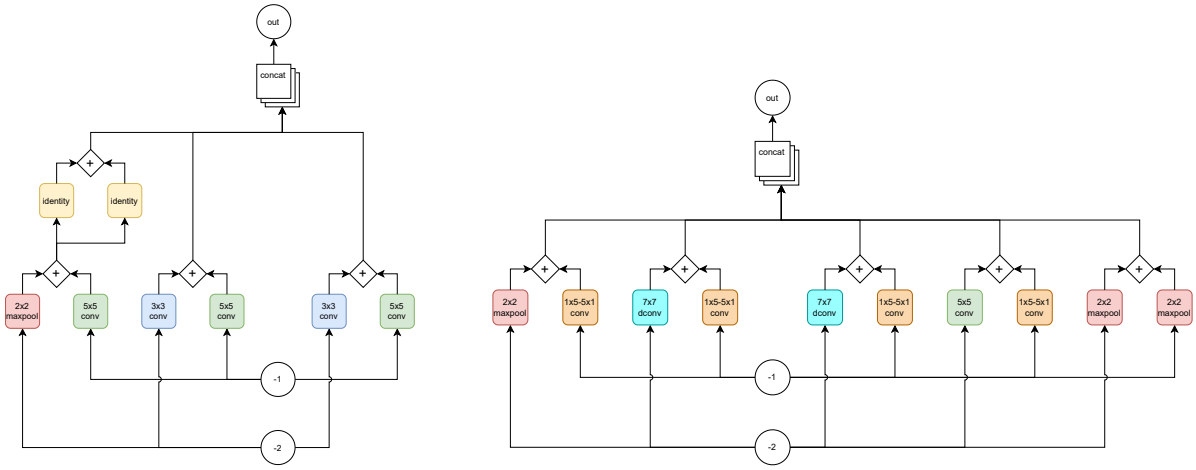
[40] A. Zela, A. Klein, S. Falkner, and F. Hutter. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search, 2018.

[41] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning, 2017.

[42] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition, 2018.

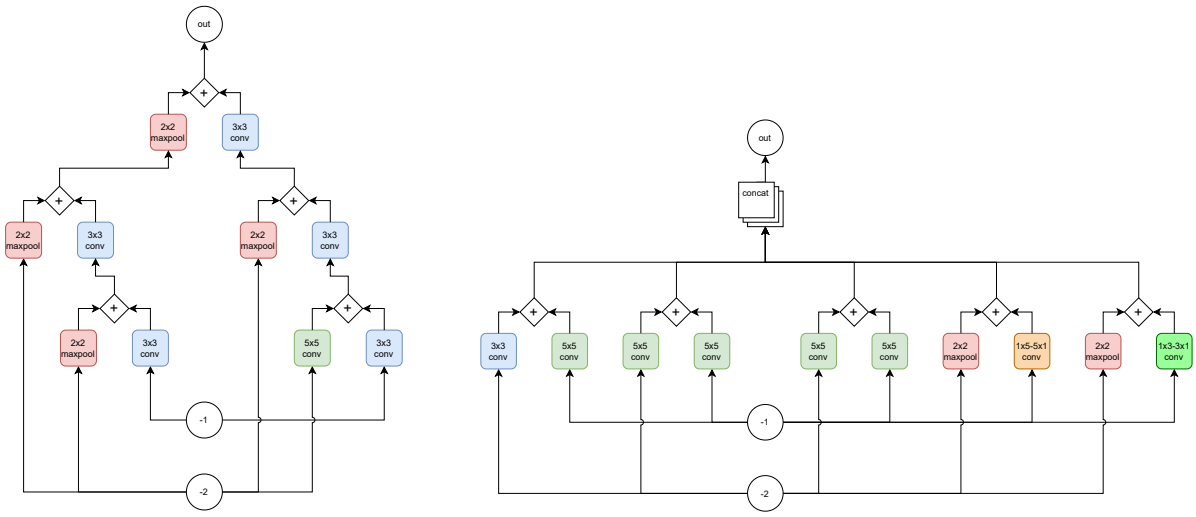
A | Appendix A

A.1. Top1 cells comparisons



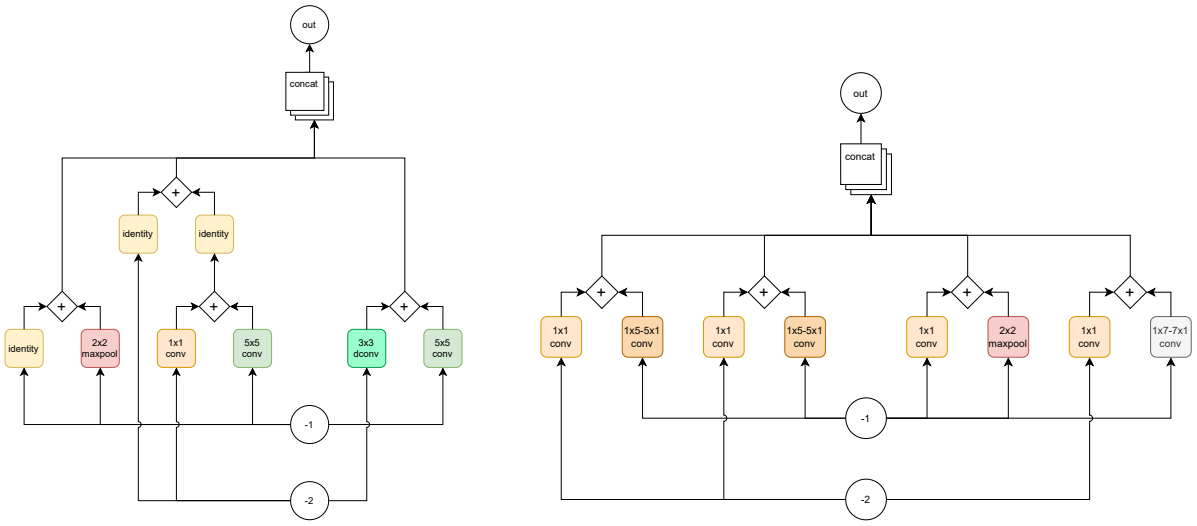
(a) POPNASv2 (b) PNAS

Figure A.1: Top1 cells found by POPNASv2 and PNAS on CIFAR10.



