Pontifícia Universidade Católica
do Rio de Janeiro

Daniela de Mattos Szwarcman

Quantum-inspired Neural Architecture Search

Tese de Doutorado

Thesis presented to the Programa de Pós-graduação em Engenharia Elétrica of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Engenharia Elétrica.

Advisor: Prof. Marley Maria Bernardes Rebuzzi Vellasco
Co-advisor: Dr. Daniel Salles Civitarese

Rio de Janeiro
February 2020

Pontifícia Universidade Católica
do Rio de Janeiro

Daniela de Mattos Szwarcman

Quantum-inspired Neural Architecture Search

Thesis presented to the Programa de Pós-graduação em Engenharia Elétrica of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Engenharia Elétrica. Approved by the Examination Committee.

Prof. Marley Maria Bernardes Rebuzzi Vellasco
Advisor
Departamento de Engenharia Elétrica – PUC-Rio

Dr. Daniel Salles Civitarese
Co-advisor
IBM Research

Dr. Bianca Zadrozny
IBM Research

Prof. Douglas Mota Dias
UERJ

Prof. Jorge Amaral
UERJ

Prof. Valmir Carneiro Barbosa
UFRJ

Prof. Ricardo Tanscheit
Departamento de Engenharia Elétrica – PUC-Rio

Prof. Karla Figueredo
Departamento de Engenharia Elétrica – PUC-Rio

Rio de Janeiro, February the 17th, 2020

Daniela de Mattos Szwarcman

Graduated in Electrical Engineering at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in 2013. Obtained her M.Sc. Degree in Electrical Engineering from PUC-Rio in 2016.

Bibliographic data

de Mattos Szwarcman, Daniela

Quantum-inspired Neural Architecture Search / Daniela de Mattos Szwarcman; advisor: Marley Maria Bernardes Rebuzzi Vellasco; co-advisor: Daniel Salles Civitarese. – Rio de janeiro: PUC-Rio, Departamento de Engenharia Elétrica, 2020.

v., 97 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Engenharia Elétrica.

Inclui bibliografia

1. Engenharia Elétrica – Teses. 2. Busca de Arquiteturas Neurais. 3. Algoritmos de Inspiração Quântica. 4. Neuroevolução. 5. Redes Neurais Convolucionais. I. Bernardes Rebuzzi Vellasco, Marley Maria. II. Salles Civitarese, Daniel. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Engenharia Elétrica. IV. Título.

CDD: 621.3

Acknowledgments

First, I would like to thank my advisor Marley for her fundamental guidance and understanding, besides always listening to my opinions and giving me valuable feedback.

I would also like to thank my co-advisor Daniel for his support, ideas, patience, friendship, and, most importantly, for always encouraging me.

I wish to thank my colleagues and friends at IBM Research, especially Emilio and Breno, who contributed to meaningful discussions on various subjects.

I am grateful to all the members of the committee for agreeing to participate and collaborate with this work.

I would like to acknowledge CNPq for financial support and that this study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

I wish to thank my parents, Dilza and Moisés, for their constant support, patience, and advice, but most of all, for their love and attention.

My eternal thanks to my sister Clara for being the best sister and the best friend, who always is there for me and makes my days happier.

Finally, I wish to appreciate the best person in the world, Gustavo, for his love, kindness, and encouragement. I will be forever grateful for his endless support during all my years as a student, for inspiring me every day, and for being my partner in life.

Abstract

de Mattos Szwarcman, Daniela; Bernardes Rebuzzi Vellasco, Marley Maria (Advisor); Salles Civitarese, Daniel (Co-Advisor). **Quantum-inspired Neural Architecture Search**. Rio de Janeiro, 2020. 97p. Tese de doutorado – Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

Deep neural networks are powerful and flexible models that have gained the attention of the machine learning community over the last decade. For a variety of tasks, they can even surpass human-level performance. Usually, to reach these excellent results, an expert spends significant time designing the neural architecture, with long trial and error sessions. In this scenario, there is a growing interest in automating this design process. To address the neural architecture search (NAS) problem, authors have presented new methods based on techniques such as reinforcement learning and evolutionary algorithms, but the high computational cost is still an issue for many of them. To reduce this cost, researchers have proposed to restrict the search space, with the help of expert knowledge. Quantum-inspired evolutionary algorithms present promising results regarding faster convergence. Motivated by this idea, we propose Q-NAS: a quantum-inspired algorithm to search for deep networks by assembling substructures. Q-NAS can also evolve some numerical hyperparameters, which is a first step in the direction of complete automation. We ran several experiments with the CIFAR-10 dataset to analyze the details of the algorithm. For many parameter settings, Q-NAS was able to achieve satisfactory results. Our best accuracies on the CIFAR-10 task were 93.85% for a residual network and 93.70% for a convolutional network, overcoming hand-designed models, and some NAS works. Considering the addition of a simple early-stopping mechanism, the evolution times for these runs were 67 GPU days and 48 GPU days, respectively. Also, we applied Q-NAS to CIFAR-100 without any parameter adjustment, reaching an accuracy of 74.23%, which is comparable to a ResNet with 164 layers. Finally, we present a case study with real datasets, where we used Q-NAS to solve the seismic classification task. In less than 8.5 GPU days, Q-NAS generated networks with 12 times fewer weights and higher accuracy than a model specially created for this task.

Keywords

Neural Architecture Search. Quantum-Inspired Algorithms. Neuroevolution. Convolutional Neural Networks.

Resumo

de Mattos Szwarcman, Daniela; Bernardes Rebuzzi Vellasco, Marley Maria; Salles Civitarese, Daniel. **Busca de arquiteturas neurais com algoritmos evolutivos de inspiração quântica**. Rio de Janeiro, 2020. 97p. Tese de Doutorado – Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

As redes neurais *deep* são modelos poderosos e flexíveis, que ganharam destaque na comunidade científica na última década. Para muitas tarefas, elas até superam o desempenho humano. Em geral, para obter tais resultados, um especialista despende tempo significativo para projetar a arquitetura neural, com longas sessões de tentativa e erro. Com isso, há um interesse crescente em automatizar esse processo. Novos métodos baseados em técnicas como aprendizado por reforço e algoritmos evolutivos foram apresentados como abordagens para o problema da busca de arquitetura neural (NAS - *Neural Architecture Search*), mas muitos ainda são algoritmos de alto custo computacional. Para reduzir esse custo, pesquisadores sugeriram limitar o espaço de busca, com base em conhecimento prévio. Os algoritmos evolutivos de inspiração quântica (AEIQ) apresentam resultados promissores em relação à convergência mais rápida. A partir dessa idéia, propõe-se o Q-NAS: um AEIQ para buscar redes *deep* através da montagem de subestruturas. O Q-NAS também pode evoluir alguns hiperparâmetros numéricos, o que é um primeiro passo para a automação completa. Experimentos com o conjunto de dados CIFAR-10 foram realizados a fim de analisar detalhes do Q-NAS. Para muitas configurações de parâmetros, foram obtidos resultados satisfatórios. As melhores acurácias no CIFAR-10 foram de $93{,}85\%$ para uma rede residual e $93{,}70\%$ para uma rede convolucional, superando modelos elaborados por especialistas e alguns métodos de NAS. Incluindo um esquema simples de parada antecipada, os tempos de evolução nesses casos foram de 67 dias de GPU e 48 dias de GPU, respectivamente. O Q-NAS foi aplicado ao CIFAR-100, sem qualquer ajuste de parâmetro, e obteve $74{,}23\%$ de acurácia, similar a uma ResNet com 164 camadas. Por fim, apresenta-se um estudo de caso com dados reais, no qual utiliza-se o Q-NAS para resolver a tarefa de classificação sísmica. Em menos de $8{,}5$ dias de GPU, o Q-NAS gerou redes com 12 vezes menos pesos e maior acurácia do que um modelo criado especialmente para esta tarefa.

Palavras-chave

Busca de Arquiteturas Neurais. Algoritmos de Inspiração Quântica. Neuroevolução. Redes Neurais Convolucionais.

Table of contents

List of figures

List of tables

List of Abreviations

AutoML – Automated Machine Learning
BN – Batch Normalization
CGP – Cartesian Genetic Programming
EA – Evolutionary algorithm
FC – Fully-connected
GA – Genetic Algorithm
NAS – Neural Architecture Search
QIEA – Quantum-inspired Evolutionary Algorithm
RL – Reinforcement Learning
SGD – Stochastic Gradient Descent

1
Introduction

Over the past few years, the machine learning community has witnessed significant progress in the performance of deep networks for many tasks, especially in image and speech recognition [1, 2]. Discussions on the subject were not in the foreground until 2006 when Hinton et al. [3] efficiently trained their Deep Belief Nets [4]. Although deep convolutional networks had been trained earlier, they only became a relevant topic in machine learning research after the layer-wise pretraining strategy [4, 5], proposed in 2006 [3].

Deep convolutional neural networks are highly responsible for the current success of deep architectures. These networks provide a way to automatically learn feature extractors from the dataset, eliminating the need for feature engineering [1]. Several researchers have proposed different deep convolutional models. For image applications, AlexNet [6], VGGNet [7], Network-in-Network [8], and ResNet [9] are successful examples of these models.

Together with this success, the demand for architecture engineering has emerged, shifting the paradigm from feature design to network design. The process of manually engineering deep architectures requires expert knowledge and a considerable amount of time to test multiple options. In this scenario, the idea of automating the network design has gained the attention of many researchers [1, 2], establishing the field of Neural Architecture Search (NAS).

In addition, the performance of deep networks is usually sensitive to design decisions regarding the architecture itself, the training procedure, the regularization methods, and hyperparameters selection. An expert must make all these decisions often by trial and error until they identify a set of choices that lead to satisfactory performance. It is worth noting that this engineering process must be repeated for every new application [1].

The area of automatic machine learning (AutoML) attempts to address the complete automation of the decision process. The goal is to make machine learning accessible to other scientists who want to apply these techniques to their domains. NAS can be seen as a subfield in the AutoML area, and it is an essential step toward the automation of machine learning methods.

Even though the automatic design of neural networks is not a new idea, its application in deep architectures is quite recent. Works on neuroevolution describe

evolutionary algorithms that evolve not only the network weights but also its structure [10–14]. In the deep architecture context, on the other hand, the NAS problem extends beyond the field of evolutionary algorithms.

Several papers have proposed new approaches to address the NAS problem, showing competitive results when compared to manually engineered networks. The majority of these papers focus on the image classification task, more specifically on the CIFAR-10 [15] benchmark dataset [1]. The algorithms are based on different techniques, such as reinforcement learning (RL) [2, 16–18], evolutionary algorithms [19–21] or even Bayesian optimization [22]. They also differ in other factors, including the performance estimation strategy and the search space description.

Despite the current success and progress of NAS algorithms, the required computational resources are still significant for many of them [2, 17, 19, 21]. The approach presented in [2] is an extreme example, in which the authors used 800 GPUs for more than three weeks to reach 96% of accuracy in the CIFAR-10 dataset.

Many researchers have considered the challenge of reducing computational cost. More recently, various solutions have been developed, usually with some trade-offs that are mostly related to search space reduction. There are efforts toward efficiency using hypernets [23], network transformations [24, 25], early stopping [18], block search [18, 26], among others. The block search strategy optimizes small cells that are later stacked in a predefined way to build the final network, which limits the exploration space substantially. Even with the new ideas, there is still room for improvement, especially concerning human bias to restrict the search space.

1.1 Objectives

In this work, we propose Q-NAS: a quantum-inspired algorithm to search for deep architectures by assembling substructures. In its implementation, our design decisions consider the computational cost issue, but without adopting the block search strategy.

Quantum-inspired evolutionary algorithms (QIEA) are a class of evolutionary methods based on quantum computing principles, such as superposition of states [27, 28]. Empirical results show that QIEAs can find better solutions with fewer evaluations when compared to similar algorithms for many optimization problems [29,30]. For applications in which the evaluation procedure is expensive, this feature can be a valuable advantage of QIEAs over other evolutionary methods. Moreover, QIEAs have been successfully applied to several optimization problems, including the neuroevolution of shallow networks [12–14, 31].

Our primary goal is to verify the applicability of a new QIEA to the NAS problem. This verification involves applying Q-NAS to benchmark datasets to

compare its results with other NAS algorithms. It also includes running Q-NAS with different parameter configurations, to analyze the impact on final results.

Furthermore, Q-NAS addresses the network search and can, at the same time, optimize some numerical hyperparameters. We intend to study this feature, comparing the performance against the fixed hyperparameter scenario.

We also want to investigate if Q-NAS can be an alternative to cell search algorithms in terms of efficiency.

Finally, we apply Q-NAS on seismic image datasets to verify its performance in a realistic scenario.

We delineate the scope of our research in the context of image classification tasks, so our experiments and conclusions reside in this context.

1.2 Contributions

In this section, we list the main contributions of this work:

- **A new QIEA for the NAS problem.** Motivated by the faster convergence achieved by other QIEA authors, we developed a new algorithm to address the NAS problem. Reducing the computational cost is essential to make NAS feasible when dealing with large datasets. To the best of our knowledge, a QIEA has never been used to address this problem before.
- **A new quantum-inspired representation for the network structure.** We designed a specific representation to deal with a categorical search space, different from other binary approaches. We encode networks in a chain-like structure of nodes that can be either a layer or a (complex) block of layers, providing significant flexibility to the Q-NAS user. There is no need to modify the algorithm to switch between simple and complex networks; the user only needs to define which type of nodes he/she wants to use. Additionally, our quantum representation enables the discovery of different network topologies, as opposed to other works using block search.
- **A combined quantum-representation, to encode network structures and numerical hyperparameters.** Q-NAS can focus only on the NAS problem, but it is also possible to optimize some numerical hyperparameters. The representation of these hyperparameters is an enhanced version of the one presented in [30].
- **A simple early-stopping mechanism to improve Q-NAS efficiency.** We studied the benefits of applying a simple early-stopping mechanism that is based only on fitness improvement and, therefore, it is not restricted to Q-NAS. In almost all cases, this simple scheme allowed us to significantly reduce the evolution runtime, indicating that it is a valuable addition to Q-NAS.

- **A study on penalization to handle invalid individuals in Q-NAS.** When generating networks for evaluation, it is possible to produce invalid architectures. We propose to address this issue via a penalization scheme. We investigated the method not only on a performance point of view but also on how the population of solutions is affected when penalization is present. This study helped us examine the influence of invalid networks in the final results and identify potential issues when the networks are deeper.
- **A study about the impact of training schemes on the final result.** As observed in [32], several authors in the NAS literature apply specially adjusted learning schedules when training their final structures. We investigated the difference in retraining our networks with distinct learning schemes. We observed that, although a special learning schedule shows better results than the others, the cosine schedule presents similar accuracy, without any tuning.
- **An environment for the study of network topologies.** Q-NAS generates thousands of structures during evolution. Since we store the population descriptions, it is possible to analyze if there are common patterns, such as a specific sequence of nodes that frequently appears. This analysis can be used to discover new primitives to assemble different architectures. Also, it can be useful to compare the networks generated for distinct tasks.

1.3 Work outline

This work comprises five additional chapters, which we describe below.

In Chapter 2, we provide the theoretical background necessary to understand the NAS context, which includes the concepts of convolutional neural networks and a review of some NAS works.

We present quantum-inspired evolutionary algorithms in Chapter 3, including the definition of essential concepts and a brief description of some previous works.

Q-NAS is introduced in the following chapter, with a focus on the newly developed quantum-representation. The steps of the algorithm are also specified.

Next, Chapter 5 describes and discusses the experiments. We present our experimental track in the order it was developed, with each section addressing a specific investigation subject.

Finally, we present the final remarks in Chapter 6, along with the next steps of our research.

2
Neural Architecture Search

In this chapter, we present the context of neural architecture search, with the focus on convolutional neural networks for image recognition problems. First, we review concepts regarding convolutional neural networks, including the required design decisions to build and train them. Next, we review recent neural architecture search algorithms for image classification problems.

2.1
Convolutional Neural Networks

In 1980, Fukushima [33] introduced an artificial neural network based on a hierarchical model of the visual system's structure, proposed in 1962. The key idea involved applying neurons with the same parameters on patches of the previous layer at different locations, to obtain a form of translational invariance [4]. This invariance is essential for tasks such as image recognition: in this situation, one is interested in the presence of a certain motif, regardless of its location [34]. Fukushima trained his network, called Neocognitron, using an unsupervised learning[1] scheme [33, 35]. Some years later, LeCun and other collaborators created the convolutional neural network (CNN), following the Neocognitron ideas. However, they used backpropagation (supervised learning) to train the network [35], achieving state-of-the-art results for pattern recognition tasks [36].

CNNs are designed to process data in the shape of multiple arrays, such as images, audio spectrograms, or volumetric images. In other words, the CNN structure takes advantage of the properties of these types of signals.

Just as the name suggests, CNNs employ the linear operation of convolution. The convolution of a function $x(t)$ with a function $w(t)$ is defined as follows:

$$s(t) = \int_{-\infty}^{\infty} x(a)w(t-a)\mathrm{d}a = (x * w)(t) \tag{2-1}$$

As an illustrative example, imagine that $x(t)$ is the position of a particle at time t and that the sensor to measure this position is noisy. It is possible to average several data points to obtain a cleaner version of signal $x(t)$. Supposing that recent observations are more relevant than old ones, we can use the weighting function

[1] Details about unsupervised learning can be found in [5]

$w(a)$, where a is the age of the measures. By applying this weighted average, we obtain a smoothed estimate $s(t)$ of $x(t)$. This operation is also defined for discrete values of t; we simply substitute the integral for an infinite summation [5].

In the context of CNNs, x is the input, w is the *kernel* or *filter*, and s is called *feature map*. Also, the input is usually a multidimensional array of data (*tensors*) and the kernel, an array of parameters that will be adjusted by the learning algorithm. For example, the input can be a 2D image I and in this case, the kernel K should also be two-dimensional. The convolution operation will then be applied simultaneously on both axes:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2-2}$$

where the infinite summation can be reduced to a finite summation, assuming that the input is zero everywhere outside the image area. Figure 2.1 shows a visual example of the convolution operation applied on a 2D image I.

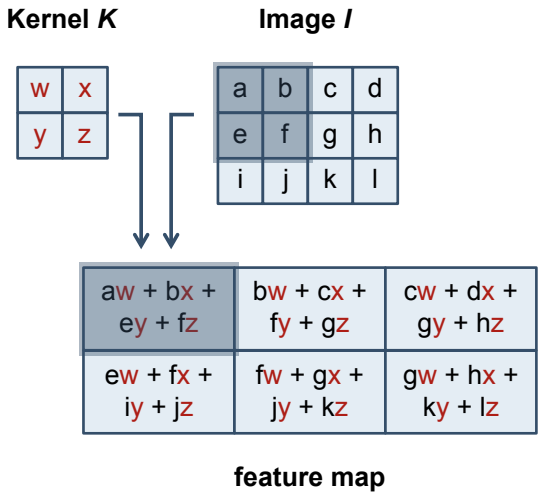

Figure 2.1: Example of 2D convolution. The output in this example is only calculated for the positions where the kernel lies entirely inside the image. This is called *valid* convolution.

The use of convolution operations in neural networks provides advantages that are related to three main ideas: sparse interactions, parameter sharing, and equivariant representations [5].

In traditional neural networks, such as the Multilayer Perceptron (MLP), all elements in the input interact with the output elements. On the other hand, in CNNs, if the convolution kernel is smaller than the input, there will be sparse connections. This idea can be seen in Figure 2.1: input b, for example, only influences the first and second outputs in the feature map. The sparse connectivity reduces the amount of memory needed to store parameters and the number of operations to calculate the output. This reduction is converted into a significant efficiency improvement [5].

The second advantage is related to the fact that each kernel member is used in all positions of the input image, i.e., the parameters are shared (see Figure 2.1). Once again, this represents an increase in efficiency compared to densely connected networks.

This specific form of parameter sharing in a CNN leads to a property called *equivariance* to translation. A function $f(x)$ is equivariant to a function $g(x)$ if $f(g(x)) = g(f(x))$. If g is a function that shifts the input, then the convolution is equivariant to g. This means that if we move an object in the input image, for example, its representation moves in the same way in the output image. It is worth mentioning that the convolution operation is not equivariant to transformations like changes in scale or rotations.

We just defined the convolution operation and its advantages, but we also need to describe how CNNs are built using convolutional layers. Figure 2.2 shows a typical set of CNN layers, composed of three stages. The first one is responsible for the convolution operations that produce a set of linear activations [5]. To define a convolutional layer, we need to specify the kernel size $m \times n$, the stride s in which the kernel will scan the input, and the number f of feature maps that the layer will produce. This specification determines that f different kernels of size $m \times n$ and stride s will slide across the input tensor, generating f output maps. In this work, we refer to a convolutional layer of size $k \times k$, stride s and f feature maps as Conv(k, s, f).

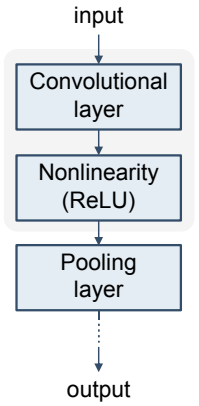

Figure 2.2: Example of a typical set of layers in a CNN. The nonlinearity can be suppressed in illustrations, and the gray area will be depicted only as *convolutional layer*.

In the second stage or *detector stage*, a nonlinear activation function is applied to the convolution outputs. The standard nonlinear function is the Rectified Linear Unit, or simply ReLU, defined as $f(x) = \max(0, x)$ [5, 34]. We remark that this nonlinearity is usually suppressed in illustrations: it is implicitly represented in the convolutional layer.

Next, a pooling function replaces the activations of a feature map with summary statistics of neighboring activations. These functions define a rectangular

neighborhood of size $k \times k$ that also slides across the input with a stride s. The values that fall into this neighborhood are replaced by a statistic [5, 34], such as the maximum value (*MaxPool*) or the average value (*AvgPool*). Here, we refer to a MaxPool layer with stride s and neighborhood $k \times k$ as MaxPool(k, s); the same applies to AvgPool. Note that k and s usually are set to be larger than 1 to *downsample* the input. By reporting the summary statistics of $k \times k$ neighborhoods spaced s pixels apart, one reduces the feature map size. This reduction is valuable to improve efficiency, especially in cases where translation invariance is more important than accurate locations. Pooling helps to make representation invariant to small translations of the input, as the summary statistics remain stable in the neighborhood [5,34]. For classification problems, pooling functions can be advantageous, as we are only interested in the presence of an object in the image. However, by replacing the pixels with summary statistics, we can lose feature position information, which is essential in applications such as object detection. This corroborates the fact that pooling layers are not always present in convolutional networks, as seen in [37].

A deep CNN is formed by repeatedly stacking these types of layers. In a typical architecture, two or three stages of convolution and pooling are stacked, followed by more convolution stages and fully-connected (FC) layers [34]. In the context of classification tasks, the last layer is responsible for outputting the scores for each category, and it is usually a fully-connected layer. If we have 100 classes in the dataset, the last fully-connected layer should have 100 units.

Figure 2.3 shows the deep architecture Alexnet that won the ImageNet challenge [38] in 2012. It was able to distinguish the 1000 classes with top-5 test error

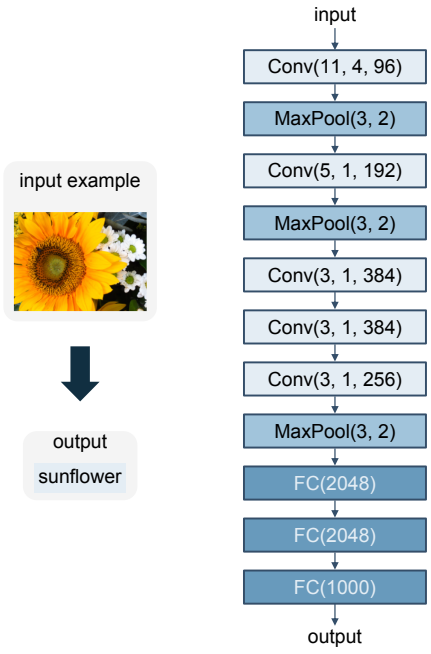

Figure 2.3: Alexnet architecture for classifying the 1000 classes in the ImageNet dataset.

rate of 15.3% [6]. Notice that it has convolutional layers followed by other convolutions or by pooling functions, but its general form matches the stacking recipe we just mentioned.

2.1.1 Training a CNN

We have defined the general form of a CNN model, and now we need to describe the *training* procedure, or how the network weights are adjusted. We will consider the supervised learning case, and the image classification task will serve as an example to illustrate this procedure.

Suppose we have a collection of images labeled as different classes, such as a car, a dog, a flower, or others. Supervised learning involves the observation of several examples of input x (our images) and its associated target y (our class label) and then learning to predict y from x [5]. Ideally, after learning from the training data collection, the model will be able to predict the classes of unseen images – ability of *generalizing* what was just learned. In the neural network context, observing the input image consists of providing it to the network so it propagates through the layers. A score for each category is generated at the output, completing the *forward propagation*. The idea is to adjust the network weights so that the highest score appears for the correct label of the corresponding image. We must define an *error function* that will measure how far the current output is from the correct answer. The learning algorithm consists of minimizing this error function (or *cost function*) with respect to the weights of the network. Stochastic Gradient Descent (SGD) and its variants are the most popular optimization algorithms for training CNNs. It consists of the following steps [34]:

1. presenting a *batch* of input arrays – images, in our example;
2. computing the output scores and errors;
3. calculating the average gradients for the batch of examples;
4. adjusting the weights accordingly.

The gradient descent technique is characterized by the observation that we can reduce a function $f(x)$ by moving in small steps with the opposite sign of its gradient. This iterative optimization algorithm proposes a new point $\boldsymbol{x}'$ as [5]:

$$\boldsymbol{x}' = \boldsymbol{x} - \lambda \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{2-3}$$

where λ is the *learning rate*, a positive real number which determines the step size. The algorithm converges when every element of the gradient is zero [5].

To actually compute the error function gradients with respect to each network weight, the *backpropagation* algorithm is used. Backpropagation repeatedly applies the chain rule of calculus, starting from the output of the network and moving to the input. Details about the backpropagation algorithm can be found in [5].

One modification to the regular SGD is the addition of *momentum*. The SGD with momentum accumulates an exponentially decaying moving average of the previous gradients and continues to move in their direction. A parameter α, called momentum, controls how fast the contributions from past gradients will decay [5].

The learning rate is a critical parameter for the SGD algorithm, as it can affect the model performance significantly. Modern mini-batch based optimizers try to address this issue by *individually adapting learning rates* for the model's parameters. These algorithms use a separate learning rate for each weight and automatically adapt these rates during training. The RMSProp is one well-known example of these algorithms: it reduces the learning rate according to the historical square gradients. It also discards contributions from the extreme past with the help of an exponentially decaying moving average. RMSProp adds the moving average decay to the list of parameters to be specified by the user. [5].

2.1.2 Regularization

In the training procedure just presented, we compute the errors on a set of inputs called the *training set*, and minimize the cost function. However, any machine learning model should have the ability of *generalization*, that is, to perform well not only on the training set but also on unseen data [5]. This requirement can be challenging, as our learning algorithm is not directly addressing it. Several strategies to reduce the *generalization error*, known as *regularization* techniques [5], have been developed. We will briefly describe here the methods relevant to this work.

One of the most common regularization techniques is the L^2 parameter norm penalty or *weight decay*. The idea is to add a penalty term to the cost function to favor weights with smaller squared L^2 norm. Considering a cost function $C(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$, the new cost with weight decay becomes [5]:

$$C_{reg}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = C(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \frac{\alpha}{2}\|\boldsymbol{w}\|^2 \tag{2-4}$$

where $\alpha \in [0, \infty)$ is a hyperparameter that controls the contribution of the penalty term relative to the original cost function. The result of minimizing $C_{reg}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is to select weights that make a trade-off between fitting the training data and having small values [5].

Even though weight decay is simple to use, one could claim that an effective way to make a model generalize better is to train it on more data. However, in reality, we have access to a limited set of examples. For some problems, it might be easy to create fake data to increase the training set, a method called *data augmentation* [5].

In our object classification task, it is straightforward to simulate real variations by applying simple operations on the images. For example, we can pad the images and then randomly crop them; small brightness alterations are also possible; random flip or rotations are common choices. Moreover, one can perform data augmentation during training (*online*). When presenting a training example to the network, it has a 50% chance of being modified by one operation. In this way, the network can become more robust to these variations without the need to collect more data.

Another successful and more recent approach is called *batch normalization* (BN) [39]. Although it was primarily designed to improve convergence in deep models, batch normalization can also act as a regularizer [5, 39, 40].

Deep models are composed of many layers, and the inputs to a layer are affected by the preceding ones. When training such models with SGD, we update the weights according to the gradients [5]. However, small changes in the weights can be amplified with increasing network depth [39]. So, after the update, the inputs to each layer can be significantly modified, which may delay convergence, as the layers need to adapt to new input distributions all the time. The idea behind BN is to reduce the variations in these distributions. Consider, for example, an activation x in the network and a mini-batch B, in which we have m values for this activation: $B = \{x_1, ..., x_m\}$. The BN transform is defined as [39]:

$$\begin{aligned} BN_{\gamma,\beta}(x_i) &\equiv \gamma\hat{x}_i + \beta; \quad \hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}; \\ \mu_B &= \frac{1}{m}\sum_{k=1}^{m} x_i; \quad \sigma_B^2 = \sum_{k=1}^{m}(x_i - \mu_B)^2 \end{aligned} \tag{2-5}$$

where μ_B and σ_B^2 are the mini-batch mean and variance, γ and β are learned parameters, and ϵ is a constant for numerical stability. Notice that each activation x in the network has its unique γ and β parameters that are learned along with the original model weights [39]. This normalization introduces additive and multiplicative noise on the layers, which can have a regularizing effect [5].

It is important to note that the presented methods can be used together: one can apply batch normalization and data augmentation simultaneously, for example.

2.1.3 Residual Learning

As already mentioned, batch normalization stabilizes the gradients, improving convergence. This method is one of the techniques that enabled researchers to train deeper networks. However, with growing depth, a *degradation* problem arises: accuracy saturates and then deteriorates fast while training error increases [9]. The authors in [9] propose the *residual learning* framework to address the degradation problem.

To understand the idea behind residual learning, first consider two networks: a shallow one and a deep counterpart, created by adding identity layers on the smaller model. The authors claim that the existence of this construction indicates that a deeper network should not present higher training errors than its shallow equivalent. Since experiments show the opposite, they also affirm that current solvers are unable to find better (or even similar) solutions than this identity topology. So, they propose to make the layers fit a residual mapping instead of the original unreferenced mapping. If the stacked identity layers were the optimal solution, they claim it would be easier to zero out the residual than directly try to fit the identity [9].

If we denote the underlying mapping as $H(\boldsymbol{x})$, the residual mapping is then $F(\boldsymbol{x}) \equiv H(x) - \boldsymbol{x}$. The original mapping becomes $F(\boldsymbol{x}) + \boldsymbol{x}$ [9]. Figure 2.4 (a) shows a *residual unit* that implements the new formulation of $H(\boldsymbol{x})$. Then, to build a residual network, we stack these units instead of standard convolutional layers.

The shortcut (or *skip connection*) in the unit of Figure 2.4 (a) can be directly used if the input $\boldsymbol{x}$ and the output $F(\boldsymbol{x}) + \boldsymbol{x}$ have the same dimensions. When the sizes do not match, there are two typical approaches to fix this issue. The first one pads the input with zeros when performing the summation. The second uses a *projection shortcut*, depicted in Figure 2.4 (b), which applies a 1×1 convolution to fix the dimensions.

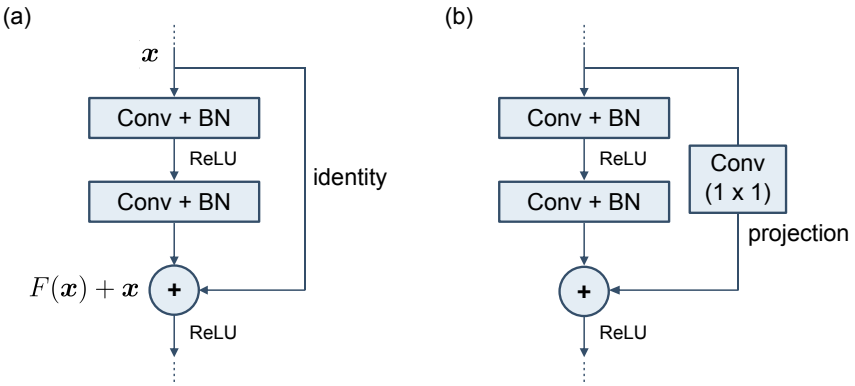

Figure 2.4: (a) Residual unit with identity shortcut. (b) Residual unit with projection shortcut.

2.2 Review of NAS methods

As seen in the last section, to create and train a CNN, the user must define not only the network layers and its characteristics, but also several hyperparameters, mostly related to the training procedure. The list of hyperparameters may include the number of examples in a batch, the learning rate, the optimizer to use along with its settings, the regularization methods, among others. These design choices require expert knowledge and even trial and error, as there are only general suggestions and no specific methodology to define them. In this context, researchers established the area of AutoML that attempts to automate these decisions completely [1].

Neural Architecture Search is the process of automating network engineering and can be considered a subfield of AutoML [1]. The idea of automatically designing neural networks has been studied before, but at a time when deep networks were not close to becoming a reality. The field of neuroevolution started with the purpose of using evolutionary algorithms to adjust the weights of neural networks [11]. The concept was extended, as several authors proposed evolutionary algorithms to adjust not only the weights of the networks but also their topologies [10–12]. The NEAT [11] and GNARL [10] are examples of such systems. Although successfully applied to smaller structures, the neuroevolution approaches do not directly scale to the context of deep networks, as they optimize connections at the neuron level [20].

The current success of deep networks, combined with the difficulty in creating them, instigated researchers again with the idea of design automation. NAS has recently become a relevant research field, as a variety of new solutions have rapidly emerged.

According to Hutter et al. [1], NAS can be characterized by three items: search space, search strategy, and performance estimation strategy. The first one is related to which architectures can be represented. The search strategy defines how to explore the search space, including the technique applied and the exploration-exploitation trade-off. Finally, the performance estimation is associated with the evaluation of the candidate networks [1]. In the next subsections, we review some NAS works, focusing on the search strategy. The search techniques include reinforcement learning, evolutionary algorithms, and others. To compare the approaches, we provide their results on the image classification benchmark dataset CIFAR-10 [15].

2.2.1 NAS with reinforcement learning

NAS research has gained more attention after Zoph and Le [2] presented competitive results applying reinforcement learning (RL) to search for deep architectures [1].

In RL systems, an agent interacts with the environment through a sequence of actions and rewards. The agent's goal is to select actions in a way that maximizes future reward [41]. A *policy* defines the behavior of the learning agent at any given time: it maps the perceived states of the environment to action decisions when in those states. Policies can be as simple as a look-up table or may involve complex search processes [42].

NAS is formulated as a reinforcement learning problem by considering the generation of a neural architecture as the agent's action. The action space is then identical to the network search space. The RL approaches are mainly distinguished by the agent's policy representation and the method to optimize it [1].

In policy gradient methods, for example, the policy is directly optimized by gradient descent. Zoph and Le [2] apply this technique in their RL based NAS: they use a recurrent neural network policy and train it with the REINFORCE algorithm. In their work, the recurrent network is responsible for generating variable-length strings that encode the candidate architectures. The decoded networks are trained using $45\,000$ examples from the $50\,000$ of the CIFAR-10 training set. The remaining $5\,000$ examples compose the validation set, which is used to calculate the accuracy of the structure. Then, they attribute this value as the agent's reward and use it to train the recurrent network policy. They achieved a test accuracy of 96.35% using 800 GPUs for more than three weeks. It is worth mentioning that after they find the structure, they conduct a grid search to optimize some numerical hyperparameters and retrain the model using these values.

Baker et al. [16] introduce MetaQNN, which uses Q-Learning to train a policy that selects CNN layers sequentially. For example, if the layer is convolutional, the options for kernel size and filters are $\{1, 3, 5\}$ and $\{64, 128, 256, 512\}$. They used ten GPUs for ten days to reach 93.08% of accuracy in the CIFAR-10 dataset.

Other researchers proposed RL based methods that, instead of searching for the entire network, focus on optimizing smaller blocks [17, 18]. A predefined *meta-architecture* determines the specific way to stack these blocks to build the final structure. This idea drastically reduces the search space and speeds up the search process, but introduces a significant bias to the networks. The meta-architectures restrict the possibilities of finding unusual structures and raises the question about how to define them [1].

Zoph et al. [17] followed this idea and applied a recurrent neural network policy, but now using Proximal Policy Optimization [43] to train it. They defined the NASNet search space for the cells, and attempt to learn two blocks: the *normal cell* and the *reduction cell*. The first one does not change the input feature map size, while the second cell reduces it by a factor of two. In the NASNet space, there is a set of 13 operations, such as 3×3 convolution and 7×7 max pooling, that can

be selected for a hidden state inside a cell. The number of filters in convolutional layers is predefined, but connectivity between the hidden states are learned. The final NASNet architecture is composed by stacking N normal cells, followed by a single reduction cell, repeated times. They were able to achieve 96.86% of accuracy in the CIFAR-10 dataset using 500 GPUs over four days.

Zhong et al. [18] adopted Q-Learning rather than policy gradient methods. Using 32 GPUs for three days, BlockQNN reached 96.46% of accuracy on CIFAR-10.

A different approach involves *reusing weights* when training networks during the search. In [25], the authors propose to explore the search space by applying transformations that preserve functionality so that weights can be reused. They employ a recurrent network policy, trained with REINFORCE, to increase the architecture depth or width using function-preserving operations, such as adding a layer initialized as identity. They obtained 95.11% of accuracy in the CIFAR-10 dataset using five GPUs for two days. Although the method does not search for cells, it relies on a seed network to begin the process. In this experiment, they set a small architecture as the starting point that already presented 87.07% of validation accuracy.

2.2.2 NAS with evolutionary algorithms

Since neuroevolutionary methods have been proposed earlier, a natural alternative to RL based approaches is evolutionary algorithms. However, as mentioned before, the former algorithms did not scale to the contemporary context of deep architectures. Recent evolutionary techniques applied to NAS use regular gradient-based training to adjust the network's weights, and the evolution occurs only in the structure space [1].

Evolutionary algorithms, as the name suggests, are inspired by the natural process of evolution. There are many variations of such methods, but they all follow the same basic idea [44]. A population of individuals represents candidate solutions to an optimization problem. All the individuals are evaluated so that a fitness value can be assigned to them. In each iteration of the algorithm (*generation*), the fittest individuals are selected to create a new population [45].

In [20], Cartesian Genetic Programming (CGP) is used to discover new architectures for the CIFAR-10 dataset. The CGP encoding scheme defines the network as a directed acyclic graph in a two-dimensional grid of computational nodes. They investigate two search spaces, which they name *ConvSet* and *ResSet*. The first one considers convolutional networks, while the second, residual networks. Their method allows for the search of multi-branch architectures, with skip connections, and they experiment with a maximum depth of 30 layers. They evaluate two individuals per generation by training them and calculating the validation accuracy. In

300 generations, they reported a test accuracy of 94.02% for the ResSet space using two GPUs for two weeks. For the ConvSet space, they ran the algorithm for 500 generations and reported a test accuracy of 93.25%.

Real et al. [19] proposed an evolutionary method that produces a fully trained network at the end, without the need for retraining. Following the idea of the successful NEAT algorithm [11], the authors start the evolution process from a single layer network and apply mutation operators that act in the structure, allowing it to grow. However, unlike NEAT, these operators act on layers instead of neurons. A regular training procedure is conducted for each network to obtain the individual's fitness (validation accuracy). The weights' values are inherited across mutations, whenever possible. The authors claim that the search space is fairly unrestricted and reported a test accuracy on the CIFAR-10 dataset of 94.60% using 250 GPUs for 256 hours.

Assunção et al. [32] proposed an approach that combines ideas from two evolutionary methods: Genetic Algorithms (GA) and Grammatical Evolution. The GA-level encodes the macrostructure, such as layer types. The grammatical level specifies the parameters and their valid ranges for each unit defined by the GA. After evolution, they retrain the final network five times and report the average accuracy. The authors provide interesting results concerning the learning rate schedule in this retraining phase. With no schedule, they obtained 88.41% of accuracy on CIFAR-10 and 92.51% when applying the same schedule as [20].

The meta-architecture idea also appears in evolutionary approaches for NAS [21, 26, 46]. Real et al. [21] apply an evolutionary algorithm to search for the normal and reduction cells in NASNet search space, as defined in [17]. New cells are generated by mutating parent cells, with simple modifications. Their mutation operator guarantees that the cell still lies in the NASNet space. At every iteration, instead of removing the worse individuals, they eliminate the oldest. They used 450 GPUs for seven days, and their best AmoebaNet-A reached 96.66% of test accuracy on CIFAR-10.

Liu et al. [26] presented a hierarchical genetic representation of directed acyclic graphs. At the lowest level of the hierarchy, the primitives are convolutions and pooling operations. They are assembled in graphs, generating higher level motifs, which can then be combined again. The authors define a small meta-architecture to insert the motifs in some of its layers so they can compute fitness via regular training. For the final network, they insert the evolved blocks in a bigger meta-architecture. They obtained an accuracy of 96.25% in the CIFAR-10 dataset, using 200 GPUs for 1.5 days.

The Genetic CNN proposed in [46] also defines a meta-architecture and evolves blocks of the structure, which are encoded as fixed-length binary strings.

The meta-architecture comprises a series of stages. Within each stage, the convolutional operations have the same number of filters, and the spatial dimensions of data remain unchanged. They applied the method to the CIFAR-10 dataset, running the evolution for 50 generations in 17 GPU days, and reached 92.90% of accuracy.

2.2.3 NAS with other methods

With the growing interest in NAS, new techniques have been emerging beyond the scope of evolutionary algorithms or reinforcement learning. Authors have proposed NAS methods using Particle Swarm Optimization [47], Bayesian optimization [22, 48], and even Gradient Descent [49, 50].

The network transformation idea mentioned in Subsection 2.2.1 – also called *network morphism* – is present in other works as well. The Auto-Keras framework [48] uses Bayesian Optimization to guide network morphism. Elsken et al. [51] propose a hill-climbing procedure that increases structures by applying network morphisms.

Liu et al. [49] introduce DARTS, which formulates the architecture search problem in a continuous space. They apply softmax over all the discrete structural choices to relax the search space to be continuous. The network is then optimized with respect to its validation performance by gradient descent. Their best accuracy in CIFAR-10 was 97.24% consuming five GPU days, which was obtained by searching for a cell and using a meta-architecture.