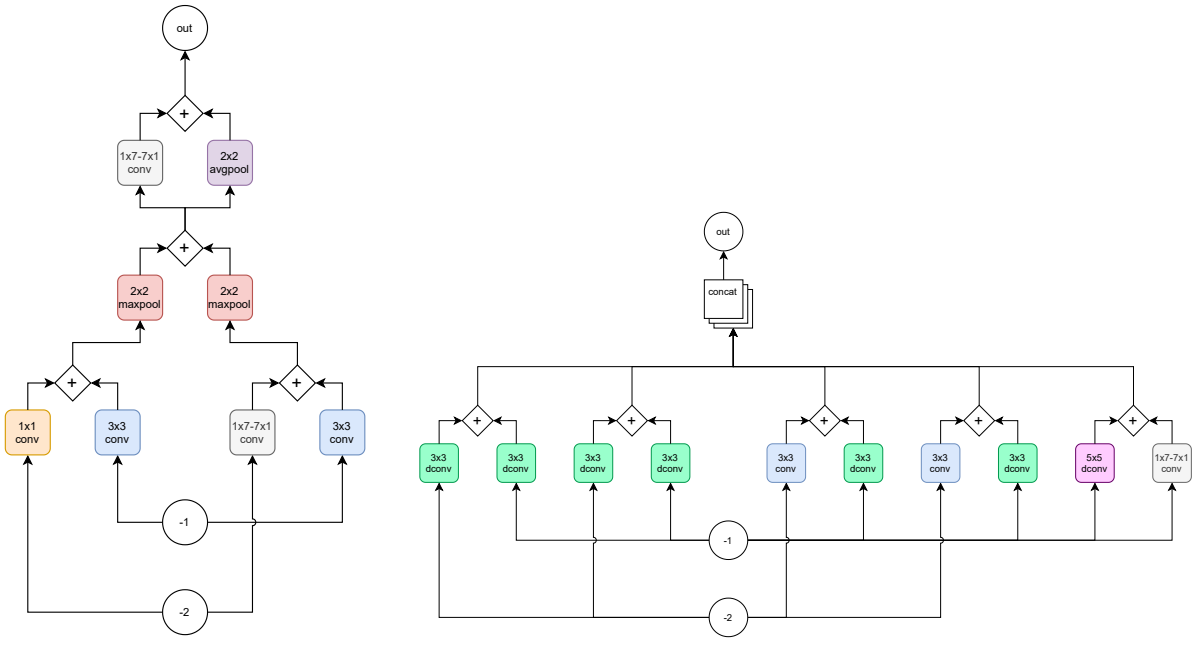
(a) POPNASv2 (b) PNAS

Figure A.2: Top1 cells found by POPNASv2 and PNAS on CIFAR100.



(a) POPNASv2 (b) PNAS

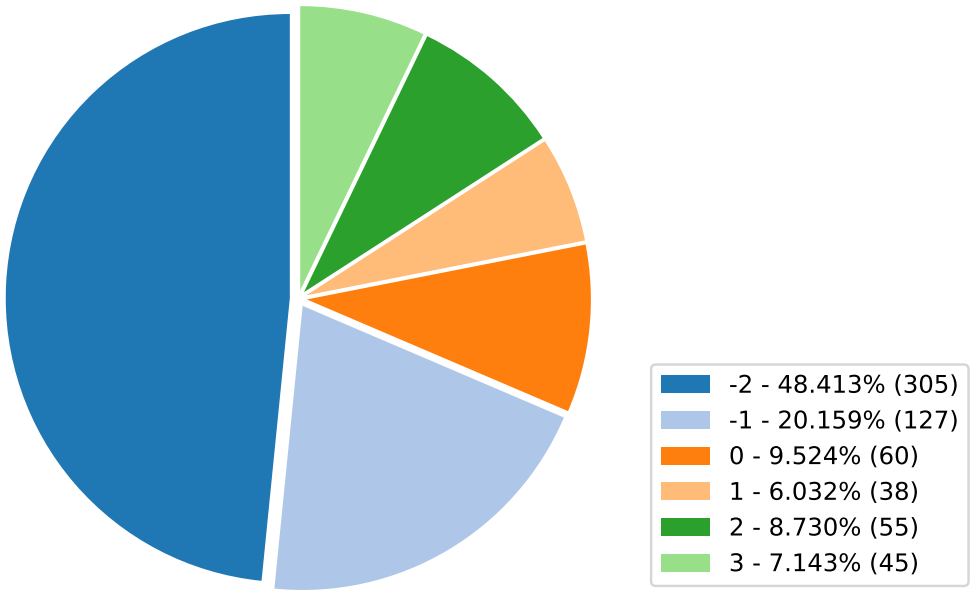
Figure A.3: Top1 cells found by POPNASv2 and PNAS on Fashion MNIST.



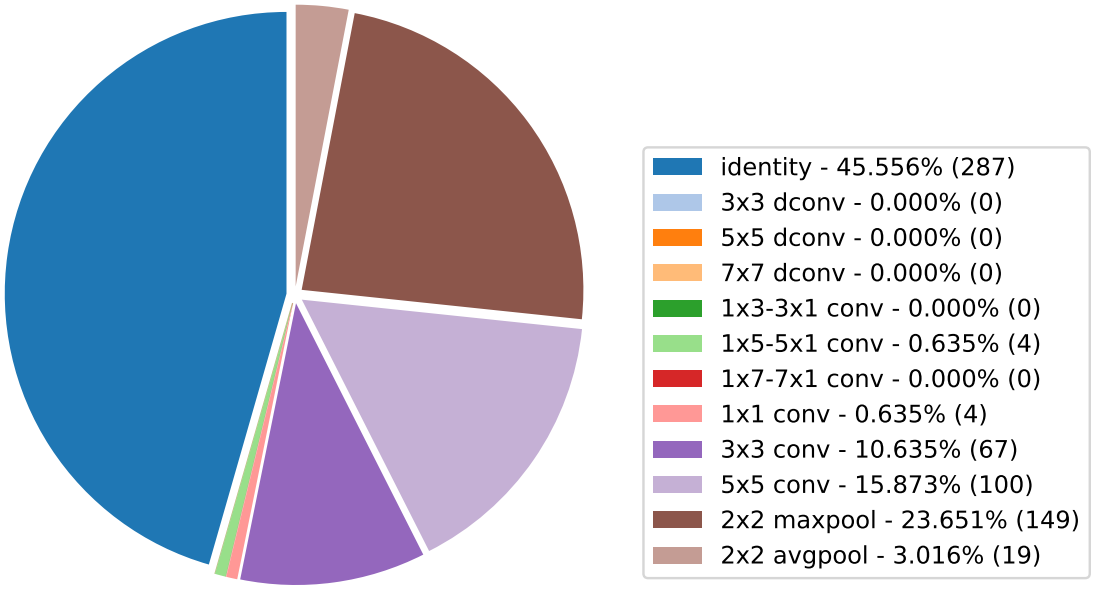
(a) POPNASv2 (b) PNAS

Figure A.4: Top1 cells found by POPNASv2 and PNAS on EuroSAT.

A.2. Inputs and operators usage comparison

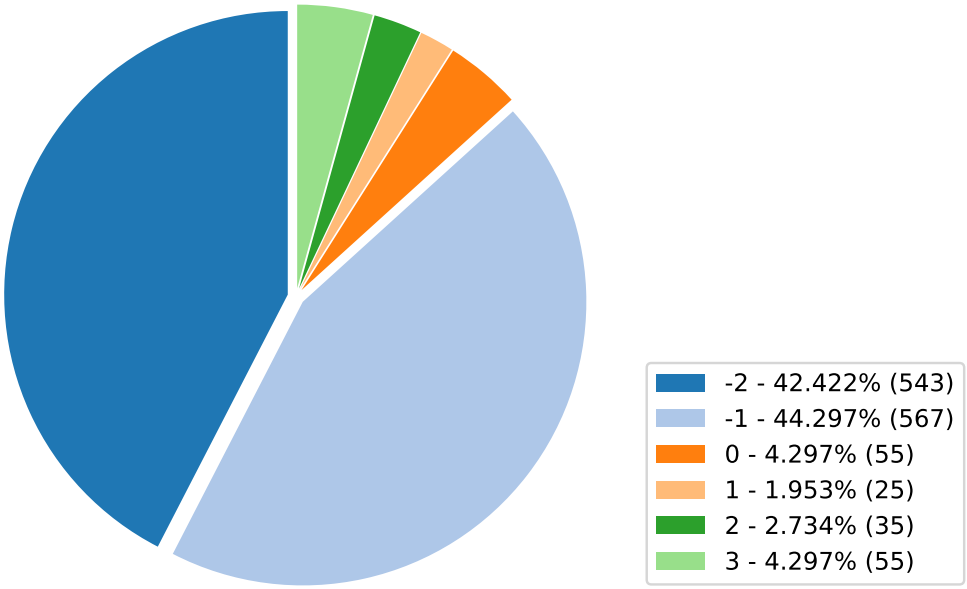


(a) inputs usage

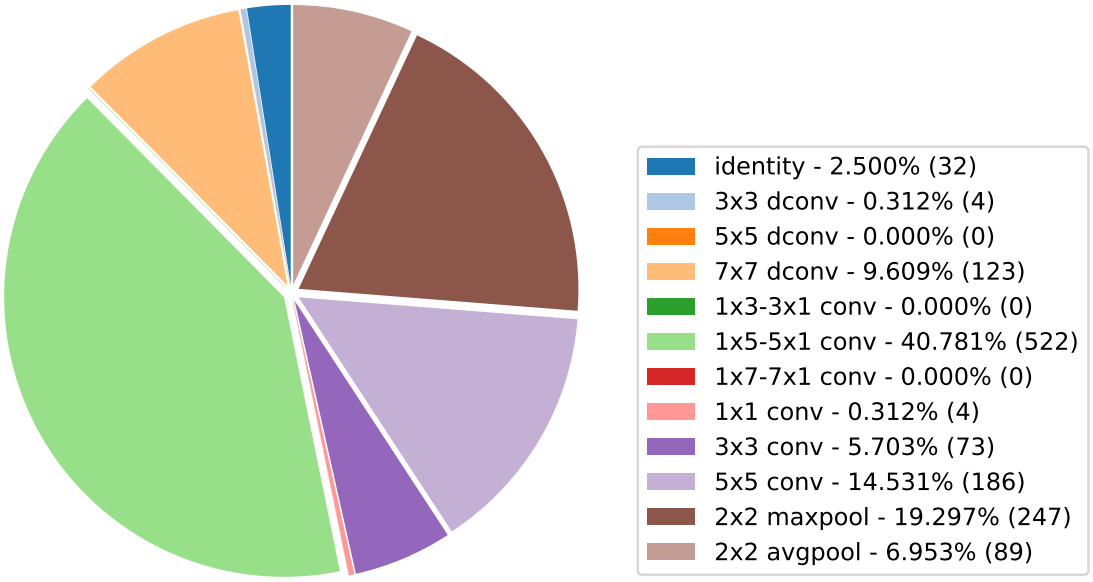


(b) operators usage

Figure A.5: POPNASv2 inputs and operators usage on CIFAR10, in step $b = 5$.

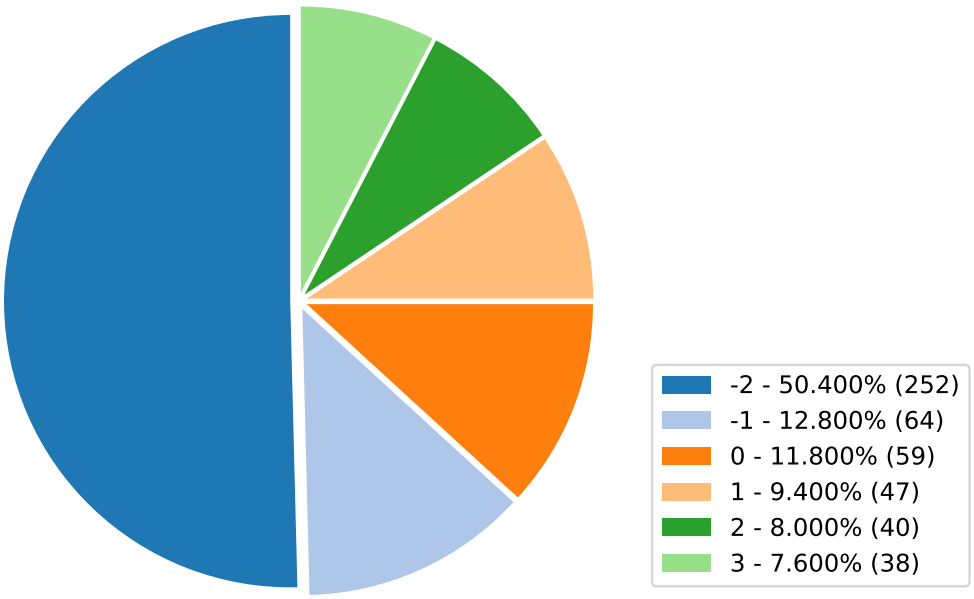


(a) inputs usage

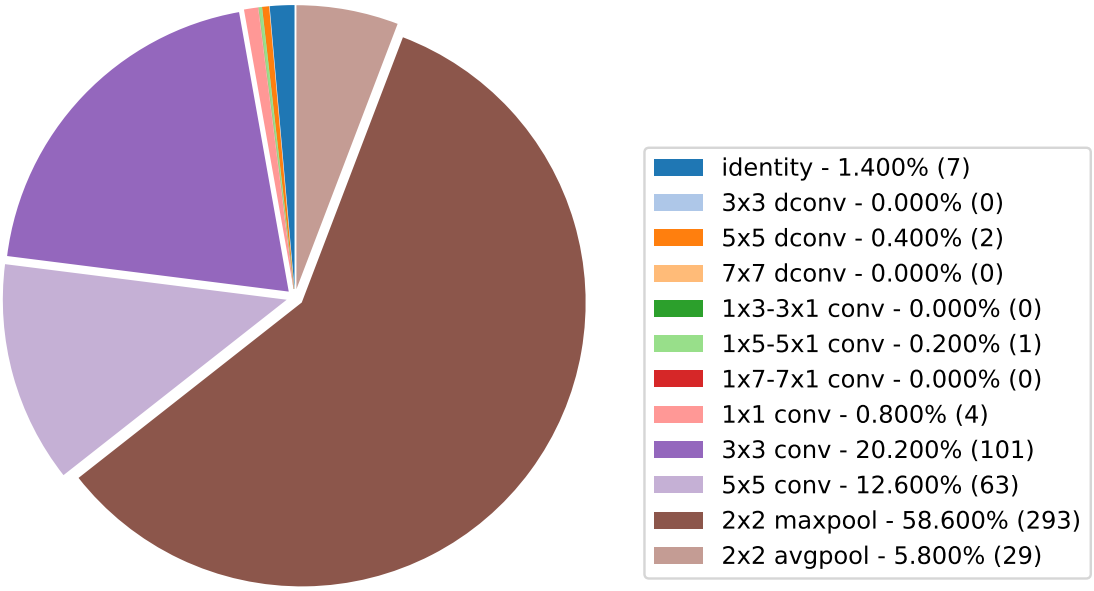


(b) operators usage

Figure A.6: PNAS inputs and operators usage on CIFAR10, in step $b = 5$.