Alternative ideas, such as GDAS [50], NAO [52], and XNAS [53], also consider NAS in a continuous search space for cells. XNAS uses a modified DARTS search space, and interpret NAS as an online selection task. XNAS reached a remarkable result on CIFAR-10: 98.40% of test accuracy in 0.3 GPU day. However, it is worth mentioning that besides using a meta-architecture with a predefined number of filters, the retraining procedure is quite complicated. They train the final network for 1500 epochs using a particular learning rate schedule, along with six different regularization techniques.

2.2.4 Final remarks

Table 2.1 summarizes the results and resource consumption for several NAS works. The consumption is reported as GPU days, which is defined as the number of days a single GPU would take to run the algorithm. Note, however, that a fair comparison would also require to consider the GPU hardware specification, which is not always available. The NAS methods described in this section show excellent results compared to hand-designed architectures. On the other hand, the

approaches that use an extensive search space are significantly more expensive regarding computational cost. The solutions proposed to reduce this cost frequently impose a strong bias to restrict the search space. The trade-off between generality and computational cost is a critical point in the NAS field, and current results evidence the need for further development.

Table 2.1: Comparing hand designed models (top) and NAS methods (bottom), by accuracy, number of parameters and GPU days of search. The '*' marks the method that used other dataset for the search and applied the network on CIFAR-10.

	accuracy (%)	# params.	GPU days
ResNet [9]	93.57	1.7M	-
VGG [7] as reported by [20]	92.06	15.2M	-
Network in Network (NiN) [8]	91.19	-	-
Maxout [54]	90.70	-	-
XNAS [53]	98.40	7.2M	0.3
DARTS [49]	97.24	3.3M	5
NASNet-A [17]	96.86	3.3M	2000
AmoebaNet-A [21]	96.66	3.2M	3150
Block-QNN-S [18]	96.46*	39.8M	96
NAS [2]	96.35	37.4M	22400
Hierarchical Evolution [26]	96.25	-	300
EAS [25]	95.11	-	10
Large-scale Evolution [19]	94.60	5.4M	2670
CGP-CNN (ResSet) [20]	94.02	1.68M	28
CGP-CNN (ConvSet) [20]	93.25	1.52M	-
MetaQNN [16]	93.08	11.18M	100
Genetic CNN [46]	92.90	-	17
NASBOT [22]	91.31	-	1.67

3
Quantum-inspired Evolutionary Algorithms

This chapter introduces the idea of quantum computing and quantum-inspired evolutionary algorithms.

3.1
Quantum computing

In the early 1980s, Richard Feynman remarked that classical computers could not simulate some quantum mechanical effects efficiently. This observation supported the idea that general computation may be executed more efficiently if it makes use of these quantum effects. The new idealized device is called the quantum computer [55]. *Quantum computing* is then the term used to describe the computational processes based on these quantum principles [56, 57].

In a classical computer, the smallest unit of information is the binary digit or *bit*, which can assume the values of 0 or 1. Quantum computers, on the other hand, manipulate quantum bits or *q-bits* [27, 56, 58]. A q-bit is a unit vector in a two-dimensional complex vector space for which the orthonormal basis (in Dirac notation) $|0\rangle, |1\rangle$ has been fixed. The basis states represent the classical bit values 0 and 1, respectively [55]. A q-bit, in contrast to the classical bit, can be in a superposition of states, that is, the q-bit state $|\Psi\rangle$ can be represented as a linear combination of $|0\rangle$ and $|1\rangle$ [27, 28, 55]:

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{3-1}$$

where α and β are complex numbers such that:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{3-2}$$

When the state is measured, the q-bit collapses to the values 0 or 1 with probabilities $|\alpha|^2$ and $|\beta|^2$, respectively. The q-bit state can be modified by means of *quantum gates*. A quantum gate is a unitary operator that acts on the q-bit basis. It can be represented by a matrix M that, in order to preserve orthogonality, must satisfy $M^\dagger M = I$, where $M^\dagger$ is the conjugate transpose of M. One can think of these unitary transformations as being rotations of the q-bit vector space [28, 55].

Although quantum computing offers great potential in terms of processing capacity [56], the research community still needs to overcome many challenges so that quantum computers can become readily available[1]. The idea of quantum-inspired computing is to create classical algorithms that take advantage of quantum physics principles but can be executed in classical computers [56, 57, 59]. The term *quantum-inspired* was selected to distinguish these algorithms from pure quantum computation, which is firmly based on the concepts of quantum mechanics. Similar inspirations have emerged in different areas, such as artificial neural networks motivated by neurobiology or genetic algorithms based on natural selection [57].

3.2 Quantum-inspired evolutionary algorithms

Embracing the quantum inspiration idea, Han and Kim [28, 60] presented the first practical quantum-inspired evolutionary algorithm (QIEA). A QIEA applies quantum computing principles, such as quantum bits and superposition of states, to solve optimization problems [27, 28]. Like other evolutionary algorithms, a QIEA is characterized by the individual representation, the evaluation function, and the population dynamics.

Figure 3.1 shows the pseudocode for a canonical evolutionary algorithm (EA) [44]. The EA maintains a population of individuals ($P(t)$) that represents possible solutions to the problem in hand. Every solution in $P(t)$ is evaluated, and a fitness value is assigned to each one. At the beginning of the loop (line 6 in Figure 3.1), *genetic operators*, such as *mutation* and *crossover*, modify some members of $P(t)$. This process generates new candidate solutions that are also evaluated. The next population is created by selecting the best individuals in $S(t)$. We proceed with the iterations (which we call *generations* in the EA context) until we reach some stopping condition, e.g., a specific number of generations. When the algorithm converges, the best individual in $P(t)$ should represent a solution that is close to the optimal one [45].

There are several EAs available in the literature, and they differ mainly in terms of the following aspects [45]:

1. how the individuals are encoded;
2. which genetic operators are applied;
3. the selection mechanism used to generate the new populations.

One key concept of QIEAs that distinguishes them from other EAs is the *quantum population*. The quantum population is the core of QIEA [61]. It represents

[1] https://www.research.ibm.com/ibm-q/learn/what-is-ibm-q

```
1:  begin
2:      t ← 0
3:      Initialize P(t)
4:      Evaluate P(t)
5:      while t ≤ T do
6:          S(t) ← Vary P(t)
7:          Evaluate S(t)
8:          P(t + 1) ← Select S(t)
9:          t ← t + 1
10:     end while
11: end
```

Figure 3.1: Simple evolutionary algorithm pseudocode.

a superposition of states that covers the search space, or more specifically, each quantum state characterizes a possible solution. Quantum individuals cannot be directly evaluated: they must be observed to generate classical individuals. In other words, since a quantum individual represents many quantum states, it can only be evaluated when it collapses to a single one.

Figure 3.2 shows a QIEA pseudocode [28, 56, 60], which is similar to the canonical EA in Figure 3.1. The QIEA procedure includes some unique steps, such as the observation of the quantum population $Q(t)$ (lines 4 and 9) and the update of the quantum individuals (line 11).


```
1:  begin
2:      t ← 0
3:      Initialize Q(t)
4:      Generate classical population P(t) observing Q(t)
5:      Evaluate P(t)
6:      Store the best solutions of P(t) in B(t)
7:      while t ≤ T do
8:          t ← t + 1
9:          Generate classical population P(t) observing Q(t − 1)
10:         Evaluate P(t)
11:         Q(t) ← Update Q(t)
12:         Store the best solutions of P(t) and B(t − 1) in B(t)
13:     end while
14: end
```

Figure 3.2: A QIEA pseudocode.

In fact, a special update is needed as a consequence of the quantum individual representation. The classical variation operators – crossover and mutation, in their traditional forms – are not adequate for the quantum individuals. The quantum individual represents the search space in a probabilistic way and we would like to have a mapping of such space that favors promising solutions. For example, suppose

we have a quantum gene represented by a q-bit $q = [\alpha \ \ \beta]^T$, with $\alpha = \beta = 1/\sqrt{2}$. Suppose further that, during evolution, candidate solutions with this gene set to 0 show better fitness scores. The algorithm should then increase α and decrease β, so that the probability of the q-bit collapsing to state 0 approaches 1.0. Notice, however, that classical operators can still be applied to the classical individuals if one wishes to [12, 30, 56, 61].

To further describe and illustrate the QIEA concepts and advantages, we will go through the already mentioned algorithm designed by Han and Kim [28, 60]. Their goal at the time was to solve combinatorial optimization problems, such as the knapsack problem[2]. They define a quantum population with N individuals $Q(t) = \{\boldsymbol{q}_1^t, \boldsymbol{q}_2^t, ...\boldsymbol{q}_N^t\}$. Each quantum individual $\boldsymbol{q}_i^t$ is a string of q-bits [28, 60]:

$$\boldsymbol{q}_i^t = \left[\begin{array}{c|c|c|c} \alpha_{i1}^t & \alpha_{i2}^t & ... & \alpha_{iG}^t \\ \beta_{i1}^t & \beta_{i2}^t & ... & \beta_{iG}^t \end{array}\right] \tag{3-3}$$

where G is the the string length (or the number of *quantum genes*), t is the generation number and $i = 1, 2, ..., N$. Similar to the normalization in Equation 3-2, $|\alpha_{ij}|^2 + |\beta_{ij}|^2 = 1$.

The quantum individual $\boldsymbol{q}_i$ can represent a superposition of 2^G states or binary solutions. Equation 3-4 shows an example of a quantum individual consisting of two q-bits:

$$\left[\begin{array}{c|c} \frac{1}{\sqrt{2}} & \frac{1}{2} \\ \frac{1}{\sqrt{2}} & \frac{\sqrt{3}}{2} \end{array}\right] \tag{3-4}$$

To calculate the probability of the state $|01\rangle$, we should first calculate the probability amplitude associated with this state. The amplitude is obtained by multiplying the positions $(0, 0)$ and $(1, 1)$ of the quantum individual matrix in Equation 3-4. So, the probability of state $|01\rangle$ is given by:

$$P_{|01\rangle} = \left(\frac{1}{\sqrt{2}} \times \frac{\sqrt{3}}{2}\right)^2 = \frac{3}{8} \tag{3-5}$$

In this example, one can see that a single quantum individual can represent four possible solutions. The classical binary representation would need four strings to represent the same amount of information. The authors claim that q-bit encoding better characterizes the population diversity compared to other EAs, as it represents a linear superposition of states probabilistically [28, 60].

[2]The knapsack problem is defined as follows. Given a set of m items and a knapsack with limited capacity, select a subset of the items to maximize the total value subject to the total weight limit.

Now that the representation is defined, we can explore the steps of the algorithm. The first QIEA procedure is the quantum population $Q(t)$ initialization. The strategy is to assign values to each quantum gene j of every individual i so that all possible states have the same initial probability or, more specifically [28, 60]:

$$\alpha_{ij}^0, \beta_{ij}^0 = \frac{1}{\sqrt{2}} \qquad i = 1, 2, ..., N; j = 1, 2, ...G \tag{3-6}$$

Next, we need to generate classical individuals by observing the quantum population (line 4 of Figure 3.2). Notice that the observation here is an analogy, as we are working with classical computers. Therefore, to generate a binary solution $\boldsymbol{p}_i^t$ to be evaluated, one selects a state for each gene independently.

The solution $\boldsymbol{p}_i^t$ is a string of G bits, formed by randomly selecting each gene to be either 0 or 1, respecting the probabilities $|\alpha_{ij}|^2$ and $|\beta_{ij}|^2$ [28, 60]. In practical terms, this can be achieved by sampling from a uniform distribution in the interval $[0, 1]$ and comparing the selected number with $|\alpha_{ij}|^2$. If this number is lower than $|\alpha_{ij}|^2$, the state is 0; otherwise, it is 1. This process is repeated for all N individuals to generate the classical population $P(t) = \{\boldsymbol{p}_1^t, \boldsymbol{p}_2^t, ..., \boldsymbol{p}_N^t\}$.

Once generated, the classical population $P(t)$ is ready for evaluation (line 5); the best solutions are stored in $B(t) = \{\boldsymbol{b}_1^t, \boldsymbol{b}_2^t, ..., \boldsymbol{b}_N^t\}$ (line 6). At generation 0, $B(t) = P(t)$. This ends the initial procedures, and we begin the loop of generations by the observation process (line 9), followed by the evaluation step (line 10).

As previously noted, we must update the quantum population so it can favor the generation of promising solutions (line 11). The authors in [28, 60] propose the use of a rotation quantum gate to update the quantum population:

$$U(\Delta\theta_j) = \begin{bmatrix} \cos(\Delta\theta_j) & -\sin(\Delta\theta_j) \\ \sin(\Delta\theta_j) & \cos(\Delta\theta_j) \end{bmatrix} \tag{3-7}$$

where $\Delta\theta_j$ is a rotation angle for each q-bit toward either 0 or 1 state depending on its sign. This matrix must be applied to one q-bit at a time [56]:

$$\begin{bmatrix} \alpha_j' \\ \beta_j' \end{bmatrix} = U(\Delta\theta_j) \begin{bmatrix} \alpha_j \\ \beta_j \end{bmatrix} \tag{3-8}$$

where $j = 1, 2, ..., G$.

The angles $\Delta\theta_j$ are obtained according to a lookup table designed by the authors, which depends on the application. These values are a function of the j^{th} bit of the best solution $\boldsymbol{b}_i^t$ and the j^{th} bit of the binary solution $\boldsymbol{p}_i^t$.

Notice that the best solutions influence the quantum individuals' update, so the quantum gate will gradually rotate the q-bits in the direction of promising solutions. Consequently, the q-bits will progressively converge to a single state – the

optimal solution. According to the authors [28], by applying this update mechanism, QIEAs can handle the balance between exploration and exploitation. This feature can be seen as a significant advantage of QIEAs over other EAs. Usually, traditional EAs need to balance the use of crossover and mutation operators to handle the exploration/exploitation equilibrium. Finding this equilibrium can be a difficult task.

The last step in the loop consists of storing the best solutions among $B(t-1)$ and $P(t)$ into $B(t)$ (line 12). In practical terms: the solutions from the current generation and the best solutions are ordered by fitness value, and the best ones from the group are selected.

This first QIEA inspired other researchers over the years, and different versions of the algorithm have been proposed to solve a variety of problems.

In [62], the authors introduce a new QIEA for ordering problems, such as the traveling salesman problem and the vehicle routing problem. They adopt a special quantum representation based on a vector of q-bits, in which only some binary states are allowed. The results show that the order-based QIEA outperformed the traditional order-based genetic algorithm.

The work in [63] also applies the q-bit representation and introduces a K-means clustering algorithm based on QIEA. Likewise, in [52], SVM parameters were tuned using a QIEA.

More recently, Ramos et al. [64] applied q-bit QIEA for feature selection in a classification task of electroencephalography data. They compared the QIEA approach with a classical genetic algorithm. The results for QIEA showed significantly better performance regarding convergence time.

Besides the quantum bit representation, researchers developed other encodings during the years. An example is the extension of the q-bit idea to quantum digits or *qudits* [65–67]. Additionally, in [29] and [30], the authors introduce a new quantum individual better suited to solve numerical problems, as it respects the condition that the observable states are continuous rather than discrete. The quantum genes in [30] are formed by a probability density function, more specifically a square pulse defined by two variables: the mean μ and the width σ. A set of genes then composes the quantum individuals, and the parameters μ and σ of each pulse are modified during the evolution in order to shift, narrow, or expand the probability densities. They tested the algorithm on several benchmark functions and compared it to traditional methods and q-bit QIEA. Their results confirm the advantages of direct numerical representation [30].

QIEAs that combine binary and real representations have also been proposed for problems in which both real and categorical variables need to be evolved.

The work in [12] proposes a hybrid QIEA that addresses a neuroevolution task, which includes feature selection, the definition of some parameters of an MLP

(Multilayer Perceptron) classifier network, and its weight optimization. The binary part of the quantum individuals is responsible for encoding the parameters that have a finite number of possibilities, naming: the number of hidden neurons, the type of activation function, and the variables that will serve as input to the network. The numerical part encodes the neuron weights, that lies in the universe of real numbers.

Another hybrid QIEA is employed to optimize the input features and parameters of a spiking neural network in the unsupervised learning context [68]. The authors applied the model to eight benchmark datasets and presented higher clustering accuracy than non-optimized spiking networks.

4
Q-NAS

This chapter introduces the proposed Quantum-inspired Neural Architecture Search (Q-NAS). First, an overview of the algorithm is provided. Then, we describe the individual representation and its specific methods. We complete the chapter presenting the steps of the algorithm in detail.

4.1
Q-NAS overview

Q-NAS is a quantum-inspired evolutionary algorithm focused on the deep neural architecture search problem. The goal of Q-NAS is to automatically design deep networks to execute a predefined task. Usually, in the NAS literature, this automation considers only the network architecture itself, disregarding its hyperparameters. Q-NAS aims to address the automation at a more challenging level, which includes finding the best architecture and optimizing some of its hyperparameters. This approach goes in the direction of complete automation, which is the goal of AutoML.

In this work, we frame Q-NAS in the context of image classification tasks. We made this decision based on the fact that most of the NAS research focus on this application [1], which provides us with a solid base to compare our work. However, we defined this context with the only purpose of developing experiments to test Q-NAS applicability and performance. Q-NAS is not restricted to image classification tasks, and it can be used in other application domains.

The scheme in Figure 4.1 illustrates the Q-NAS context, considering a classification task. The user specifies the search space for both the hyperparameters and the network structure. The first specification consists of defining the value range for each hyperparameter. The second one comprises the selection of building blocks (layer functions) that will be used to assemble the networks. The output of the system, as illustrated in Figure 4.1, is a network description along with the hyperparameters' values.

We remark that Q-NAS does not evolve the weights of the networks. Regular gradient based training is conducted for this purpose, as will be described in Section 4.4.

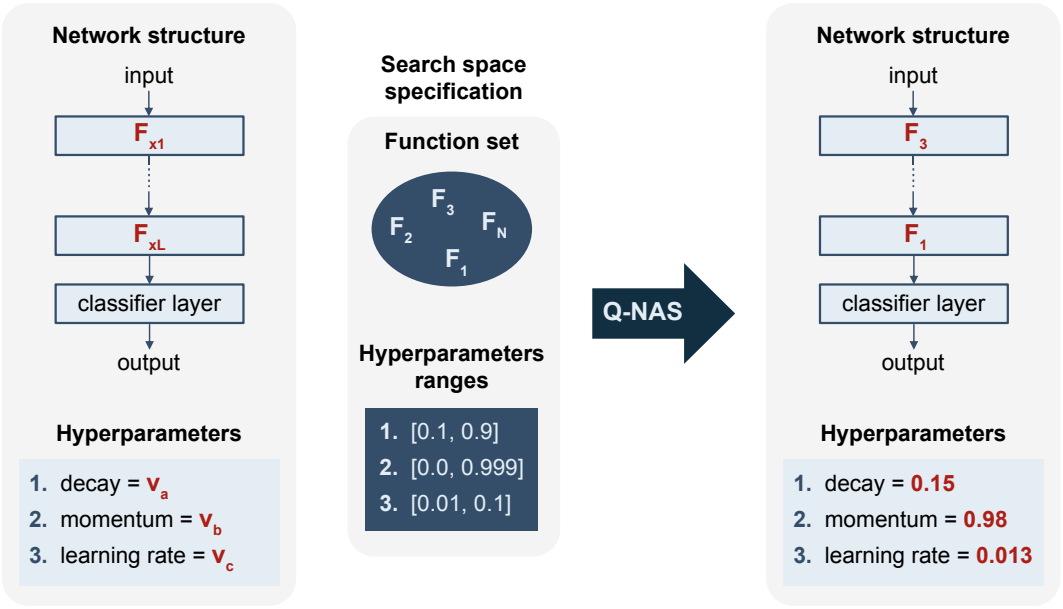

Figure 4.1: Q-NAS context in a classification task. We define the network building blocks in the function set. The numerical hyperparameters ranges for the search are also specified. Q-NAS provides the optimized network structure and hyperparameters' values.

Considering that we want to address the network structure design and the hyperparameter optimization, Q-NAS must be able to represent both of these elements. Thus, we chose to divide the quantum chromosome into two parts: one is responsible for encoding the network structure (categorical) search space, and the other encodes the numerical space of some hyperparameters.

Figure 4.2 shows a simplified Q-NAS flowchart: in the loop of generations, we sample solutions from the quantum individuals, evaluate them, and update the quantum population. The overall steps are similar to those from the q-bit QIEA, introduced in Section 3.2. The dark boxes indicate the steps that must consider the particular quantum representation. Each part of the chromosome has a specific observation and update procedures. Therefore, as depicted in Figure 4.2, we observe each part of the chromosome separately, but the classical individual is evaluated as one unique solution. The same reasoning applies to the update procedure.

The quantum individual representation is the essence of Q-NAS, and the primary factor that discriminates it from the previously discussed QIEA. The next sections cover the specific details and methods of each part of the quantum individual: the numerical hyperparameters representation and the network structure representation.

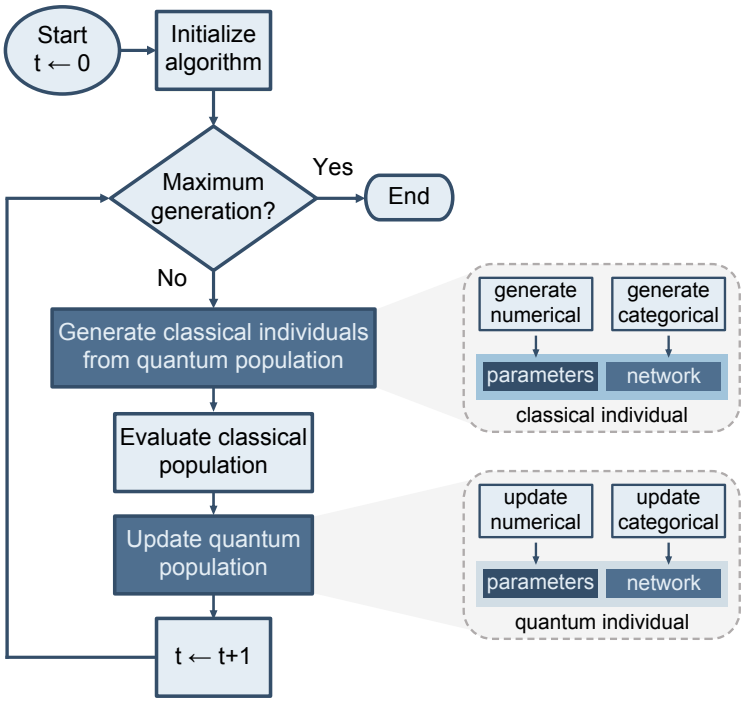

Figure 4.2: Q-NAS flowchart. The dark boxes highlight the Q-NAS steps that are executed separetely for each part of the chromosomes.

4.2 Numerical hyperparameters representation

The numerical representation is based on the quantum individuals proposed in [30,61]; they respect the fact that the search space is continuous. The idea is to define a probability density function (PDF) $p_{ij}(x)$ for each variable h_j to be optimized, respecting its domain. Then, if we have a set of G numerical hyperparameters to optimize, a quantum chromosome $\boldsymbol{q}_i$ can be defined as an array:

$$\boldsymbol{q}_i = [p_{i1}(x), ..., p_{iG}(x)] \tag{4-1}$$

Each function $p_{ij}(x)$ represents the probability of observing the quantum gene in a certain range of values when the superposition of states collapses. The PDFs can be defined in terms of the wavefunction $\Psi_{ij}(x)$ and its complex conjugate $\Psi^*_{ij}(x)$, associated to the quantum gene j:

$$p_{ij}(x) = \Psi^*_{ij}(x)\Psi_{ij}(x) \tag{4-2}$$

Following [12, 30, 61], we chose a uniform PDF (square pulse) because of its unbiased characteristics and simplicity. The uniform distribution allows us to represent the initial search space without introducing a bias toward any specific value. Additionally, it facilitates the observation procedure, as described later.

The square pulse $p_{ij}(x)$ can be defined by its lower and upper limits, l_{ij} and u_{ij}, respectively. These are the minimum and maximum values the numerical

hyperparameter h_j can assume. Figure 4.3 shows an example of a square pulse defined by $l_{ij} = 0.1$ and $u_{ij} = 0.9$, along with its respective cumulative distribution function (CDF).

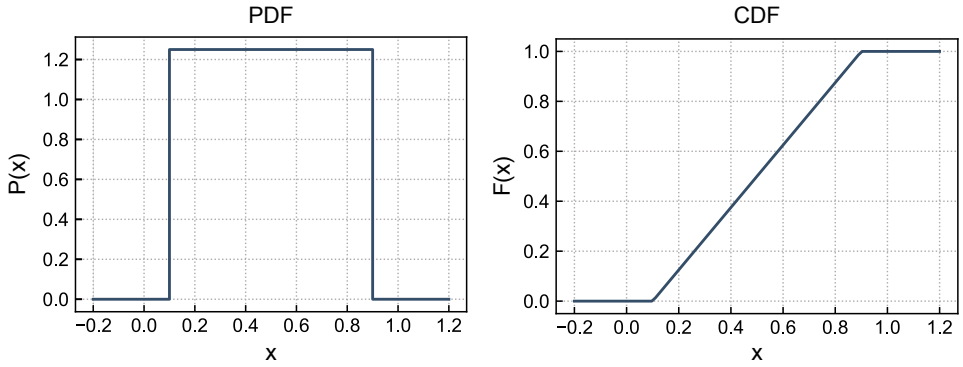

Figure 4.3: Example of a uniform PDF and its corresponding CDF.

In Q-NAS, each numerical quantum gene consists of the pair (l_{ij}, u_{ij}), instead of the mean $\mu_{ij} = (l_{ij} + u_{ij})/2$ and width $\sigma_{ij} = u_{ij} - l_{ij}$ used in [12, 30, 61]. This modification simplified the quantum update strategy, as detailed later in this section.

The observation procedure for the numerical quantum individuals consists of sampling from the PDFs. Like in the q-bit QIEA, we observe each gene independently. The Numpy library [69], for example, provides functions to sample from a variety of distributions. However, we decided to use a standard sampling procedure that, in the case of uniform PDFs, can make our observation process more efficient. Despite its simplicity, we detail the sampling method to explain this benefit. First, we need to calculate the CDF $F_{ij}(x)$ by integrating the uniform PDF $p_{ij}(x)$:

$$F_{ij}(x) = \begin{cases} 0 & \text{if } x < l_{ij} \\ \frac{x - l_{ij}}{u_{ij} - l_{ij}} & \text{if } x \in [l_{ij}, u_{ij}) \\ 1 & \text{if } x \geq u_{ij} \end{cases} \tag{4-3}$$

as illustrated by the example in Figure 4.3. The sampling procedure comprises simple steps: (1) generate a random number r in the interval $[0, 1]$; (2) find the x such that $F_{ij}(x) = r$. The scheme in Figure 4.4 illustrates these steps, using the same CDF of Figure 4.3.

As $F_{ij}(x)$ inside the domain is a straight line, its inverse is easily determined, and x is calculated by:

$$x = r \times (u_{ij} - l_{ij}) + l_{ij} \tag{4-4}$$

Note that the observation procedure is reduced to the evaluation of a simple expression. Since the observation is independent for each gene, we can evaluate

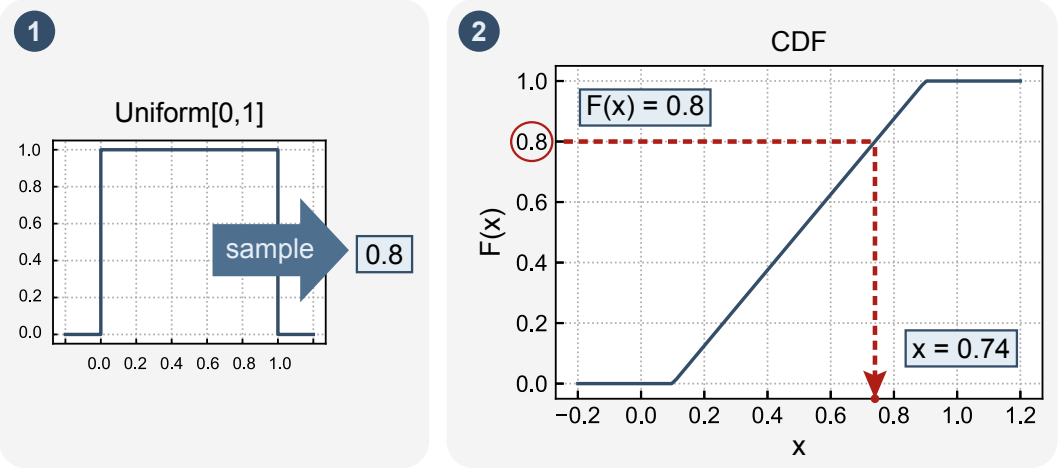

Figure 4.4: Scheme illustrating an example of the sampling procedure.

Equation 4-4 for the entire array of numerical quantum genes at once, as a vector operation.

Once we have generated the classical individuals, it is possible to apply traditional recombination and mutation operators. In the case of the numerical variables, there are a lot of variation operators available in the literature. However, as discussed in [30], the quantum individuals already provide a good diversity representation, and thus the mutation operator might be unnecessary.

Following [12, 30], we decided to apply the arithmetic crossover operator [45] on the numerical part of the classical population. A random mask based on a *crossover rate* is generated to select which genes should be modified: it chooses the indices of chromosomes i and genes j. The operation is defined in Equation 4-5:

$$C'[i,j] = C[i,j] + r \cdot (P[i,j] - C[i,j]) \tag{4-5}$$

where (i,j) are the indices of the genes selected by the random mask, r is a random value in the interval $[0,1]$, C is the current classical population, and P is the classical population stored in the previous generation.

A final aspect of the numerical representation that needs to be addressed is how to update the quantum individuals. In general, we want to reduce the search space and also map the most promising search areas. Regarding our numerical quantum individuals, we can accomplish this by reducing and shifting the square pulses.

The works in [30] and [56] use the $1/5$ rule, which can reduce or increase the square pulse width. The mean of the pulses is modified according to the value of the classical genes. More specifically, the center of the pulse is shifted in the direction of a classical gene by a factor λ selected by the user.

The authors in [12] propose a heuristic to update the quantum individuals using the best classical ones. They shift the mean μ of the pulse in the direction

of the best classical values and modify the pulse width σ according to the range of classical values of the current best population. A random mask is used to define which quantum genes will be updated. The random mask is generated based on the parameter *update_quantum_rate*, which is equivalent to the crossover rate presented before. We selected this heuristic, as it uses information from the best individuals in both the center and the width of the pulses. Equation 4-6 shows its original formulation based on μ_{ij} and σ_{ij}:

$$\begin{aligned} d^t &= \max_{i=1..N} c_{ij}^t - \min_{i=1..N} c_{ij}^t, \\ \mu_{ij}^{t+1} &= \mu_{ij}^t + r * \left(c_{ij}^t - \mu_{ij}^t\right), \\ \sigma_{ij}^{t+1} &= \sigma_{ij}^t + r * \left(d^t - \sigma_{ij}^t\right) \end{aligned} \tag{4-6}$$

where r is a random number in the range $[0, 1]$ and c_{ij}^t is the jth current value of classical individual i. Note that σ_{ij} can increase or decrease depending on the difference between σ_{ij}^t and d^t. Rewriting Equation 4-6, we adapted the heuristic to our representation of lower and upper limits, l_{ij} and u_{ij}, respectively:

$$\begin{aligned} l_{ij}^{t+1} &= l_{ij}^t + r * \left(c_{ij}^t - l_{ij}^t - \frac{d^t}{2}\right), \\ u_{ij}^{t+1} &= u_{ij}^t + r * \left(c_{ij}^t - u_{ij}^t + \frac{d^t}{2}\right) \end{aligned} \tag{4-7}$$

The heuristic presented in [30, 56] and the one just mentioned do not prevent the pulses from stepping outside the initial domain. The authors in [12,30,56] correct the pulse width but not the mean, which can make the sampling procedure generate values outside the original domain. To address this issue, they assign the boundary values to any invalid sample. In practice, this procedure deforms the PDF, which is no longer uniform. Figure 4.5 illustrates this problem: if the blue pulse is shifted to the position of the red one, all the probability in the gray area goes to the boundary value of the pulse (0.9 in this example).

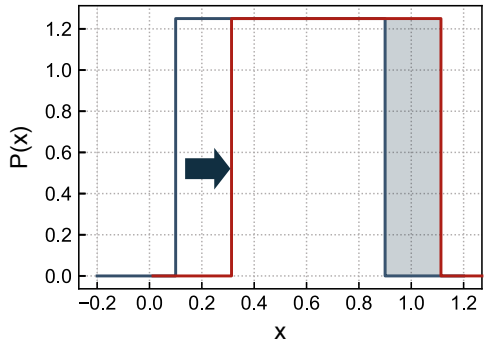

Figure 4.5: Example of possible problem with the update procedures in [12, 30, 56].

To address this issue, we reformulated the previous approach, truncating the pulse after its update if it goes outside the initial domain. By adopting the lower and upper limits to represent the pulses instead of the center and width, the domain checking and pulse truncation are straightforward. In our reformulation, we can directly compare the pulse's lower and upper limits with the domain minimum and maximum allowed values, and truncate the values if necessary. To sample new values, we use Equation 4-4 that only requires the lower and upper limits. If the limits are inside the domain, Equation 4-4 does not generate invalid values and, therefore, no additional correction is needed. Figure 4.6 shows a PDF at the beginning and the end of evolution. Notice the significant reduction in the x range in the final pulse, although its area is still 1.0.

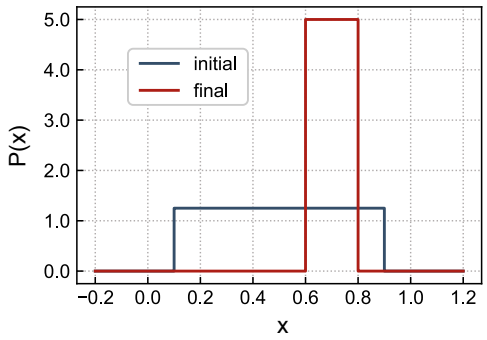

Figure 4.6: Example of a PDF at the beginning and the end of evolution.

Now that we detailed the particular aspects of the numerical representation, we will do the same for the network encoding in the next section.

4.3 Network representation

For the numerical part of the chromosome, we only defined the quantum representation, as the classical individuals are already the values that will be used in the evaluation step. In the network architecture part, we also need to define the classical individual form. For complex entities, such as network structures, direct representations are usually difficult to design. It is possible to create an intermediate representation to encode the complex object so the algorithm can operate on it. A decoding procedure is required to map back the representation to the real object.

We represent networks in a *chain-like* structure with a fixed size L, in which every node has a function associated with it. These functions can be designed to be as simple as a network layer function, or they can be a block with several layers and skip connections. As we are currently working with image classification tasks, the last network layer is fixed to be a classifier (fully connected) layer. Figure 4.7 shows the chain-like structure and some examples of functions.

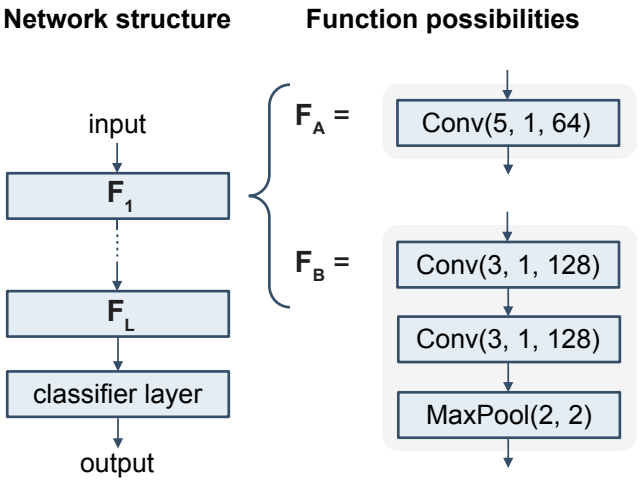

Figure 4.7: Network representation and function possibilities. The functions for each node can be as simple as one unique layer (F_A) or a more complex structure as F_B.

In the current version of Q-NAS, skip connections between nodes are not allowed. Such ramifications, or any other, can only appear when encapsulated in a function (node); our network representation remains as a chain structure.

The user specifies to the algorithm a list of predefined functions that will form the search space for every node in the network. Note that it is also possible to include in the list a *NoOp* function that assigns a *no operation* process to the node. *NoOp* allows us to represent variable-length networks, even though we fix a maximum size for the structure. This way, we can simplify the network representation, as it does not need to handle variable-size architectures, but preserve flexibility regarding the network depth. The *NoOp* idea is present in Genetic Programming algorithms [70].

The predefined function names are mapped to integers in the range $[0, M-1]$ so we can define our classical individual $\boldsymbol{p}_i$ as an array of integers:

$$\boldsymbol{p}_i = [g_{i1}, ..., g_{iL}]; \qquad g_{ij} \in [0, M-1] \tag{4-8}$$

where L is the number of nodes in our networks, and M is the number of functions available in the search space.

At this moment, we have the two essential ingredients to elaborate the quantum individual for the network structure: the classical representation and the search space characteristics. The network is described as a list of nodes and our search space is categorical and finite.

Considering these aspects, the quantum individual defines a probability mass function (PMF) for each node in the structure. A quantum gene encoding a single node is then represented by an array:

$$\boldsymbol{g}_j = [x_{j1}, ..., x_{jM}]; \quad x_{jk} \in [0.0, 1.0); \quad \sum_{k=1}^{M} x_{jk} = 1.0 \tag{4-9}$$

If we have N individuals, a network size L and M functions in the function list, the quantum population is an array of shape (N, L, M). All nodes start with the same PMF, but the user can choose the specific initial probability values for each function. This means that it is possible to start the evolution giving an initial bias to one or more functions.

The observation procedure, as before, requires the process of sampling from the probability distributions. The difference from the numerical case is that we are now dealing with categorical variables. We use the NumPy [69] implementation to sample from discrete distributions. Similar to the other cases, each gene is sampled independently.

Figure 4.8 summarizes the entire generation process for the network part of the chromosome: from the user input parameters to the final decoded network. The decoding process consists of simply reversing the integer map to get the node function names.

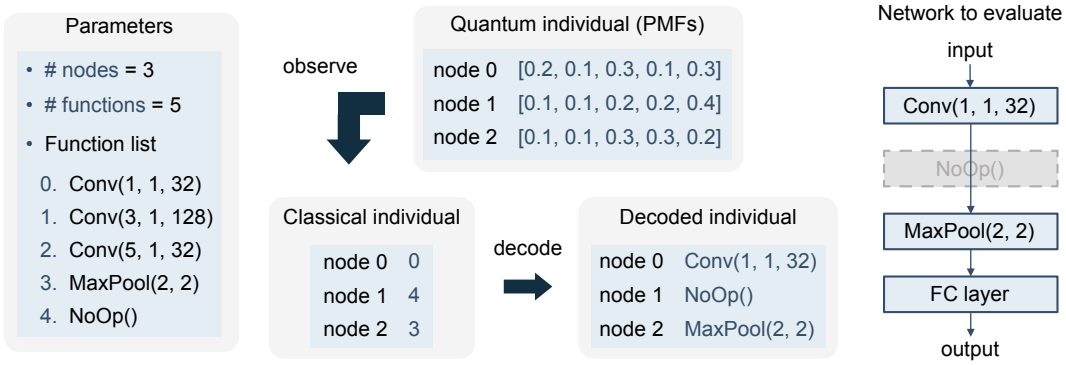

Figure 4.8: Network quantum individual and a generated architecture. The Q-NAS' parameters are listed on the left. *Conv(k, s, f)* stands for a convolution layer with kernel size $k \times k$, stride s and f filters. The observation of the quantum individual is carried out by sampling from the PMF of each node. The decoding process is a mapping from integers to function names. The final architecture includes a fully connected (FC) classifier layer at the end of the structure.

Unlike the described numerical representation, our network quantum individuals can lead to invalid structures. To illustrate this problem, imagine we have a pooling operation in our function list that reduces the feature map size in half. For a given input image size, there is a maximum number of times this pooling function can appear in the network before the feature map reaches unit pixel size. However, since we sample each node independently, we cannot handle this issue directly. A possible solution, as applied in [20], is to verify the structure and keep resampling individuals until a valid one is found. This approach requires the definition of a verification scheme and can lead to long sampling loops. Another idea is to give zero fitness to invalid networks and let the evolution itself take care of the prob-

lem. Nevertheless, one can argue that this method can make the algorithm miss the opportunity to evaluate potentially useful structures at every iteration.

Considering these points, we decided on a third option that involves penalizing invalid architectures. First, we developed a simple procedure to correct an invalid structure: when building the decoded network for evaluation, we ignore all pooling operations that appear after the allowed number is reached. This means that if the maximum is 5 and the decoded list has 6 pooling functions, the last one is ignored. The parameter *penalize_number* defines the maximum number of reducing layers a network can have without being subject to penalization. Every additional reducing layer will decrease 0.01 from the fitness value[1]. For example, if *penalize_number* $=$ 3 and the network has 5 reducing layers, its fitness is reduced by 0.02. Instead of waiting for correctly sampled individuals, this approach corrects them at evaluation time, so they can be trained and then penalized. It also differentiates networks with a few invalid layers from others that greatly exceed the *penalize_number*.

Finally, the quantum update procedure must be created for the network part of the chromosome. We also followed the idea of using the best classical individuals information to modify the quantum population.

We designed the simple heuristic described in Figure 4.9, which is applicable for the discrete PMFs. Our goal is to increase the probability of a promising function in a node by a random factor, which can assume the maximum value of 0.05. The other probabilities in the same node must be reduced to guarantee that the total sum is 1.0. We decrease the other probabilities proportionally to their current size. This proportional decrease ensures that small probabilities will never get negative. The maximum update value was chosen to be conservative, but it can be increased for future testing.


```
1: begin
2:     Generate random mask, based on update_quantum_rate
3:     Chosen nodes positions = idx
4:     for each node position i in idx do
5:         Get best_classical_individuals[i] function f.
6:         Increase the probability for f in node i by:
               update_value = random() * 0.05
7:         Subtract update_value from the probabilities other than f in node i
            proportionally to their current size
8:     end for
9: end
```

Figure 4.9: Q-NAS network quantum update. The loop is only depicted here for clarity; the actual program applies the operation in the entire array of individuals at once.

[1]The factor of 0.01 represents a strong penalization based on the fitness range $[0, 1]$ and our preliminary analysis of fitness curves.

4.4 Q-NAS steps

We already defined the complete quantum individual, with the network and hyperparameters representation, including their unique observation and update procedures. In this section, we detail the Q-NAS steps.