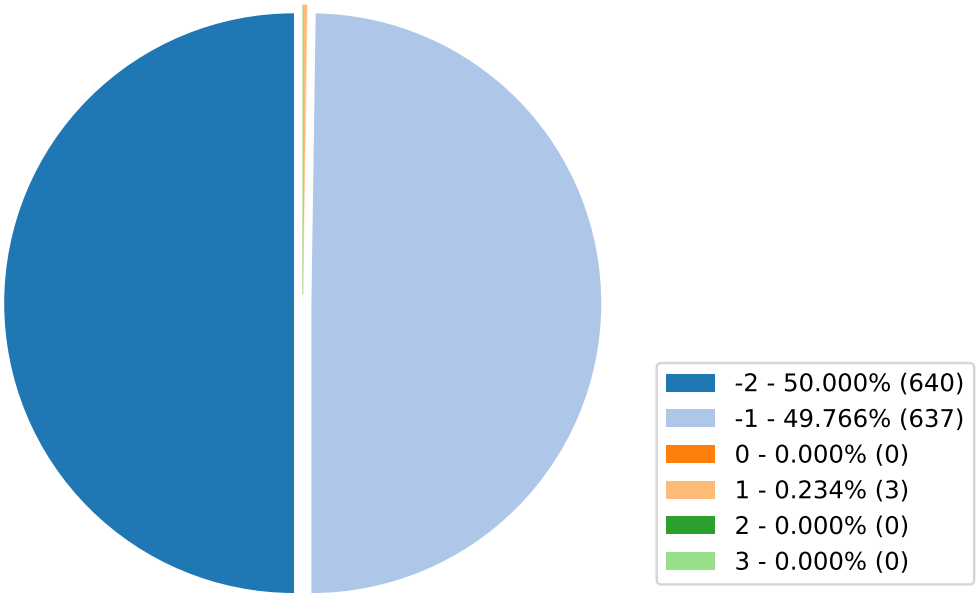


(a) inputs usage

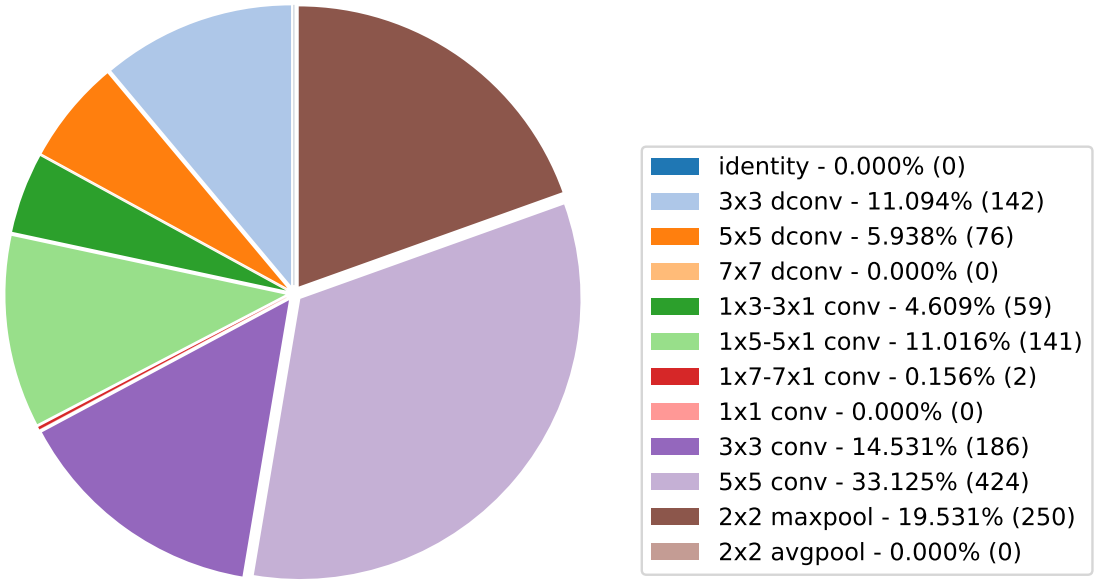


(b) operators usage

Figure A.7: POPNASv2 inputs and operators usage on CIFAR100, in step $b = 5$.

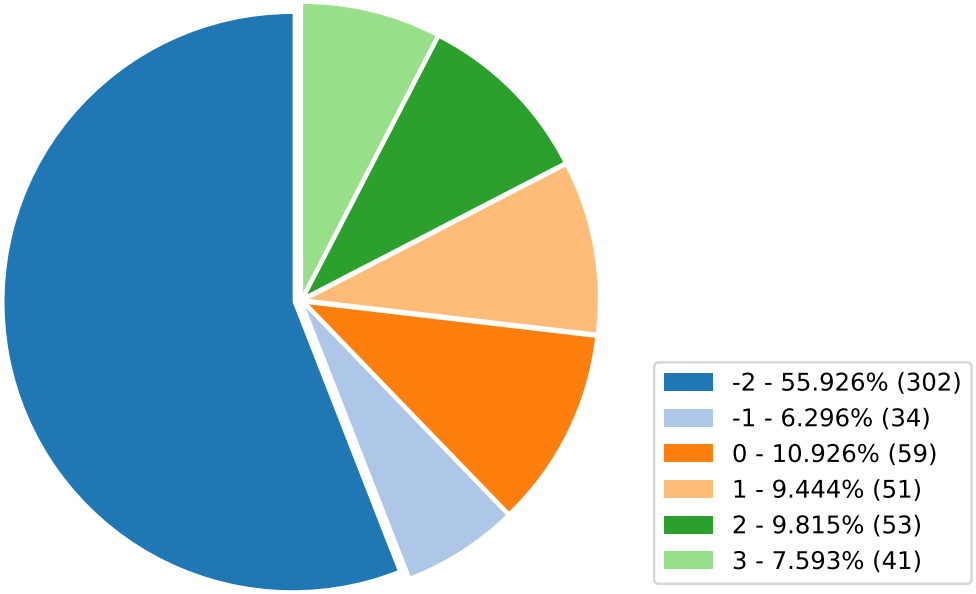


(a) inputs usage

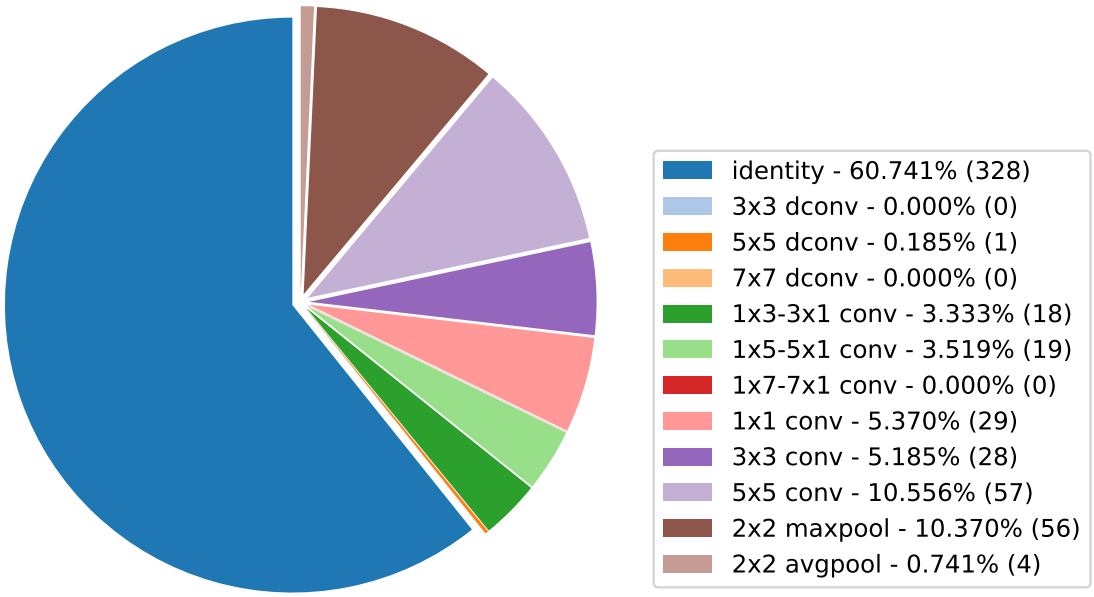


(b) operators usage

Figure A.8: PNAS inputs and operators usage on CIFAR100, in step $b = 5$.

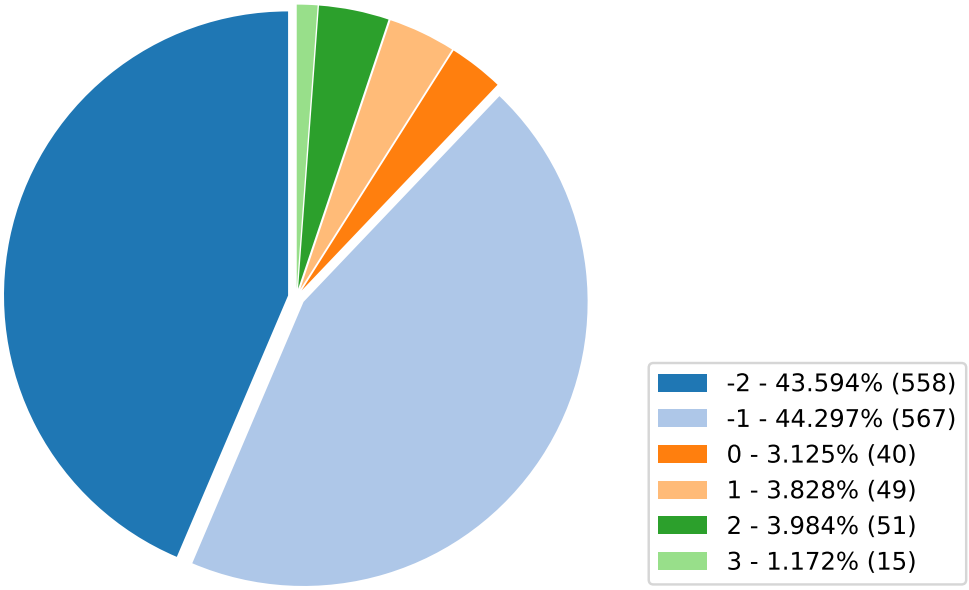


(a) inputs usage

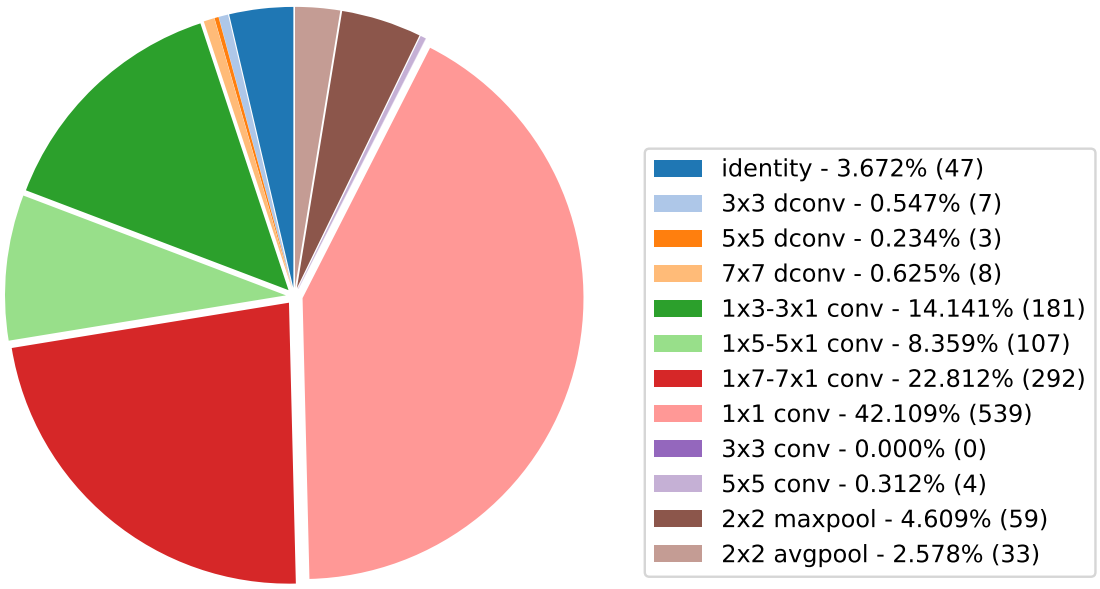


(b) operators usage

Figure A.9: POPNASv2 inputs and operators usage on Fashion MNIST, in step $b = 5$.