In Figure 4.10, we provide the summarized steps of the algorithm, which are similar to the previously presented QIEA (Figure 3.2). Here, we will refer to the quantum population as $Q(t)$, which consists of a set of N quantum individuals $q_i^t, i = \{1, 2, .., N\}$.

```
1:  t ← 0
2:  Initialize Q(t)
3:  while t ≤ T do
4:      Generate classical population C(t) observing Q(t)
5:      if t = 0 then
6:          Evaluate C(t)
7:          P(t) ← C(t)
8:      else
9:          C(t) ← recombination between C(t) and P(t)
10:         Evaluate C(t)
11:         P(t) ← best individuals from [C(t) ∪ P(t)]
12:     end if
13:     Q(t + 1) ← update Q(t) based on P(t) values
14:     t ← t + 1
15: end while
```

Figure 4.10: Q-NAS algorithm. $Q(t)$ is the quantum population, $C(t)$ is the classical population, and $P(t)$ is the saved classical population.

First, we need to initialize $Q(t)$ (line 2 in Figure 4.10). The initialization procedure involves assigning initial probabilities to the quantum individuals. For the hyperparameters, the user specifies the ranges for each one, that is, the lower and upper limits. Then, to initialize this part of the quantum chromosome, we assign these values as the limits l_{ij} and u_{ij} of the PDFs. Additionally, for the network part of the chromosome, the user can provide initial probabilities for each function in the function set. However, if he/she does not provide these values, the program assigns the same probability to all functions, creating a uniform PMF. All nodes are initialized with the same PMF.

The loop of generations t starts with the observation of quantum individuals to generate candidate solutions (line 4). As discussed earlier, we observe each part of the quantum chromosome – numerical and structure – in well defined separate procedures. We then combine these elements in one single classical individual.

Therefore, the candidate solution comprises a network architecture description along with the hyperparameters' values.

It is important to mention that each quantum individual can generate one or more classical ones, as long as they generate the same number of individuals. Otherwise, we would favor some quantum individuals over the others, which could be questionable, as they cannot be directly evaluated [56]. Considering this possibility, the number of individuals in the classical population $C(t)$ is a multiple of N:

$$C(t) = \{\boldsymbol{c}_1^t, \boldsymbol{c}_2^t, ...\boldsymbol{c}_{m \cdot N}^t\}, \qquad m \in \mathbb{N} \tag{4-10}$$

The parameter *repetition* (m) specifies the number of classical individuals per quantum individuals that will be generated. If *repetition* $= 2$, for example, each quantum individual generates two classical ones.

Once we have the classical population $C(t)$, we evaluate each individual $\boldsymbol{c}_i^t$ independently (line 6 in Figure 4.10). Our evaluation comprises the following steps:

1. train the network for a relatively small number of epochs (50) using a subset of the training data and the evolved hyperparameters;

2. from epoch 45 onward, evaluate the network according to a predefined metric (*accuracy*, in our context) using a validation dataset, at the end of each epoch.

3. assign the best of the 5 evaluation results as the individual's fitness.

The idea of training the networks for a restricted number of epochs is present in several works [16, 20, 25, 26]. Nevertheless, the specific number differs, and the authors do not provide any discussion about it. We use the same number as Suganuma et al. [20], and we ran some experiments with 20 and 30 epochs of training. Our preliminary results indicated that 50 was a better choice between the tested values, but we did not explore this topic thoroughly. The same logic applies to the final accuracy evaluation: authors propose different methods with no clear justification. We calculate accuracy only at the final five epochs to save time, as it is expensive to stop training and perform evaluations. Also, we take the best out of five values because of common fluctuations, specially when training for few epochs. Finally, we remark that the network weights are initialized using the method proposed in [71], also applied by Suganuma et. al [20].

It is important to mention that we set a timeout for these training sessions. If the first 45 epochs of training (before accuracy evaluations start) take more than 90 minutes, the candidate network receives a fitness value of zero. We made this decision in order to make the algorithm exclude structures that take too long to train, thus creating pressure toward more efficient models. Note that the 90 minutes limit is

quite loose for the datasets used in this work, and it can be modified. In the majority of our experiments, the candidate networks do not reach this time limit.

After all the individuals were assigned a fitness value, we can rank them to save the best ones in $P(t)$. However, in generation $t = 0$, we do not have any previous population, so we store in $P(t)$ all the individuals we just evaluated (line 7 in Figure 4.10).

In the other generations ($t > 0$), we already have stored some classical individuals in $P(t)$, so we can apply recombination after generating $C(t)$ (line 9 in Figure 4.10). Notice that we only defined a crossover operator for the hyperparameters part of the chromosome, so recombination is carried out only for this part. We decided not to apply crossover operators in the network part, based on the following observation. Blocks of subsequent nodes can repeatedly appear during evolution, and new or unexpected sequences might be discovered. The application of crossover operators may introduce some noise and disturb the analysis of the occurrence of these blocks.

Unlike the first generation, since we already have $P(t)$ when we evaluate a new population $C(t)$, we must decide how to select individuals to be stored (line 11). There are several selection mechanisms available in the literature, such as [45]:

- Replace the k worst individuals from the old population with the k best from the new population (*steady-state*);
- Replace all the individuals from the old population, except for the best one (*elitism*);
- Replace all individuals from the old population (no steady-state or elitism).

We developed Q-NAS to use a steady-state technique, in which we select the best individuals from the old and new populations. Consider a population of K individuals. Every generation, we create $C(t)$ with size K, then we keep the K best individuals from $[C(t) \cup P(t)]$. Our studies in [72] analyzed the elitism selection, which did not show any improvement over this steady-state method. Furthermore, the variation that elitism promoted in the population only brought noise and no additional benefit. Therefore, we did not consider elitism in this work.

Finally, quantum individuals are updated based on the best classical individuals (line 13 in Figure 4.10). Similar to the observation process, this procedure is executed separately for each part of the chromosome. Following [30,56,61], we use the parameter *update_quantum_gen* to establish the frequency in which the update procedure will be conducted. More specifically, if *update_quantum_gen* $= 5$, the quantum update, described in sections 4.2 and 4.3, takes place every five generations, with a rate defined by *update_quantum_rate*. The update step completes the algorithm loop, which is repeated for T generations.

When the evolution is complete, we retrain the final architecture from scratch for 300 epochs using the optimized hyperparameters and all the available training data. We do not change the weight initialization method for the retraining phase. We evaluate the network every ten epochs using a validation dataset. The accuracies from the periodic evaluations are used to save the best model during the retraining phase. When training is over, the best validation model is applied to the test data so we can obtain the final accuracy value. The test accuracy is used to compare our models among different experiments and with other works.

We close this chapter with Table 4.1, which summarizes the Q-NAS' parameters and their meanings.

Table 4.1: Summary of Q-NAS' parameters and their description.

Parameter name	Description
crossover_rate	rate for the numerical crossover operator
max_generations	maximum number of generations to run the algorithm
max_num_nodes	maximum number of nodes in the network structure (network size)
num_quantum_individuals	number of quantum individuals in the quantum population
penalize_number	maximum number of reducing layers a network can have without suffering penalization
repetition	number of classical individuals each quantum individual will generate
update_quantum_gen	periodicity, in generations, in which the quantum individuals will be updated
update_quantum_rate	rate for all the quantum update operations
params_ranges	initial ranges of the hyperparameters to be optimized
fn_dict	dictionary defining the functions for the network search space and their initial probabilities

5
Experiments

This chapter presents the experiments we conducted, indicating how they influenced decisions about the Q-NAS algorithm and further investigation directions. First, we provide an overview of the experiments, describing their primary goals. The parameter settings, which are common to different tests, are also given. Then, in the subsequent sections, we detail each experiment, provide the results, and discuss the outcomes.

5.1
Experiments overview

In the previous chapter, we described the Q-NAS algorithm, considering the context of classification tasks. This chapter presents the experiments, which apply Q-NAS to different image classification datasets.

We considered three main goals when designing the experiments. The first one involves answering fundamental questions about Q-NAS, regarding its applicability and operation. Our second goal arises naturally: verify if Q-NAS can be successfully applied to a more challenging dataset. Finally, the third objective comprises a case study in which we apply Q-NAS to real datasets. We associate each goal with a set of experiments.

In the first group, we use the CIFAR-10 benchmark dataset, which contains $60\,000$ colored images of size 32×32 pixels, divided into training and test sets – $50\,000$ and $10\,000$ examples, respectively. The images are labeled for ten categories, such as *dog*, *cat*, or *airplane*. We argue that CIFAR-10 is an adequate choice to study Q-NAS and explore its parameters because the literature provides many results on this dataset. We present this first group of experiments from Section 5.2 to Section 5.8, in chronological order, precisely how their results guided our investigation.

The second set of experiments requires a more challenging dataset, so we selected CIFAR-100. It has the same properties as CIFAR-10, except for the number of classes that is ten times bigger. Thus, for the $50\,000$ training examples, CIFAR-100 has only 500 examples per class. These experiments are described in Section 5.9.

Lastly, for our case study in Section 5.10, we define a seismic image classification task and use Q-NAS to find a network that can solve it.

In addition to the main goals, we provide some configurations that are common to several experiments so the reader can have a general idea about the parameters. Notice, however, that we modify these configurations throughout the chapter, and we highlight the particular changes in the corresponding section.

Table 5.1 shows a parameter configuration shared among various experiments. Note that *penalize_number* $= 0$, which means that no penalization is applied.

Table 5.1: Parameter configuration of the Q-NAS algorithm.

parameter	value	parameter	value
crossover_rate	0.5	*num_quantum_ind*	5
max_generations	300	*repetition*	4
max_num_nodes	20	*update_quantum_gen*	5
penalize_number	0	*update_quantum_rate*	0.1

As discussed in Chapter 4, the user must determine the functions that will form the search space for all network nodes. Table 5.2 specifies a list of functions that we used in many experiments. In this case, we selected three types of functions, which we call: *ConvBlock*, *Pooling*, and *NoOp*. Following the work in [20], the *ConvBlock* comprises a convolutional layer, batch normalization, and ReLU activation. We also use zero-padding in the convolution layer input borders. The others are straightforward: the *Pooling* function can be a max-pooling or an average pooling layer; *NoOp* is the no-operation function that allows us to represent variable-length networks.

Table 5.2: Layer functions.

function name	function	kernel size	stride	filters	initial probability
conv_1_1_32	ConvBlock	1	1	32	0.042
conv_1_1_64	ConvBlock	1	1	64	0.042
conv_3_1_32	ConvBlock	3	1	32	0.042
conv_3_1_64	ConvBlock	3	1	64	0.042
conv_3_1_128	ConvBlock	3	1	128	0.042
conv_3_1_256	ConvBlock	3	1	256	0.042
conv_5_1_32	ConvBlock	5	1	32	0.042
conv_5_1_64	ConvBlock	5	1	64	0.042
max_pool_2_2	MaxPool	2	2	-	0.167
avg_pool_2_2	AvgPool	2	2	-	0.167
no_op	NoOp	-	-	-	0.333

In Chapter 4, we demonstrated that to define convolution and pooling functions completely, we need to specify some parameters, including kernel size and strides. Note that one can be as general as desired regarding the options of kernel,

strides, and number of filters. The options listed in Table 5.2 are somewhat biased toward simplicity and efficiency. In other words, this list favors function specifications that are relatively inexpensive concerning computational cost, e.g., *ConvBlocks* with a small number of filters. Furthermore, one can observe that the convolutional layers have a stride of 1, which means that they do not reduce the input size since they apply zero-padding.

Notice that Table 5.2 also includes the initial probability for each function. We equally divided the probabilities between the three types of functions: *NoOp* received $1/3$, and both *ConvBlock* and *Pooling* received $1/3$ divided by the number of options of each kind. This division guarantees that we are not favoring a specific type of function merely because it has more options available.

To complete the definitions, we need to set some training configurations for the Q-NAS evaluation step. First of all, in Section 4.4, we reported that only a subset of the training data is used. Some preliminary results with CIFAR-10 showed that a subset of $10\,000$ examples and a batch size of 256 lead to a good equilibrium between evolution time and final accuracy. The sampling procedure guarantees class balance: it randomly selects 900 examples per class for the training set (9000 in total) and 100 for the validation set (1000 in total).

We apply standard data augmentation and pre-processing techniques, also based on our preliminary results. As suggested in [20], the data augmentation comprises random crop after zero-padding the borders and random horizontal flipping. Mean subtraction (calculated over all the training data) is the only pre-processing technique applied. Finally, besides the BN layers included in our *ConvBlocks*, weight decay regularization is used in our training sessions. For the hardware configuration, the reader can refer to Appendix A.

In each of the following sections, we describe an experiment and discuss its outcomes. Due to resource constraints, we repeat runs only three or five times. Therefore, we do not compare them statistically, and we provide all outcomes without any summarization. Note that it is common in the literature to present NAS results of only a few runs [18, 20, 23, 25, 46, 73]. Moreover, in a single run, we evaluate about 6000 networks, and, hence the final accuracy is meaningful.

5.2 Optimizer and Hyperparameters

This first experiment aims to investigate the optimizer used for training the candidate networks and the possibility of evolving hyperparameters.

The SGD with momentum is a frequent choice of optimizer among NAS research [19, 20, 46, 74]. However, more sophisticated options are available, such as RMSProp [5] or Adam [75]. Currently, there is no consensus in the literature on

which one should be used; the choice strongly depends on the user [5]. We want to compare Q-NAS results when using the popular SGD with momentum and when applying a more elaborate optimizer. To accomplish this, we created two configurations: one for the SGD with momentum (*mom-1*) and one for the RMSProp (*rms-2*), using Tensorflow's default values for the optimizers' parameters (Table 5.3).

Additionally, we want to evaluate if some hyperparameters can be optimized concurrently with the network search. The idea is to compare Q-NAS under two circumstances: in the first one, Q-NAS only considers the structure – hyperparameters are fixed; in the other, it must look for both architecture and hyperparameters. We selected configuration *rms-2* to represent the first situation, as RMSProp has more parameters than the Momentum optimizer. Then we created setup *rms-3*, which uses RMSProp and evolves its parameters along with weight decay (see Table 5.3). Note that *rms-3* has wide search ranges that cover the values from *rms-2*.

Table 5.3: Hyperparameter configuration for the experiment.

config	decay	learning rate	momentum	weight decay
mom-1	–	1.0e-3	0.9	1.0e-4
rms-2	0.9	1.0e-3	0.0	1.0e-4
rms-3	[0.1, 0.999]	1.0e-3	[0.0, 0.999]	[1.0e-5, 1.0e-3]

We decided not to evolve the learning rate at this moment because it is a critical hyperparameter that, as observed in preliminary experiments, can affect the results and disturb the investigation itself. Furthermore, RMSProp has an adaptive learning rate scheme that adjusts the learning rate to a certain degree. We thus fixed the learning rate as $1.0 \cdot 10^{-3}$.

We repeated five runs of Q-NAS on CIFAR-10 for each optimizer setup. The other parameters and the function list are defined in Table 5.1 and Table 5.2, respectively. Table 5.4 presents the results, including the final evolution accuracy (best *fitness*) and the number of layers in the winner network. Recall that, as reported in Section 4.4, we retrain the best network using the complete dataset – $45\,000$ examples for training and $5\,000$ for validation, in CIFAR-10. Table 5.4 also shows the accuracies for the retraining phase, for both the validation and test sets. We highlight the best test result only for clarity; no decisions are made based on this outcome.

The runs with the momentum optimizer led to worse test accuracy than the ones with RMSProp and also to smaller networks. The momentum optimizer might need more than 50 epochs to achieve lower losses for deeper networks, increasing the pressure for smaller architectures that usually train faster. This difference suggests that using the RMSProp optimizer can offer performance advantages over the SGD.

Configurations *rms-2* and *rms-3* produced similar results, although *rms-2* generated slightly better architectures and also the best network with 89.84% of

Table 5.4: Results for each optimizer configuration.

#	config	total time	# of layers	accuracy		
				fitness	retrain validation	retrain test
1	mom-1	70 h	8	0.6940	0.8664	0.8611
2	mom-1	64 h	11	0.7000	0.8460	0.8405
3	mom-1	68 h	11	0.6940	0.8520	0.8511
4	mom-1	67 h	9	0.6950	0.8548	0.8502
5	mom-1	67 h	12	0.6830	0.8642	0.8539
1	rms-2	72 h	10	0.7890	0.8958	0.8877
2	rms-2	71 h	12	0.7780	0.8892	0.8746
3	rms-2	68 h	11	0.7810	0.8960	0.8895
4	rms-2	75 h	11	0.7900	0.8900	0.8790
5	rms-2	72 h	10	0.7820	0.9006	**0.8984**
1	rms-3	67 h	16	0.7740	0.8902	0.8800
2	rms-3	64 h	12	0.7790	0.8712	0.8679
3	rms-3	73 h	11	0.7710	0.8686	0.8553
4	rms-3	70 h	11	0.7700	0.8832	0.8704
5	rms-3	71 h	12	0.7630	0.8700	0.8693

test accuracy (*rms-2 #5*). Although this seems to be a moderate performance, we should emphasize that we restricted the maximum network size to 20 layers, so all of the tested networks are small. Our best architecture has only ten layers, which is considerably smaller than most state-of-the-art networks. Furthermore, this accuracy is already comparable to the results of the hand-designed model Maxout [54] (90.70%). These outcomes indicate that Q-NAS can extract the capabilities of simple layers and small networks to a great extent.

Figure 5.1 depicts the best (decoded) architectures for configurations *rms-2* and *rms-3*. In both structures, typical sequences, such as two convolutional layers followed by a pooling layer, are present (see the brackets in Figure 5.1). However, the *rms-3* best architecture also contains convolutions of different kernel sizes interleaved, which contrasts with conventional hand-designed networks.

Figure 5.2 shows how the network part of a quantum individual evolved in *rms-2 #5*. One can note some correspondence between the final structure and the quantum individual. For example, in node 0, we have *conv_3_1_256* with high probability, which is the function of layer 0 on the left side of Figure 5.1. On the other hand, as the quantum individual represents probabilities, the classical individuals it generates can be entirely different. These probabilities never go to zero, so they always allow the appearance of less favored functions.

Nodes	Function names
0.	conv_3_1_256
1.	no_op
2.	conv_3_1_128
3.	no_op
4.	no_op
5.	no_op
6.	conv_3_1_64
7.	conv_3_1_64
8.	no_op
9.	conv_3_1_256
10.	conv_3_1_256
11.	max_pool_2_2
12.	conv_3_1_128
13.	no_op
14.	no_op
15.	no_op
16.	no_op
17.	no_op
18.	max_pool_2_2
19.	conv_3_1_128

rms-2 #5

Nodes	Function names
0.	no_op
1.	conv_1_1_32
2.	no_op
3.	conv_3_1_64
4.	conv_5_1_64
5.	conv_3_1_128
6.	conv_5_1_64
7.	max_pool_2_2
8.	conv_3_1_128
9.	conv_3_1_128
10.	avg_pool_2_2
11.	conv_3_1_128
12.	no_op
13.	conv_3_1_128
14.	max_pool_2_2
15.	conv_5_1_64
16.	avg_pool_2_2
17.	conv_3_1_32
18.	no_op
19.	conv_1_1_64

rms-3 #1

Figure 5.1: Final structures for runs *rms-2 #5* and *rms-3 #1*.

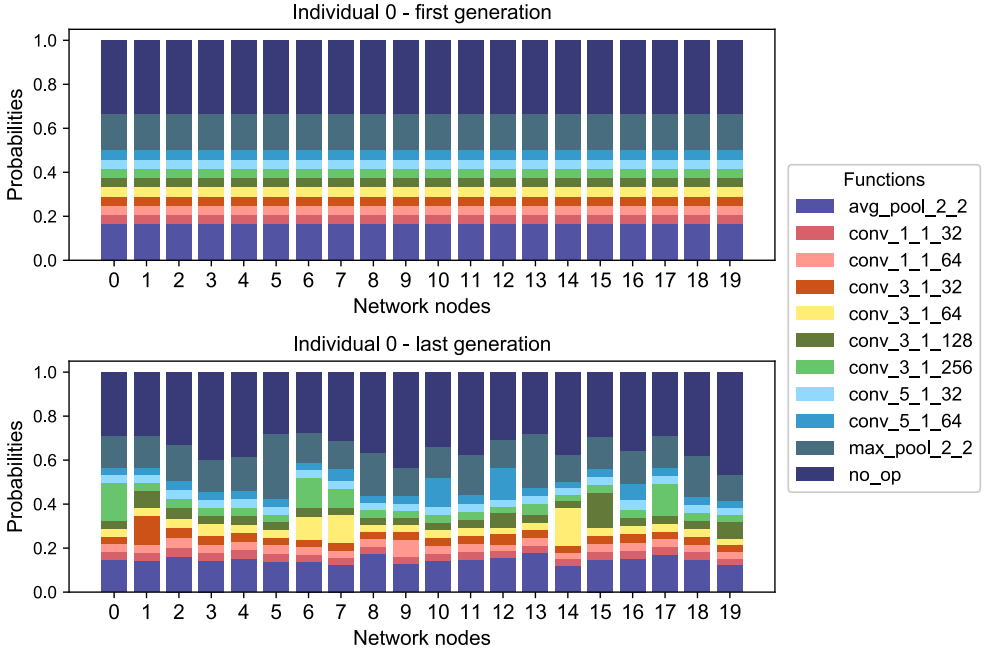

Figure 5.2: Quantum individual 0 for run *rms-2 #5*.

A final and important observation is that Q-NAS was able to find networks with 88% of test accuracy even in the more challenging scenario of configuration *rms-3*. The numerical part of a quantum individual in its initial and final states for run *rms-3 #1* is shown in Figure 5.3. For all hyperparameters, the pulses did not decrease substantially, which indicates that they are not much influential to the results, and a relatively wide range of values is acceptable. Indeed, the default Tensorflow values, represented in *rms-2*, were sufficient to produce similar and even better results.

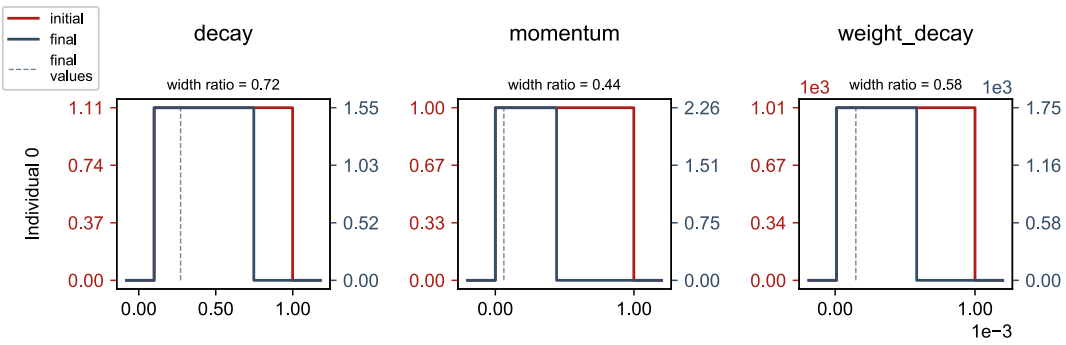

Figure 5.3: Numerical quantum individual 0 for *rms-3 #1* in its initial and final states. The width ratio between initial and final pulses are also given.

5.3 Analysis of parameters

In this section, we analyze the impact of the Q-NAS' parameters on the network search, using the CIFAR-10 dataset as the classification task. We designed experiments varying one parameter at a time to study its influence on the evolutionary process. Ideally, for a relatively wide range of parameter values, we would expect Q-NAS to find satisfactory architectures. We include here the study of the following parameters (see Table 4.1): initial function probabilities, *crossover_rate*, *update_quantum_gen*, *update_quantum_rate*, *repetition*, and *num_quantum_individuals*. We explore the remaining parameters in other sections.

5.3.1 Initial function probabilities

We begin this set of experiments by exploring the initial function probabilities. The idea is to create a scenario in which the user would try to bias the search with more efficient functions and compare it with the unbiased case. For example, the user could remove convolutions with larger kernel sizes from the function list and increase the probabilities of convolutions with fewer filters. Table 5.5 shows the function list we used to represent this scenario: there are no convolutions with 5×5 kernels and the initial probabilities of *conv_3_1_32* and *conv_3_1_64* are bigger. Table 5.5 also shows the probabilities for the unbiased case.

We used the values in Table 5.1 for the other parameters in both cases. Also, we selected setup *rms-3* in Table 5.3 for the optimizer and the hyperparameters' ranges.

Q-NAS was executed five times for each scenario, and the results are presented in Table 5.6. In terms of test accuracy, both configurations were able to achieve 87%, but the unbiased case was slightly better, reaching more than 86% in all runs. On the other hand, the biased case presented faster runtimes, which can be explained by the emphasis given to convolutions with fewer filters to train. The results for the

Table 5.5: Layer functions and their initial probabilities as divisions, e.g., $1/3/8 = 0.042$.

function name	function	kernel size	stride	filters	initial probability	
					unbiased	biased
conv_1_1_32	ConvBlock	1	1	32	$1/3/6$	$1/3/8$
conv_1_1_64	ConvBlock	1	1	64	$1/3/6$	$1/3/8$
conv_3_1_32	ConvBlock	3	1	32	$1/3/6$	$\mathbf{2}/3/8$
conv_3_1_64	ConvBlock	3	1	64	$1/3/6$	$\mathbf{2}/3/8$
conv_3_1_128	ConvBlock	3	1	128	$1/3/6$	$1/3/8$
conv_3_1_256	ConvBlock	3	1	256	$1/3/6$	$1/3/8$
max_pool_2_2	MaxPool	2	2	-	$1/3/2$	$1/3/2$
avg_pool_2_2	AvgPool	2	2	-	$1/3/2$	$1/3/2$
no_op	NoOp	-	-	-	$1/3$	$1/3$

current experiment are worse than the previous ones in Table 5.4, indicating that the inclusion of 5×5 convolutions in the function list was beneficial to the search.

Table 5.6: Results for each configuration.

#	bias	total time	# of layers	accuracy		
				fitness	retrain validation	retrain test
1	no	69 h	11	0.7720	0.8770	**0.8779**
2	no	70 h	12	0.7730	0.8656	0.8605
3	no	67 h	11	0.7790	0.8632	0.8669
4	no	69 h	10	0.7890	0.8708	0.8626
5	no	70 h	16	0.7770	0.8756	0.8661
1	yes	65 h	13	0.7870	0.8788	0.8704
2	yes	62 h	13	0.7710	0.8530	0.8511
3	yes	62 h	14	0.7710	0.8604	0.8548
4	yes	63 h	9	0.7750	0.8730	0.8667
5	yes	62 h	9	0.7690	0.8560	0.8516

Figure 5.4 shows the relative frequency of occurrence of each function during evolution, for the best runs in each scenario. Take function *conv_3_1_256* in the left graph, for example: considering all network nodes throughout evolution, it appeared around 10% of the nodes. In the biased case, the relative frequencies of the preferred functions are higher, showing that the initial bias influenced the search, as expected. Furthermore, one can observe that convolutions with more filters appear more frequently in the unbiased scenario.

In the following subsections, we investigate parameters directly related to the evolutionary process, such as the crossover rate or the number of individuals in each generation. The biased setup will be our baseline for all experiments in this section,

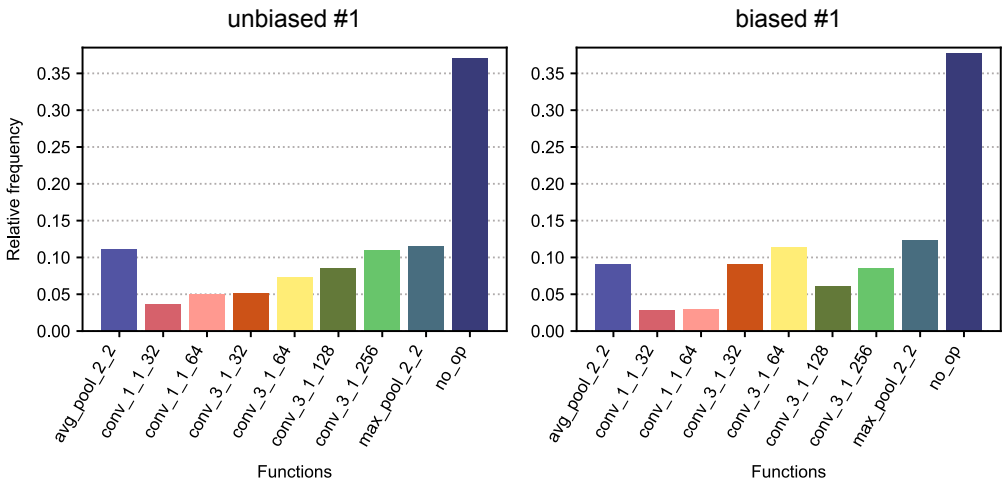

Figure 5.4: Relative frequency of occurrence of each function during evolution, for the best runs of biased and unbiased cases.

as its function list promoted shorter runtimes. The function list of Table 5.5 will be left unchanged, as well as the optimizer settings (*rms-3* in Table 5.3). The values included in Table 5.1 will be the control configuration, for which we will vary one parameter at a time in the next experiments.

5.3.2 Crossover rate

The second parameter of Q-NAS to be analyzed is the crossover rate. In the baseline configuration, it is set to 0.5. We ran Q-NAS three times for two additional crossover rates: 0.1 and 0.9. Table 5.7 shows these results along with the three best ones from the biased scenario of Table 5.6, repeated here for clarity.

The changes in the crossover rate directly influence the hyperparameters' evolution. However, as seen in Section 5.2, these specific hyperparameters did not affect the results in a significant way. Therefore, even if we dramatically change the course of evolution of these hyperparameters, we should not expect considerable changes in the final results. All configurations in Table 5.7 reached accuracies above 87%, confirming this idea.

Figure 5.5 shows quantum genes representing the *decay* parameter, for the best runs in Table 5.7. Observing the pulses, one can see that the crossover rate influences the final width of the quantum genes. Higher crossover values seem to stimulate the pulses to get thinner. However, this parameter alone is not sufficient to determine the behavior of this quantum gene, as the final pulse for a crossover rate of 0.9 is wider than the one for the 0.5 case. The genes encoding the other hyperparameters showed similar responses to each crossover rate value.

Table 5.7: Results for each crossover value.

#	crossover rate	total time	# of layers	accuracy		
				fitness	retrain validation	retrain test
1	0.5	65 h	13	0.7870	0.8788	0.8704
3	0.5	62 h	14	0.7710	0.8604	0.8548
4	0.5	63 h	9	0.7750	0.8730	0.8667
1	0.1	62 h	14	0.7670	0.8862	**0.8813**
2	0.1	64 h	14	0.7690	0.8628	0.8607
3	0.1	61 h	13	0.7760	0.8468	0.8383
1	0.9	62 h	15	0.7730	0.8788	0.8740
2	0.9	61 h	12	0.7680	0.8642	0.8621
3	0.9	61 h	10	0.7680	0.8708	0.8597

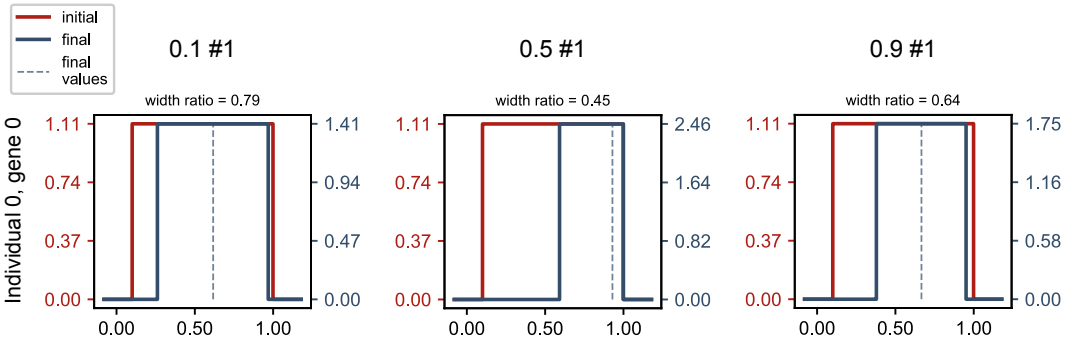

Figure 5.5: Genes representing the decay hyperparameter for the quantum individual 0 of the best runs in Table 5.7.

5.3.3 Frequency of quantum updates

The *update_quantum_gen* parameter determines the periodicity (in generations) of the quantum updates. Unlike the *crossover_rate*, this parameter directly affects the complete quantum individual. We consider the baseline value of 5 a conservative choice, that is, a relatively low update frequency. We explore here the values of 3 and 1: updates occurring every three generations and every generation, respectively. Table 5.8 shows the results.

One item that stands out in Table 5.8 is the evolution time: it increases significantly with the frequency of updates (lower values of *update_quantum_gen*). Despite the runtime contrast, the test accuracy is comparable in all configurations.

We remark that the update procedure, by itself, is not responsible for the runtime increase. Executing only the update procedure for 300 iterations demanded 2.43 seconds of an i7 processor. We attribute this increase to the saturation of quantum individuals, as detailed below.

Table 5.8: Results for each *update_quantum_gen* (update generations) value.

#	update generations	total time	# of layers	accuracy		
				fitness	retrain validation	retrain test
1	5	65 h	13	0.7870	0.8788	0.8704
3	5	62 h	14	0.7710	0.8604	0.8548
4	5	63 h	9	0.7750	0.8730	0.8667
1	3	63 h	14	0.7760	0.8540	0.8524
2	3	65 h	13	0.7740	0.8766	0.8677
3	3	69 h	12	0.7750	0.8514	0.8449
1	1	80 h	13	0.7860	0.8714	0.8694
2	1	81 h	11	0.7810	0.8890	**0.8781**
3	1	91 h	10	0.7820	0.8732	0.8645

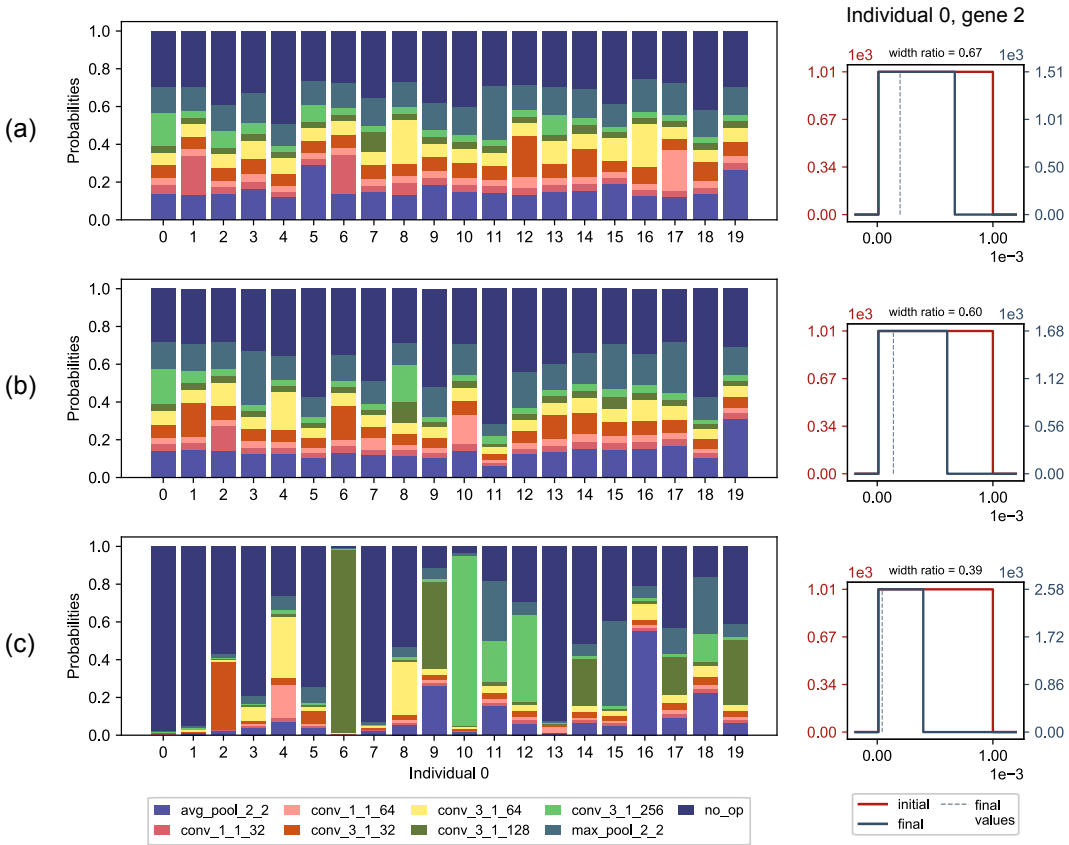

Figure 5.6: Network quantum individuals and genes representing the weight decay hyperparameter for the best run of each *update_quantum_gen* value. The pairs (value, run number) are: (a) 5 #1, (b) 3 #2, and (c) 1 #2.

Figure 5.6 shows the network part of quantum individuals, as well as quantum genes representing the weight decay parameter, for the best runs of each value of *update_quantum_gen* in Table 5.8. Observe that the network individual for

update_quantum_gen $= 1$ is significantly more saturated than the others. This individual also shows many nodes saturated with *conv_3_1_128* and *conv_3_1_256*, that are slower to train compared to the other options. Therefore, the classical population will have more networks with these functions, increasing the total runtime.

The final width of the pulses is also affected by the *update_quantum_gen* parameter. However, the effect seems to be weaker than in the network part of the chromosome.

5.3.4 Update rate

The *update_quantum_rate* controls the number of quantum genes that will be modified at each update: higher rates lead to more genes being adjusted at a time. Again, the baseline value of 0.1 is a conservative choice, and we investigate higher rates (0.5 and 0.9, specifically). The results for the runs with higher rates, along with the baseline, are listed in Table 5.9.

Table 5.9: Results for each *update_quantum_rate* value.

#	update quantum rate	total time	# of layers	accuracy		
				fitness	retrain validation	retrain test
1	0.1	65 h	13	0.7870	0.8788	0.8704
3	0.1	62 h	14	0.7710	0.8604	0.8548
4	0.1	63 h	9	0.7750	0.8730	0.8667
1	0.5	76 h	14	0.7790	0.8612	0.8635
2	0.5	78 h	13	0.7850	0.8826	0.8779
3	0.5	86 h	13	0.7940	0.8958	0.8853
1	0.9	104 h	10	0.7930	0.8756	0.8775
2	0.9	87 h	10	0.7930	0.8858	0.8777
3	0.9	88 h	14	0.7900	0.8956	**0.8930**

The runtime seems to increase with the update rate, a similar effect to the update frequency. On the other hand, the runs for higher rates presented higher fitness values and also reached better final accuracies.

In Figure 5.7, we can see quantum individuals for the best runs of 0.5 and 0.9 rates. The network part of the chromosomes shows intense saturation, even stronger than the ones in Figure 5.6. As in the last experiment, we can associate the runtime increase to this saturation. The influence on the numerical quantum genes is also noticeable and stronger than in the cases of Figure 5.6.

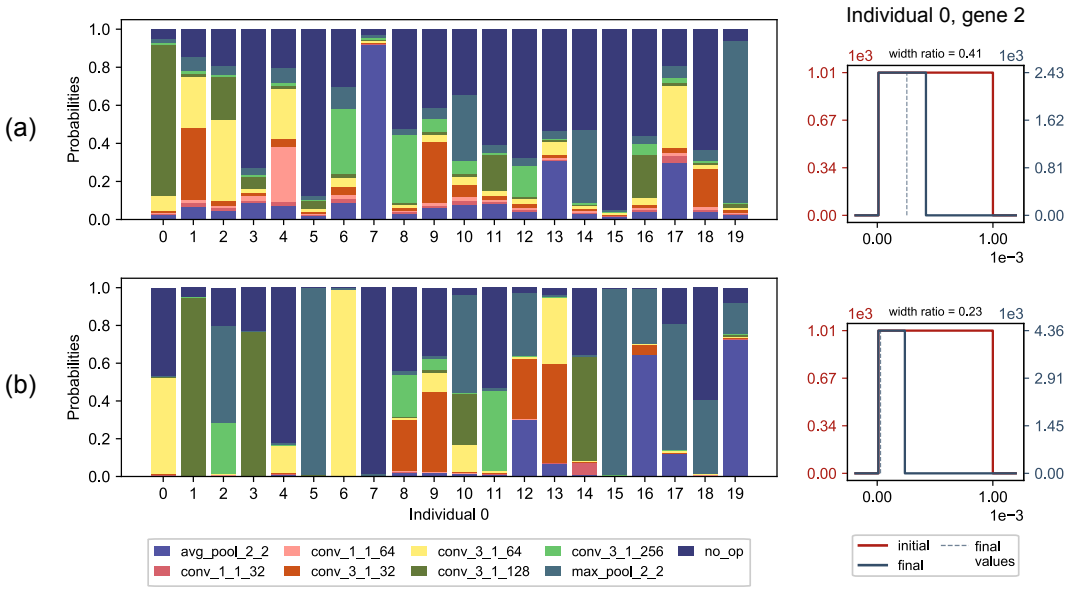

Figure 5.7: Network quantum individuals and genes representing the weight decay hyper-parameter for runs (a) 0.5 #3 and (b) 0.9 #3.

Although higher rates produced better results than the conservative baseline, strong saturation is not beneficial to evolution. This contrast suggests that further experiments are needed to select a combination of parameters that improves the balance between performance, runtime, and saturation. However, we highlight that, even without this enhancement, for the broad range of rate values, Q-NAS generated networks with more than 87% of accuracy.

5.3.5 Number of individuals

Two parameters determine the number of classical individuals: *repetition* and *num_quantum_individuals*. Here we explore configurations varying each one of them separately and also altering both simultaneously. In the baseline configuration, we have five quantum individuals, and repetition is 4, making 20 classical individuals. Fixing the number of quantum individuals as 5, we added the repetition values of 1 and 2. Keeping the repetition equal to 4, the additional numbers of quantum individuals are 1 and 3. Finally, we explored the extremes: one quantum individual with a repetition factor of 20, and 20 individuals for a repetition of 1. We did not use setups that create more than 20 classical individuals, restricting the computational resources for the benefit of efficiency. The results for the runs with all of these configurations are listed in Table 5.10.

The executions with setups that generate fewer classical individuals (4 and 5) presented shorter evolution times. A limited number of candidate networks at every generation can reduce the chance to evaluate several slow networks. The results are

Table 5.10: Results for each *num_quantum_individuals* (# of q-ind) and *repetition* (rep) configuration.