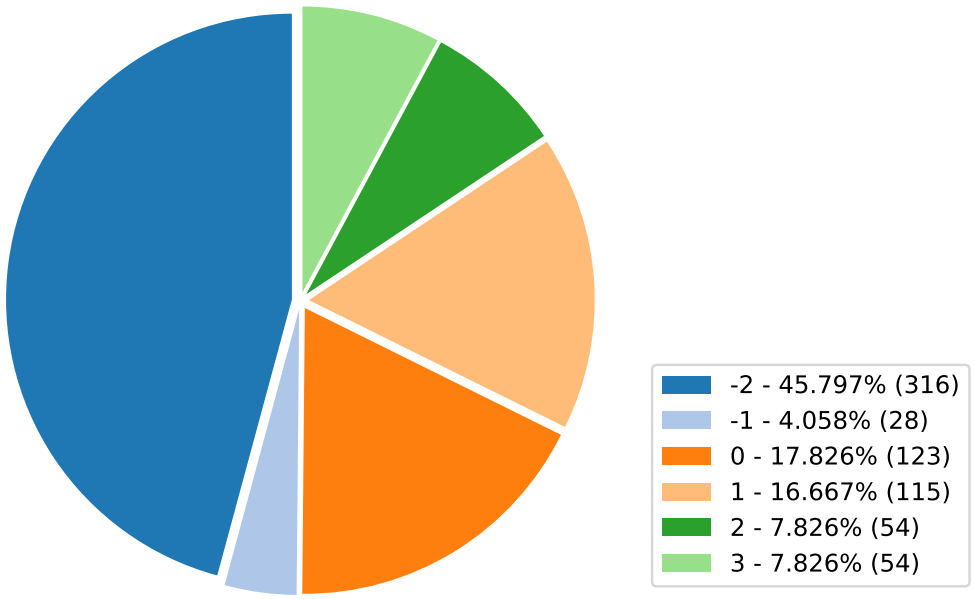


(a) inputs usage

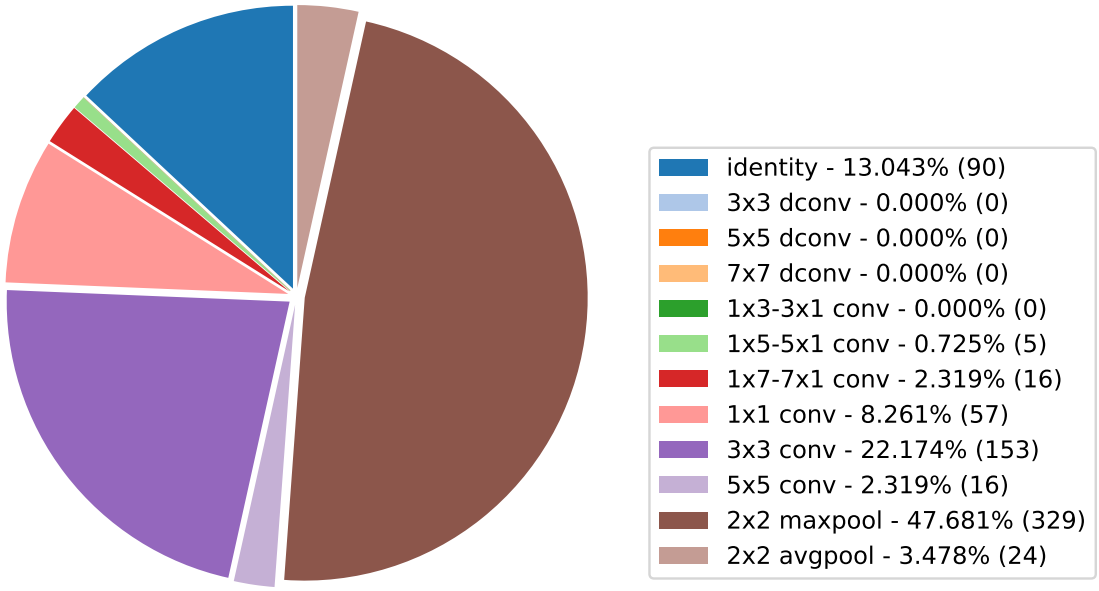


(b) operators usage

Figure A.10: PNAS inputs and operators usage on Fashion MNIST, in step $b = 5$.

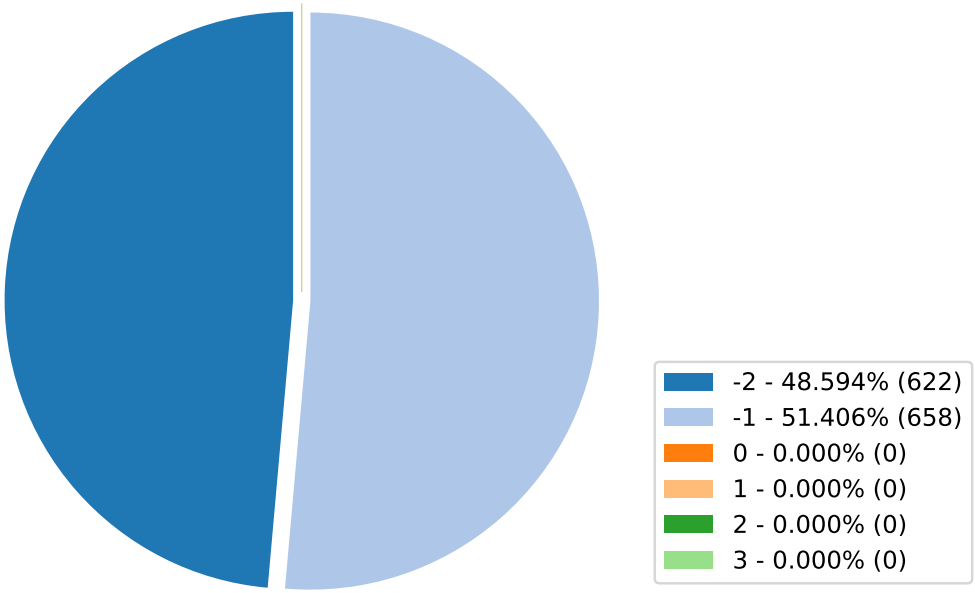


(a) inputs usage

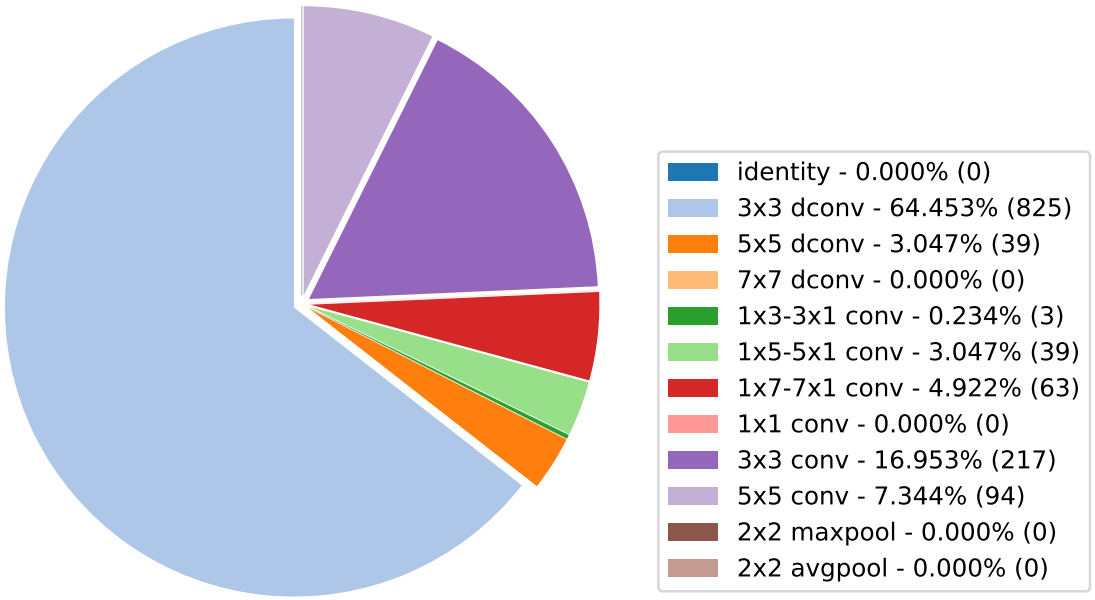


(b) operators usage

Figure A.11: POPNASv2 inputs and operators usage on EuroSAT, in step $b = 5$.