#	# of q-ind.	rep.	# of class. ind.	total time	# of layers	accuracy		
						fitness	retrain validation	retrain test
1	5	4	20	65 h	13	0.7870	0.8788	0.8704
3	5	4	20	62 h	14	0.7710	0.8604	0.8548
4	5	4	20	63 h	9	0.7750	0.8730	0.8667
1	5	1	5	40 h	12	0.7650	0.8330	0.8205
2	5	1	5	39 h	13	0.7820	0.8810	0.8752
3	5	1	5	41 h	12	0.7790	0.8998	**0.8982**
1	5	2	10	48 h	13	0.7740	0.8624	0.8595
2	5	2	10	51 h	15	0.7650	0.8840	0.8822
3	5	2	10	55 h	13	0.7750	0.8700	0.8654
1	1	4	4	38 h	14	0.7580	0.8480	0.8431
2	1	4	4	37 h	9	0.7640	0.8590	0.8384
3	1	4	4	43 h	14	0.7620	0.8656	0.8662
1	3	4	12	53 h	12	0.7830	0.8826	0.8762
2	3	4	12	54 h	13	0.7630	0.8144	0.8081
3	3	4	12	57 h	12	0.7680	0.8542	0.8522
1	1	20	20	62 h	12	0.7840	0.8688	0.8687
2	1	20	20	61 h	13	0.7820	0.8726	0.8686
3	1	20	20	59 h	11	0.7960	0.8662	0.8582
1	20	1	20	64 h	14	0.7800	0.8816	0.8772
2	20	1	20	60 h	11	0.7730	0.8644	0.8565
3	20	1	20	63 h	12	0.7820	0.8662	0.8528

comparable regarding the final accuracy, with some exceptions (runs for 4 and 12 classical individuals). It is expected that more classical individuals should increase the chance of finding better structures. Nevertheless, the results in Table 5.9 are not sufficient to confirm this idea, as one can see good accuracies for 5, 10, and 20 classical individuals. For example, compare the following runs in Table 5.9, defined as (quantum individuals, repetition, run number): (5, 1, #3), (5, 2, #2), (20, 1, #1).

Looking at *num_quantum_individuals* and *repetition* separately, one can see that the runs with five quantum individuals were generally better than the others. All quantum individuals begin with the same probability distribution, so they all represent the search space equally at first. If one quantum individual becomes saturated during evolution, the others can still maintain the variability of the search. In the extreme case of 20 quantum individuals and repetition of 1, though, we observed that

some quantum individuals do not change much during evolution, indicating that this combination is not ideal.

The results throughout this section are promising regarding the robustness of Q-NAS. We varied several parameters and observed that some of them affect more the evolutionary process than others, but the results were similar in most of the runs. Even though a parameter tuning study can help improve the results, many setups were able to produce satisfactory outcomes.

5.4 Sampling

In the last section, we studied the impact of several parameters on the evolutionary process. Another essential element of investigation is the influence of the dataset sample. In all experiments so far, the evaluation procedure trained candidate networks using the same subset of $10\,000$ examples from the $50\,000$ training images of CIFAR-10. Although the winner networks are retrained with the complete dataset, we want to verify if using different samples in the evolution can affect the final results for CIFAR-10. We thus propose to run Q-NAS with a fixed parameter configuration for three different dataset samples and compare the results.

We selected the *rms-2* runs from Section 5.2 as our baseline results (*sample 1*), because the best network so far was generated by one of these runs. Two additional samples were created with the same sampling scheme (samples *2* and *3*). We ran Q-NAS five times for each new sample using the parameter settings of the *rms-2* runs. Table 5.11. shows the results; the *rms-2* runs are repeated here for clarity.

For all samples, Q-NAS was able to find networks reaching more than 89% of test accuracy, with a new best value of 90.09% for run *sample-2 #5*. Also, the variation between runs of each sample is similar: the difference between maximum and minimum accuracies is about 2%, and *sample 1* is slightly worse than the others. These observations indicate that the influence of the sample is small for the CIFAR-10 dataset. In this case, we still have a reasonable number of examples of each class in the subset, so using a reduced dataset in evolution is an adequate strategy to improve efficiency. However, when the number of examples per class is already small, sampling might affect the results and needs to be carefully investigated.

Another interesting observation is the difference in the values for the accuracy during evolution (fitness): for *sample 1*, they are smaller than the others. One possible explanation is the small validation set in the sampled datasets. The fitness value is the best validation accuracy of the candidate network. As the validation set contains only 100 examples per class, different samples can contain a better or worse representation of the dataset. One could increase the validation set to reduce this difference, in detriment of efficiency, as evaluation would take longer to conclude.

Table 5.11: Results for each sample.

#	sample	total time	# of layers	accuracy		
				fitness	retrain validation	retrain test
1	1	72 h	10	0.7890	0.8958	0.8877
2	1	71 h	12	0.7780	0.8892	0.8746
3	1	68 h	11	0.7810	0.8960	0.8895
4	1	75 h	11	0.7900	0.8900	0.8790
5	1	72 h	10	0.7820	0.9006	0.8984
1	2	70 h	13	0.8090	0.8958	0.8858
2	2	70 h	10	0.8070	0.8954	0.8993
3	2	69 h	14	0.8000	0.8838	0.8820
4	2	72 h	12	0.8040	0.8940	0.8905
5	2	71 h	12	0.8110	0.8990	**0.9009**
1	3	70 h	12	0.7870	0.8956	0.8938
2	3	67 h	10	0.8010	0.8842	0.8819
3	3	69 h	11	0.8000	0.8982	0.8878
4	3	70 h	13	0.8020	0.9024	0.8963
5	3	72 h	13	0.7910	0.8914	0.8817

However, as the final test accuracy seems to be little affected, we do not consider increasing the validation set in this work.

5.5 Retrain analysis

The discussions so far have focused on evolution, but it is also necessary to examine the retraining phase. As detailed in Section 4.4, the final retraining procedure differs from the training scheme of the evolution in two points: the dataset and the number of epochs. We retrain the final network using the complete dataset for more epochs; the other settings are unchanged, or, if applicable, we use the evolved values.

On the other hand, as pointed by Assunção et al. [32], several authors use special schemes for the final retraining phase: different optimizers, learning rate schedules, additional regularization methods, among others [16, 18, 20, 53, 76]. We selected two such schemes from the NAS literature to investigate further and compare them to our method. More specifically, we retrained some of our final networks using the new schemes to analyze the behavior of each training procedure.

The first scheme [23] uses SGD with a momentum of 0.9 for 300 epochs and *cosine decay* for the learning rate. The initial learning rate is set to 0.1, and it decays

as a half period cosine function (see Figure 5.8). We will refer to this approach as *cosine scheme*.

The second one [20], which we call *special scheme*, also uses SGD with a momentum of 0.9, but for 500 epochs, and applies a unique learning rate schedule. The learning rate values in this schedule are modified in specific epochs. The values for starting epoch and learning rate are $[(0; 0.01), (5; 0.1), (250; 0.01), (375; 0.001)]$ (see Figure 5.8). It is worth mentioning that this schedule was developed and tested for the CIFAR-10 dataset, so it might not be an adequate option for other datasets.

We selected the networks from the runs for *sample 2*, in Section 5.4 to retrain using the schemes above. We repeat the results for our retrain method (column *evolution scheme*) and add the new ones in Table 5.12.

Table 5.12: Test accuracies for different retraining schemes.

#	evolution scheme	cosine scheme	special scheme
1	0.8858	0.9221	0.9245
2	0.8993	0.9138	0.9266
3	0.8820	0.9208	0.9264
4	0.8905	0.9197	0.9216
5	0.9009	0.9233	**0.9274**

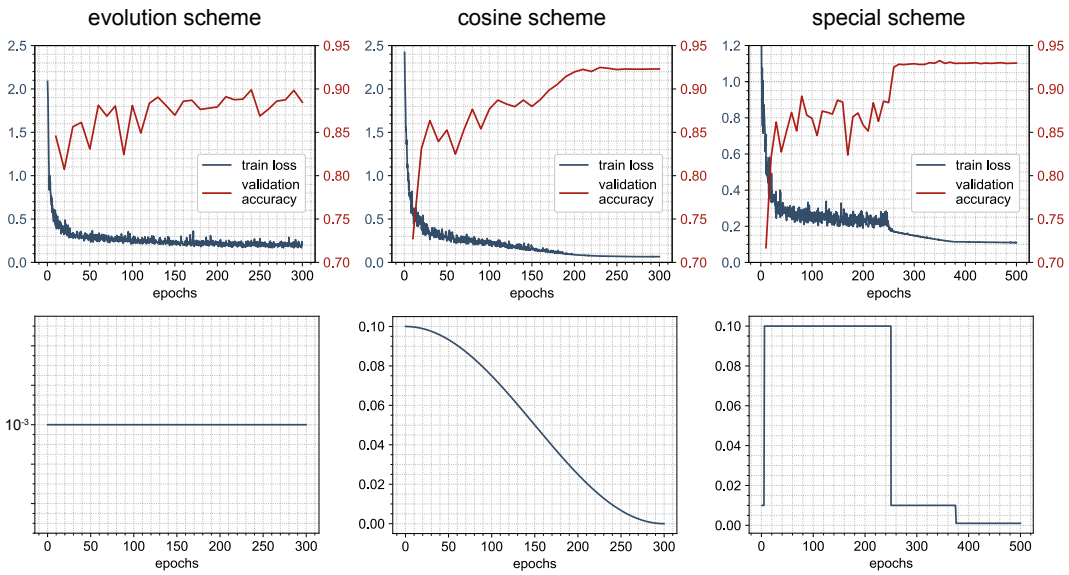

Figure 5.8: Retraining schemes. Bottom: the learning rate as a function of training epochs for each scheme. Top: the training loss and validation accuracy for the best network (run #5).

For all networks in Table 5.12, it is possible to see a significant improvement in the test accuracy when using the new schemes. The training loss and validation accuracy curves in Figure 5.8 helps explain this behavior. In our scheme, we used the RMSProp optimizer with a fixed learning rate of 0.001, which is the same

configuration applied during evolution for *sample-2* runs. The validation accuracy in the early stages of training (less than 70 epochs in Figure 5.8) is higher for our scheme. However, it increases only by a small amount from this point forward. The other schemes benefit from the decrease in the learning rate and show a significant accuracy improvement at the end of the training session.

The *special scheme* presented the best results for all networks, but compared to the *cosine scheme*, the difference is negligible, especially if we consider that the *cosine scheme* runs for 300 epochs and the *special scheme* for 500. Furthermore, the cosine decay is a more generic learning rate schedule; it was not specially designed for a specific dataset, and yet provided remarkable results.

When we compared the optimizers RMSProp and SGD with momentum in Section 5.2, we concluded that RMSProp was a better choice. Specifically, the RMSProp executions led to higher accuracies when using the same optimizer and learning rate for evolution and retraining phases (runs *mom-1* and *rms-2* in Table 5.4). On the other hand, the optimizer SGD with momentum, in conjunction with the cosine scheme, improved the retraining results.

Considering all the points discussed here, we decided to keep using RMSProp for the evolution phase, with fixed hyperparameters, and the *cosine scheme* for the retrain phase from now on. Although our experiments in sections 5.2 and 5.3 showed that it is possible to evolve numerical hyperparameters with Q-NAS, we did not see a clear advantage of evolving the RMSProp parameters in the CIFAR-10 task. Additionally, for short periods of training with a fixed learning rate, as occurs during the evolutionary process, the RMSProp reaches higher accuracies.

Regarding the retraining phase, the *cosine scheme* only requires the definition of an initial learning rate, as opposed to a specially tuned learning rate schedule. Therefore, we consider that using the *cosine scheme* for the retraining phase has the same level of automation as applying a selected optimizer with a fixed learning rate.

5.6 Penalization

In Section 4.3, we described the selected method to address the problem of invalid structures: a penalization scheme along with a correction procedure. The structures are always corrected if necessary, even in the absence of penalization. We did not apply penalization ($penalize\_number = 0$) in the previous experiments, and in this section, we investigate this feature.

The problem concerning structures with an excessive number of reducing layers can be aggravated with increasing network sizes. As the nodes are sampled independently in Q-NAS, a bigger network has a higher chance of including more reducing layers. So far, we set the maximum number of nodes to 20, and we propose

to study the penalization effect also on deeper networks (*max_num_nodes* $= 30$).

We selected the *sample-2* runs as our control results (absence of penalization). Thus, we adopted the configuration of these runs, including the dataset sample, for the new experiments. The CIFAR-10 dataset has images of 32×32 pixels, so the maximum number of pooling layers allowed is five. Therefore, we decided to penalize networks with more than three layers of reducing functions (only *Pool* in our function list of Table 5.2).

One can argue that the presented penalization method adds a new parameter to Q-NAS and that we could use, instead, an intrinsic penalization scheme: initial probability bias. To investigate this possibility, we created an additional configuration where we divided the initial probabilities in the following way: *NoOp* receives $1/3$, *ConvBlock* receives $3/6$, and *Pool*, $1/6$. The pooling functions begin with three times less probability than the convolutional functions.

In summary, we will run Q-NAS three times for each new configuration: two for *penalize_number* $= 3$ (maximum network size of 20 and 30) and two for the initial bias scheme (maximum sizes of 20 and 30). Table 5.13 lists these results and the baseline; it also shows the number of pooling layers in the winner networks.

Table 5.13: Results for each configuration defined in the second column.

#	penalize? -bias? -nodes	total time	# of layers	# of pool layers	accuracy		
					fitness	retrain validation	retrain test
1	no-no-20	70 h	13	5	0.8090	0.9262	0.9221
2	no-no-20	70 h	10	2	0.8070	0.9172	0.9138
3	no-no-20	69 h	14	4	0.8000	0.9244	0.9208
4	no-no-20	72 h	12	4	0.8040	0.9258	0.9197
5	no-no-20	71 h	12	3	0.8110	0.9248	0.9233
1	yes-no-20	65 h	11	2	0.8000	0.9024	0.8970
2	yes-no-20	69 h	14	4	0.8030	0.9308	0.9296
3	yes-no-20	70 h	11	2	0.7940	0.9150	0.9183
1	yes-no-30	78 h	17	5	0.7770	0.9254	0.9306
2	yes-no-30	73 h	17	5	0.7790	0.9266	0.9136
3	yes-no-30	78 h	14	3	0.7760	0.9178	0.9154
1	no-yes-20	131 h	14	3	0.8040	0.9196	0.9147
2	no-yes-20	132 h	17	4	0.8040	0.9240	0.9250
3	no-yes-20	134 h	11	2	0.8150	0.8852	0.8792
1	no-yes-30	149 h	20	5	0.8030	0.9260	0.9246
2	no-yes-30	150 h	18	3	0.8080	0.9306	0.9275
3	no-yes-30	151 h	19	5	0.8010	0.9300	**0.9311**

Regarding the test accuracies, the penalized runs reached the same levels as the non-penalized control runs. For the configuration *yes-no-30*, a network of size 17 achieved an accuracy of 93.06%. Notice that, for this setup, the number of pooling layers in the final structure is higher than in the 20 nodes runs, as expected. This result can indicate that for even bigger networks, the penalization scheme might not be sufficient to maintain the number of pooling layers inside the allowed limit.

When comparing the various settings, the most evident difference between penalization and bias schemes is the evolution time: for the latter, it is two times higher. The initial bias favors the occurrence of more convolutional layers in each network, which can increase the training time and, consequently, the evolution time. The accuracies, however, are similar to those from the penalization runs, with a new best value of 93.11%. We consider the penalization scheme the best choice because it is a more general method, and it did not increase the runtime.

In Table 5.13, we highlight run *#1*, for the bias scheme with 30 nodes, in which the final network description had more pooling layers than the maximum allowed. The number of pooling layers from the actual corrected network is different from the decoded individual description. This is an example of unwanted behavior, which the penalization scheme should help minimize. However, it cannot guarantee it: when the best network appears early in evolution, the penalization cannot help, as it will only affect the subsequent structures. In our experiments, this situation was rare even when no penalization was applied.

To further investigate this issue, we analyzed the number of pooling layers in each network generated in the evolutionary process. The top graphs on Figure 5.9 show the relative frequency of pooling layers in all generated networks, for the best penalization and baseline runs. Consider, for example, the top graph for the baseline run, indicated by *no-no-20*. It shows that about 12% of the generated networks had four pooling layers. We extended the same analysis to the saved population, which contains the best individuals that guide the evolution (see the bottom of Figure 5.9).

The top graphs of Figure 5.9 show that several invalid networks are generated (more than five pooling layers), especially in the 30-nodes configuration. However, the majority of the networks in the winning population are valid, indicating that invalid networks have little influence on the final results.

Comparing graphs *no-no-20* and *yes-no-20* in Figure 5.9, we highlight some observations. Regarding the generated population, it seems to be slightly affected by the penalization, as the distribution and average are similar. The effect of penalization becomes evident in the winning population, as we notice a considerable decrease in the average number of pooling layers.

As penalization acts after the networks are generated, its impact should be stronger in the best population. Ideally, the chances of generating invalid architec-

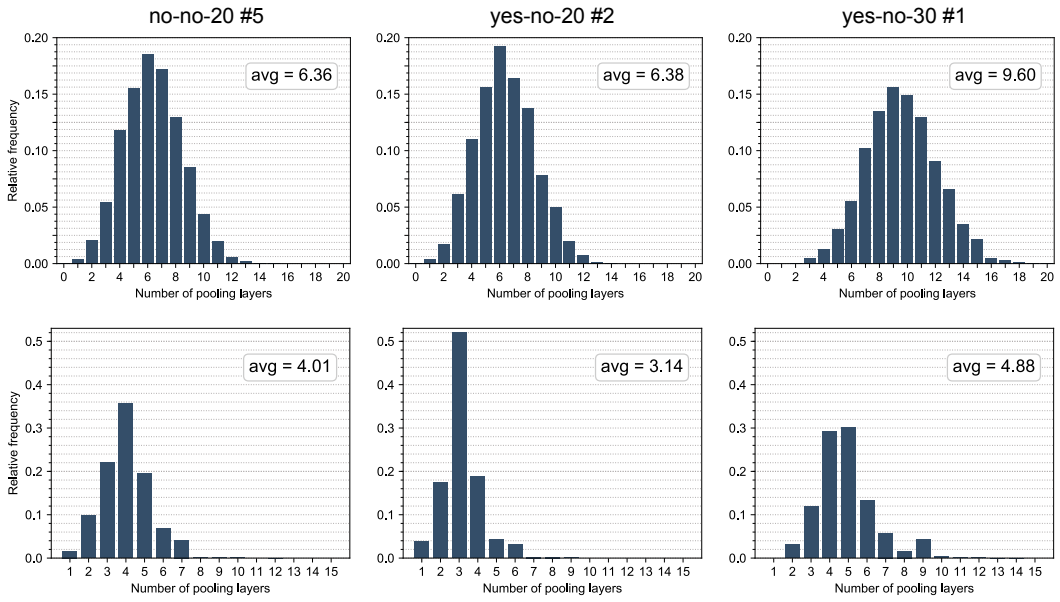

Figure 5.9: Relative frequency of pooling layers in the networks for the generated population (top) and the saved (best) population (bottom). Each column shows the relative counts for the best runs of baseline, 20 nodes with penalization, and 30 nodes with penalization.

tures should decrease with time, since the best structures influence the quantum individuals. However, for the number of iterations we ran Q-NAS, this effect was not apparent.

A possible alternative to the penalization scheme is a mechanism that corrects not only the network at training time but also its description as a classical individual. Thus, Q-NAS would have access to the corrected structures and update the quantum individuals accordingly. We intend to investigate this option in future work.

5.7 Function set

The function lists of Table 5.2 and Table 5.5 used in the previous experiments are somewhat biased toward efficiency, i.e., there are more options of convolutions with fewer filters, which are faster to train. In this section, we explore more challenging function lists in terms of the number of options and their impact on the training time. We created two function lists to represent these scenarios.

The first one is an expanded version of Table 5.2: it contains more convolutional functions, including options with 512 filters (Table 5.14). The structure search space, for a maximum network size of 20, increases from $11^{20} = 6.7 \cdot 10^{20}$ to $15^{20} = 3.3 \cdot 10^{23}$ options (including the invalid ones, as in reality, they are generated). This function set is similar to the one presented in [16].

The second list explores residual units instead of convolutional functions. We adopted two types of units: the *ResidualV1*, with identity shortcut, and the *ResidualV1Pr*, with projection shortcut, both depicted in Figure 2.4. Table 5.15

shows the options we considered. Just like the convolution-based lists, only the pooling operations are responsible for feature map size reduction.

Table 5.14: Functions for expanded convolution set.

function name	function	kernel size	stride	filters	initial probability
conv_k_1_f	ConvBlock	[1, 3, 5]	1	[64, 128, 256, 512]	0.028
max_pool_2_2	MaxPool	2	2	-	0.167
avg_pool_2_2	AvgPool	2	2	-	0.167
no_op	NoOp	-	-	-	0.333

Table 5.15: Functions with residual blocks.

function name	function	kernel size	stride	filters	initial probability
bv1_3_1_f	ResidualV1	3	1	[64, 128, 256]	0.055
bv1p_3_1_f	ResidualV1Pr	3	1	[64, 128, 256]	0.055
max_pool_2_2	MaxPool	2	2	-	0.167
avg_pool_2_2	AvgPool	2	2	-	0.167
no_op	NoOp	-	-	-	0.333

To compare the outcomes, we selected the runs for configuration *yes-no-20* from the last section as our control results. Accordingly, for the runs with the new function sets, we used the evolution parameters of Table 5.1, except for the *penalize_number* parameter, which we set to 3. As in the *yes-no-20* runs, we used the dataset sample 2, along with the RMSProp configuration *rms-2* from Table 5.3.