(a) inputs usage



(b) operators usage

Figure A.12: PNAS inputs and operators usage on EuroSAT, in step $b = 5$.

B | Appendix B

B.1. Software implementation

POPNASv2 is implemented in python, using Tensorflow 2 library and the Keras module to easily build and train the neural network models explored by the algorithm. The implementation is available at: `https://github.com/AndreaFalanti/popnas`.

The software is composed by multiple python modules, each one specialized in a set of tasks and functions required by the search method. This section provides a list of the main modules of the software, together with a brief description of their functionality, which can be useful to guide other fellow developers and researchers in investigating the source code.

run.py is the entry point of the algorithm, which parses the command line arguments, set ups the output folder and the resources. After the initialization step is complete, it starts the search procedure by calling the functionalities provided by *train.py*.

train.py contains the main logic of the search method, implementing the workflow of POPNASv2. This module initializes the instances of the other modules and coordinates them to perform the search, which starts with the initial thrust, followed cyclically by a training step, predictors update step and expansion step.

encoder.py handles the set of functions used to properly encode and decode a cell specification and to generate proper blocks for the selected search space. Cell encodings are a convenient way to store the information of the cell structure, from which the model generator can build the actual neural network model.

model.py implements the model generator, which purpose is to process any cell encoding to build a Keras neural network model from it. The model generated is built through Keras functional API, following NAS literature concepts. Each cell is composed by multiple blocks, where each operation is a custom layer defined in *ops.py*. The final model is just the composition of a stack of cells, followed by GAP and Softmax.

manager.py guides the training of each selected cell, calling the model generator to build

the actual neural network architecture and processing the training results. It also handles the Tensorflow dataset generation.

controller.py contains the logic for training both accuracy and time predictors, storing the final models to perform the performance evaluation strategy. It handles the logic required for the expansion and exploration steps.

plotter.py is a utility module which process the analytical data gathered during the search and generate plots from them, correlated with relevant metrics.

log-service.py is another utility module, that defines the logger format, its handlers and initializes the subfolders contained in the output log folder.

B.2. User guide

In this section it is briefly explained how to install and use the POPNASv2 software. For more details, refer to the README file present in the `repository`.

Dependencies

The project requires the following dependencies:

- python = ">=3.7.0, <3.9.0"
- tensorflow = "2.7.0"
- scikit-learn = "1.0.1"
- pandas = "1.1.5"
- tqdm = "4.62.0"
- matplotlib = "3.3.4"
- mypy = "0.910"
- catboost = "1.0.3"
- igraph = "0.9.8"
- tfds-nightly = "4.4.0-alpha.202201030107"
- shap = {git = "https://github.com/AndreaFalanti/shap.git"}
- pydot = "1.4.2"
- numba = "0.53.1"
- psutil = "5.8.0"
- seaborn = "0.11.2"
- tf2onnx = "1.9.3"
- keras-tuner = "1.1.0"
- tensorflow-addons = "0.15.0"
- tensorflow-probability = "0.15.0"

Note that SHAP is installed through my personal fork since the 0.40.0 version has a trivial bug in plot generation, which is fixed in the fork to simplify the installation.

If you want to use aMLLibrary to build the time predictor, you need these additional requirements:

- xgboost = "1.4.2"
- hyperopt = "0.2.5"
- mlxtend = "0.18.0"
- eli5 = "0.11.0"

If you are using CatBoost, which is the default option, you can skip the installation of these dependencies.

Installation

To set up the virtual environment and install the required dependencies locally it is advised to use *poetry*, a package manager for python available at `https://github.com/python-poetry/poetry`.

To use a GPU locally, you must satisfy Tensorflow GPU hardware and software requirements. Follow `https://www.tensorflow.org/install/gpu` instructions to set up your device, make sure to install the correct versions of CUDA and CUDNN for Tensorflow 2.7 (see `https://www.tensorflow.org/install/source#linux`). Otherwise, it is possible to generate a GPU enabled Docker container with all the dependencies, by running this command in a terminal placed in *src* folder:

```
docker build -f ../docker/Dockerfile -t falanti/popnas:tf2.7.0gpu .
```

Executing POPNASv2

Running POPNASv2 is very simple, just open a terminal in *src* folder and run:

```
python run.py
```

or if running a Docker container:

```
docker run -it --rm -v %cd%:/exp --name popnas falanti/popnas:tf2.7.0gpu
↪ python run.py
```

The run use by default the configuration file contents of *configs/run.json* to parametrize the run. If you want to use a different custom configuration file, simply use the run the script with the **-j** flag specified:

```
python run.py -j {path to your JSON config}
```

Run configuration

This section provides exhaustive details on how to configure a POPNASv2 run.
For further information, see also the README file provided in the repository.

Command line arguments

All command line arguments are optional.

- *-j*: specifies the path of the json configuration to use. If not provided, *configs/run.json* will be used.
- *-r*: used to restore a previous interrupted run. Specifies the path of the log folder of the run to resume.
- *--cpu*: if specified, the algorithm will use only the CPU, even if a GPU is actually available. It must be specified if the host machine has no gpu.
- *--pnas*: if specified, the algorithm will not use a time predictor, disabling time estimation and Pareto front generation. This will make the computation extremely similar to PNAS algorithm.

Json configuration file

The run behavior can be customized through the usage of custom json files. By default, the *run.json* file inside the *configs* folder will be used. This file can be used as a template and customized to generate new configurations. A properly structured json config file can be used for starting the algorithm, by specifying its path in -j command line arguments.

Here it is presented a list of the configuration sections in which the json file is divided, accompanied with a brief description of their fields.

Search Space:

- *blocks*: defines the maximum amount of blocks a cell can contain.
- *max_children*: defines the amount of top-K cells the algorithm picks up to expand at the next iteration.
- *max_exploration_children*: defines the maximum amount of cells the algorithm can train in the exploration step.
- *lookback_depth*: maximum lookback depth to use.

- *lookforward_depth*: maximum lookforward depth to use. TODO: actually not supported, should always be null.
- *operators*: list of operators that can be used inside each cell. Note that the string format is important, since they are recognized by regexes. Actually supported operators, with customizable kernel size(@):
  - identity
  - @x@ dconv (Depthwise-separable convolution)
  - @x@-@x@ conv
  - @x@ conv
  - @x@ maxpool
  - @x@ avgpool
  - @x@ tconv (Transpose convolution)

CNN hyperparameters:

- *epochs*: defines for how many epochs each child network has to be trained.
- *batch_size*: defines the batch size dimension of the dataset.
- *learning_rate*: defines the learning rate of the child CNN networks.
- *filters*: defines the initial number of filters to use.
- *weight_reg*: defines the L2 regularization factor to use in CNNs. If *null*, regularization is not applied.
- *use_adamW*: use adamW instead of standard L2 regularization.
- *drop_path_prob*: defines the max probability of dropping a path in *scheduled drop path*. If set to 0, then *scheduled drop path* is not used.
- *cosine_decay_restart*: dictionary for hyperparameters about cosine decay restart schedule.
  - *enabled*: use cosine decay restart or not (plain learning rate).
  - *period_in_epochs*: first decay period in epochs.
  - [*t_mul, m_mul, alpha*]: see `tensorflow documentation`.

- *softmax_dropout*: probability of dropping a value in output Softmax. If set to 0, then dropout is not used. Note that dropout is used on each output when *multi_output* flag is set.

CNN architecture parameters:

- *motifs*: motifs to stack in each CNN. In NAS literature, a motif usually refers to a single cell, here instead it is used to indicate a stack of N normal cells followed by a single reduction cell.
- *normal_cells_per_motif*: normal cells to stack in each motif.
- *concat_only_unused_blocks*: when *true*, only block outputs not used internally by the cell will be used in final cell output concatenation, following PNAS and NASNet. If set to *false*, all block outputs will be concatenated in final cell output.
- *multi_output*: if true, each CNN generated will have an output (GAP + Softmax) at the end of each cell.

RNN hyperparameters (controller, optional):

If the parameters are not provided or the object is omitted in JSON config, default parameters will be applied. They depend on the model type chosen for the controller.

- *epochs*: how many epochs the LSTM predictor is trained on, at each expansion step.
- *lr*: LSTM learning rate.
- *wr*: LSTM L2 weight regularization factor. If *null*, regularization is not applied.
- *er*: LSTM L2 weight regularization factor applied on embeddings only. If *null*, regularization is not applied.
- *embedding_dim*: LSTM embedding dimension, used for both inputs and operator embeddings.
- *cells*: total LSTM cells of the model.

Dataset:

- *name*: used to identify and load a Keras or Tensorflow dataset. Can be *null* if the path of a custom dataset is provided.
- *path*: path to a folder containing a custom dataset. Should be *null* if you want to use a dataset already present in Keras or Tensorflow datasets.

- *classes_count*: classes present in the dataset. If using a Keras dataset, this value can be inferred automatically.
- *folds*: number of dataset folds to use. When using multiple folds, the metrics extrapolated from CNN training will be the average of the ones obtained on each fold.
- *samples*: if provided, limits the total dataset samples to the number provided (integer). This means that the total training and validation samples will amount to this value (or less if the dataset has actually fewer samples than the value indicated). Useful for fast testing.
- *data_augmentation*: dictionary with parameters related to data augmentation
  - *enabled*: use data augmentation or not.
  - *perform_on_gpu*: perform data augmentation directly on GPU (through Keras experimental layers). Usually advised only if the CPU is very slow, since the CPU can prepare the images while the GPU trains the network (asynchronous prefetch), instead performing data augmentation on the GPU will make the process sequential, always causing delays even if it is faster to perform.

Others:

- *predictions_batch_size*: defines the batch size used when performing both time and accuracy predictions in controller update step (predictions about cell expansions for blocks b+1). Incrementing it should decrease the prediction time linearly, up to a certain point, defined by hardware resources used.
- *pnas_mode*: if *true*, the algorithm will not use the temporal regressor and Pareto front search, making the run very similar to PNAS.
- *use_cpu*: if *true*, only CPU will be used, even if the device has usable GPUs.

Output folder structure

The results of all runs are stored under the *logs* folder. By default, the folder will be named with the timestamp in which the run is started. The output folder is composed of multiple subfolders:

- *best-model* folder contains the checkpoint of the best model with B blocks.
- *csv* folder contains the csv files with the data about the models trained, the pareto fronts generated and the predictions made.

- *plots* folder contains the plots generated by the plotter module during the run.
- *predictors folder* contains the output of each predictor trained during the run.
- *restore* folder contains files used to restore a stopped run, without losing the progress made before.
- *tensorboard-cnn* folder contains Tensorboard data and other relevant info, like the network summary and the total FLOPS. Using Tensorboard on this folder allows to visualize the evolution of the accuracy and loss metrics during training, for each network explored by the algorithm.

List of Figures

2.1 Hypothesis space visualization 6
2.2 Biological neuron model 9
2.3 Artificial neuron model 10
2.4 Backpropagation example 12
2.5 Fully-connected neural network example 14
2.6 Convolution example 15
2.7 CNN architecture example 16
2.8 RNN unroll 17

3.1 NAS workflow overview 20
3.2 Macro and micro search example 22
3.3 NASNet controller 24
3.4 POPNAS search procedure 38

4.1 Block DAG example 45
4.2 Cell equivalence example 46
4.3 POPNASv2 CNN architecture 49
4.4 POPNASv2 search procedure 52
4.5 POPNASv2 LSTM structure 57

5.1 Comparison between old and tentative feature sets 64
5.2 Feature correlation heatmap 65
5.3 SHAP feature analysis 67
5.4 Abstract cell DAG for the feature set 69
5.5 Tentative and definitive feature sets comparison 71

6.1 POPNASv2 Pareto front for CIFAR10 (b=5) 76

A.1 CIFAR10 top1 cells 87
A.2 CIFAR100 top1 cells 87
A.3 Fashion MNIST top1 cells 88

A.4 EuroSAT top1 cells 88
A.5 POPNASv2 inputs and operators usage on CIFAR10, in step $b = 5$. 90
A.6 PNAS inputs and operators usage on CIFAR10, in step $b = 5$. 91
A.7 POPNASv2 inputs and operators usage on CIFAR100, in step $b = 5$. . . . 92
A.8 PNAS inputs and operators usage on CIFAR100, in step $b = 5$. 93
A.9 POPNASv2 inputs and operators usage on Fashion MNIST, in step $b = 5$. 94
A.10 PNAS inputs and operators usage on Fashion MNIST, in step $b = 5$. 95
A.11 POPNASv2 inputs and operators usage on EuroSAT, in step $b = 5$. 96
A.12 PNAS inputs and operators usage on EuroSAT, in step $b = 5$. 97

List of Tables

3.1 Hyperparameter set used in NAS-bench-101 32
3.2 NAS-Bench-101 and NAS-Bench-201 comparison 34

5.1 Equivalence check results 61
5.2 Cells introduced by exploration step 62
5.3 Time features extracted from DAG of cell example *c*. 70
5.4 Time regressor comparison 72

6.1 Dataset characteristics 74
6.2 POPNASv2 accuracy predictor results 75
6.3 POPNASv2 time predictor results 75
6.4 POPNASv2 and PNAS search performance comparison 77
6.5 POPNASv2 and PNAS top-1 networks comparison 78