We repeated the runs for each function set three times; these results and the baseline are listed on Table 5.16. All runtimes for the new function sets are considerably higher than before. This increase is expected because (1) we included many convolutions with more filters in the expanded set, and (2) residual networks are slower to train. It is interesting to note that even in more challenging scenarios, Q-NAS was able to find structures with the same (or better) level of test accuracy. The residual units provided a new best network, with 93.85% of accuracy, found at generation 51 (about 42 hours of execution) of run *#2*.

Figure 5.10 shows quantum individual 0 and the final architecture for the best runs of each function set. Both quantum individuals demonstrate some preference for convolutions with more filters in many nodes. Considering that each residual unit has two convolutional layers, one can notice that the architecture depicted in Figure 5.10 (b) is reasonably deeper (24 layers) than the structure in (a), and the difference in final accuracy is small. On the other hand, our best residual network outperformed a

Table 5.16: Results for each function set.

#	function set	total time	# of layers	# of pool layers	accuracy		
					fitness	retrain validation	retrain test
1	conv	65 h	11	2	0.8000	0.9024	0.8970
2	conv	69 h	14	4	0.8030	0.9308	0.9296
3	conv	70 h	11	2	0.7940	0.9150	0.9183
1	conv exp	198 h	10	4	0.8110	0.9292	0.9295
2	conv exp	199 h	13	4	0.8110	0.9330	0.9270
3	conv exp	210 h	13	2	0.8060	0.9184	0.9179
1	residual	245 h	19	3	0.8310	0.9210	0.9194
2	residual	251 h	24	4	0.8260	0.9428	**0.9385**
3	residual	249 h	24	2	0.8250	0.9300	0.9292

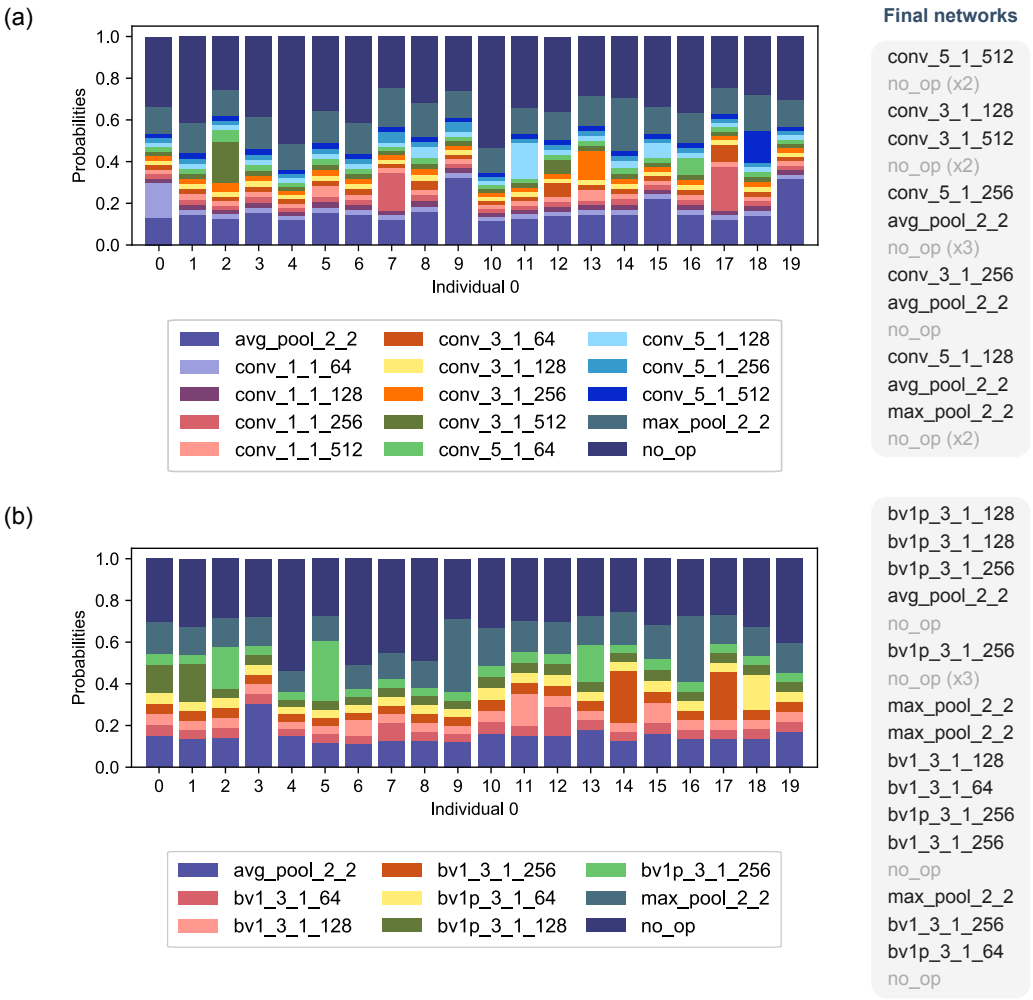

Figure 5.10: Quantum individuals and final architectures for the best runs of each function set: (a) run *conv exp* #1 and (b) run *residual* #2

ResNet with 110 layers (93.57%, as seen in Table 2.1). This final architecture has a quite balanced mixture of each residual unit type, while the mentioned ResNet uses only identity shortcuts.

5.8 Early-stopping

In this section, we complete our analysis of Q-NAS applied to CIFAR-10 by studying the addition of an early-stopping mechanism. Our previous work [72] showed that Q-NAS could benefit from a simple early-stopping method, significantly reducing the total runtime, while maintaining the level of test accuracy. These results, however, were based on Q-NAS runs that took no more than 60 hours to complete. We want to apply this mechanism to more runtime-critical examples, like the ones from the function set experiments (Section 5.7).

The early-stopping criterion is based on the behavior of the best fitness throughout evolution. The curves exhibited a stepwise characteristic with long plateaus, which we illustrate later. In the previous analysis, we related these plateaus not to stagnation but to the difficulty in finding architectures with a stand-out performance. The early-stopping criterion considers the small increments in fitness [72]: *stop if the best individual fitness does not improve above a threshold of* 0.005 *for* 80 *generations*.

Since we store the population descriptions and fitness values, it is possible to apply the criterion on the saved data. Therefore, we used the data from the runs in Table 5.16 and retrained the best individual at the generation indicated by the early-stopping rule. Table 5.17 compares the networks at the last generation and the early-stopping generation, in terms of test accuracy, evolution time, and the number of layers and parameters.

The early-stopping method was able to reduce the evolution time in more than 60% for most of the runs in Table 5.16. Run *#3* for the residual set showed the best improvement on this matter: the early-stopped generation represents 22% of the total execution time. Furthermore, test accuracies were only slightly affected. One can also observe that some accuracies did not change, which means that the best network was already available when the mechanism indicated to stop the evolution. For these runs, Q-NAS found the final structures with only a few iterations.

Regarding the number of parameters, in many cases, the networks from the early-stopped generation have fewer weights to train than the final ones. We highlight the runs in Table 5.17 indicated by *3-c* + and *3-res*: the structures from early-stopping have fewer parameters but higher test accuracy. We attribute this result to the way we evaluate the candidate architectures. The fitness value is an estimation of the network capacity that might not always reflect the architecture with the best

Table 5.17: Results for early-stopping applied to the function set runs: *c 1* (usual convolutional set), *c +* (expanded conv set), and *res* (residual set).

run -set	no early-stopping					with early-stopping				
	gen.	evol. time	L	param. (M)	test acc.	gen.	evol. time	L	param. (M)	test acc.
1-c 1	300	65 h	11	1.66	0.8970	100	22 h	11	1.66	0.8970
2-c 1	300	69 h	14	1.60	0.9296	94	22 h	13	0.55	0.9121
3-c 1	300	70 h	11	0.82	0.9183	163	38 h	11	0.35	0.9102
1-c +	300	198 h	10	5.91	0.9295	92	57 h	12	3.85	0.9263
2-c +	300	199 h	13	3.20	0.9270	90	55 h	13	5.05	0.9205
3-c +	300	210 h	13	4.67	0.9179	93	58 h	13	3.80	0.9370
1-res	300	245 h	19	1.72	0.9194	160	98 h	19	1.72	0.9194
2-res	300	251 h	24	7.07	0.9385	131	80 h	24	7.07	0.9385
3-res	300	249 h	24	7.07	0.9292	87	54 h	21	3.32	0.9304

*gen. = generations; L = number of layers; param. = number of parameters

generalization capabilities. To favor efficiency, we evaluate individuals by training them with a limited dataset for just a few epochs. Therefore, a network can show better accuracy in this reduced dataset, but when trained with the complete data, it might not be the best one. We remark that in all experiments so far, this method did not produce networks with low test accuracy. So, we consider that our approach provides an adequate balance between efficiency and the final quality of results.

Figure 5.11 shows the best and the average fitnesses in the population during evolution, for two runs of Table 5.17. Observe that the best individual is unchanged for many generations (plateaus). Additionally, the best fitness value at the generation indicated by early-stopping (red dot in the graphs) is close to the final one.

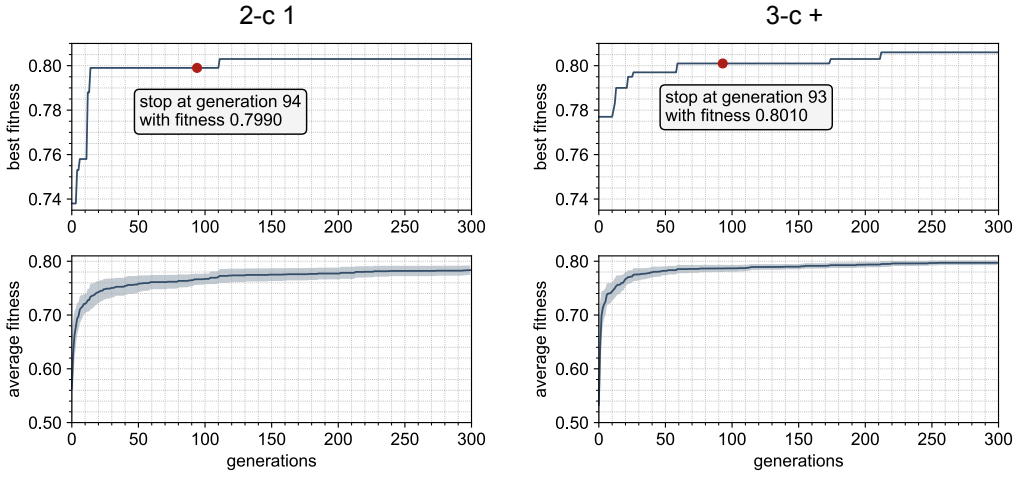

Figure 5.11: Best and average fitness in the population for runs *2-c 1* and *3-c +*. The generation indicated by early-stopping is also shown.

We end the series of CIFAR-10 experiments comparing our results with the state-of-the-art. To support our analysis, we include Table 5.18, which incorporates the literature outcomes of Table 2.1 and our best results presented in this section.

Table 5.18: Comparing our results with some literature models. The '*' marks the methods that used other datasets for the search and applied the network on CIFAR-10.

	accuracy (%)	# params.	GPU days
Hand-designed models			
ResNet [9]	93.57	1.7M	-
VGG [7] as reported by [20]	92.06	15.2M	-
Network in Network (NiN) [8]	91.19	-	-
Maxout [54]	90.70	-	-
NAS			
NAS [2]	96.35	37.4M	22400
EAS [25]	95.11	-	10
Large-scale Evolution [19]	94.60	5.4M	2670
CGP-CNN (ResSet) [20]	94.02	1.68M	28
Q-NAS 2-res + early-stop	**93.85**	**7.07M**	**67**
Q-NAS 3-c+ + early-stop	**93.70**	**3.8M**	**48**
CGP-CNN (ConvSet) [20]	93.25	1.52M	-
MetaQNN [16]	93.08	11.18M	100
Q-NAS 2-c1	**92.96**	**1.6M**	**58**
NASBOT [22]	91.31	-	1.67
NAS with cell search			
XNAS [53]	98.40	7.2M	0.3
DARTS [49]	97.24	3.3M	5
NASNet-A [17]	96.86	3.3M	2000
AmoebaNet-A [21]	96.66	3.2M	3150
Block-QNN-S [18]	96.46*	39.8M	96
Hierarchical Evolution [26]	96.25	-	300
Genetic CNN [46]	92.90	-	17

The run *2-c 1* represents the best accuracy for the smaller convolutional set (92.96%), with a total runtime of 58 GPU days. This accuracy is better than the hand-designed architectures, excluding ResNet. The VGG network contains more than 15 million parameters and presented 92.06% of accuracy, while ours performed better and has only 1.6 million. With the addition of the early-stopping mechanism, we obtained a runtime of 67 GPU days for our best accuracy of 93.85%, surpassing the performance of ResNet.

The NAS systems that apply the meta-architecture strategy (bottom of Table 5.18) present the best results regarding accuracy values and, in some cases, also efficiency. However, as discussed in Section 2.2, this strategy represents a strong

bias in the search, which we proposed to reduce. Therefore, we focus our analysis on the methods that do not follow this approach.

The methods NAS and EAS provide the best accuracies, but NAS spent $22\,400$ GPU days, which is considerably more than our runtimes. On the other hand, EAS achieved 95.11% of accuracy using only ten GPU days. As mentioned in Section 2.2, EAS uses a seed network to start the search, which is not identical to the meta-architecture approach but raises the same questions about bias. Also, according to the authors, the seed network already presents 87% of validation accuracy.

The method referred to as *Large-scale Evolution* in Table 5.18 reached 94.60% of accuracy, which is slightly better than our results, but using significantly more computational resources compared to Q-NAS. Considering the MetaQNN approach, all of our results are better regarding the number of parameters and execution time. Our network *2-c 1* has only 14% of the total parameters in the MetaQNN structure, with a difference in the accuracy value of just 0.12 percentage points.

The CGP-CNN approach is competitive with respect to efficiency: they achieve 94.02% of accuracy in only 28 GPU days. However, in this experiment, the authors report the evaluation of 600 networks [20], which is ten times less than our runs. In Subsection 5.3.5, we varied the number of individuals per generation and, consequently, the number of evaluated networks. In the runs where we tested $1\,500$ structures (five individuals per generation), the maximum runtime was 8.5 GPU days. Furthermore, CGP-CNN uses specially designed learning schedules for the final retraining phase. As we confirmed in our retrain study of Section 5.5, this approach can improve accuracies for the CIFAR-10 dataset. Training our best network with this particular scheme increased the accuracy from 93.85% to 94.18%. However, as also mentioned before, it may require tuning for each different dataset, which goes against the idea of automation.

In summary, our results indicate that Q-NAS is competitive regarding the balance between performance, efficiency, and automation. Without using meta-architectures or specially designed learning schemes, we were able to outperform hand-designed models and other NAS methods, in less than 70 GPU days.

5.9 CIFAR-100

After the comprehensive analysis presented in the previous experiments, in this section, we proceed to our second goal: applying Q-NAS to CIFAR-100. The idea is to repeat some experiments we conducted for CIFAR-10 to evaluate the performance of Q-NAS when applied to a more challenging dataset. More specifically, we selected the sampling and function set experiments from Section 5.4 and Section 5.7, respectively. CIFAR-100 has the same number of training images as

CIFAR-10, but 100 classes instead of 10. Thus, with fewer examples per class, it is interesting to analyze the impact of sampling in the final results. Additionally, with a more difficult task, it is relevant to evaluate the difference in using simple or complex function sets.

The NAS literature does not provide many results on CIFAR-100, and some of them perform the search on other datasets to only retrain the final network on CIFAR-100. Table 5.19 includes the methods from Table 2.1 that also present results using CIFAR-100. Table 5.19 serves as the baseline to evaluate our results, which will be introduced in the next subsections.

Table 5.19: Results from the literature on CIFAR-100. The '*' marks the methods that used other datasets for the search and applied the network on CIFAR-100.

	accuracy (%)	# params.	GPU days
Hand-designed models			
ResNet-1001 [77]	77.30	10.2M	-
ResNet-164 [77]	75.67	1.7M	-
Network in Network (NiN) [8]	64.32	-	-
Maxout [54]	61.43	-	-
NAS			
Large-scale Evolution [19]	77.00	40.4M	-
MetaQNN [16]	72.86*	11.18M	100
NAS with cell search			
XNAS-Small [53]	86.40*	3.7M	0.3
Block-QNN-S [18]	81.94	39.8M	96
Genetic CNN [46]	70.97*	-	-

5.9.1 Sampling

Repeating the methodology presented on Section 5.4, we generated three samples from the CIFAR-100 dataset containing 10 000 images (9 000 for training and 1 000 for validation). Recall from Section 5.1 that our sampling procedure ensures class balance.

We executed Q-NAS three times for each sample. For these runs, we used the same settings as the penalization experiments (Section 5.6) with a maximum network size of 30. This configuration corresponds to the optimizer setup *rms-2* in Table 5.3 and the parameters of Table 5.1, with $max\_num\_nodes = 30$ and $penalize\_number = 3$. Like in Section 5.4, the selected function set is the one from Table 5.2. The results for these runs are listed in Table 5.20.

Table 5.20: Results for each sample of CIFAR-100.

#	sample	total time	# of layers	param. (M)	# of pool layers	accuracy		
						fitness	retrain validation	retrain test
1	1	73 h	12	1.18	3	0.3650	0.6508	0.6495
2	1	74 h	14	2.94	2	0.3600	0.6522	0.6431
3	1	73 h	16	1.03	3	0.3700	0.6610	0.6595
1	2	72 h	15	1.37	3	0.3710	0.6462	0.6469
2	2	72 h	16	1.45	4	0.3890	0.6826	0.6775
3	2	73 h	16	0.63	4	0.3820	0.6114	0.6117
1	3	76 h	17	0.78	4	0.3530	0.6608	0.6551
2	3	71 h	13	1.98	2	0.3330	0.6692	0.6727
3	3	74 h	13	2.02	5	0.3470	0.7034	**0.6995**

Table 5.19 demonstrates that CIFAR-100 is more challenging than CIFAR-10. The hand-designed models MaxOut and NiN present test accuracies of 61.43% and 64.32% for CIFAR-100, respectively, while for CIFAR-10, they both show more than 90% of accuracy. For all samples, Q-NAS was able to generate networks that outperform these hand-designed models, which is a significant result, as we applied the algorithm without adjusting any parameter. However, some samples reached better accuracies than others, which might indicate that the impact is not negligible.

Our best accuracy (69.95%) is close to the value of 70.97% reported by the authors of Genetic-CNN. This particular network generated by QNAS falls into the case where its description has more pooling layers than the maximum allowed. The structure was found early (at generation 27) with significantly better fitness, so even with penalization, the invalid description was kept in the population.

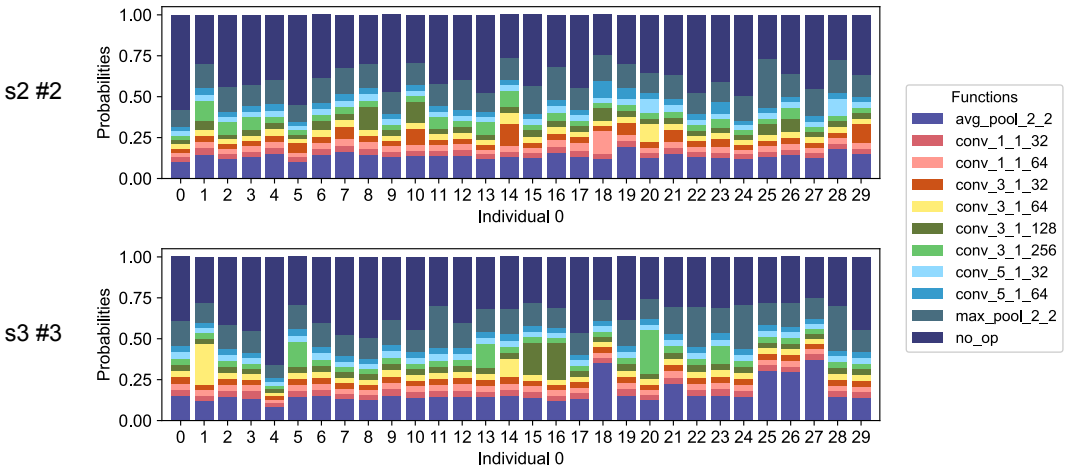

Figure 5.12: Quantum individuals for the best runs for sample 2 (#2) and sample 3 (#3).

Figure 5.12 shows quantum individuals from the two best runs in Table 5.20. Note that some nodes demonstrate a preference for convolutions with more filters. However, unlike the CIFAR-10 experiments, there are also many nodes with high probabilities for convolutions with 64 and 32 filters. The final structures corroborate with this idea: in most of the runs from Table 5.20, the number of parameters is less than 2 million, even for networks with more than 16 layers.

5.9.2 Function set

We now repeat the function set experiments from Section 5.7 using the CIFAR-100 dataset. The configuration for Q-NAS is maintained: evolution parameters from Table 5.1 with *penalize_number* $= 3$ and optimizer configuration *rms-2* in Table 5.3. For the expanded convolutional set (Table 5.14) and residual set (Table 5.15), we executed Q-NAS three times. We selected *sample-3* runs from the last subsection as control, and we used this dataset sample for the new runs (results in Table 5.21).

Table 5.21: Results for each function set in the CIFAR-100 task.

#	function set	total time	# of layers	param. (M)	# of pool layers	accuracy		
						fitness	retrain validation	retrain test
1	conv	76 h	17	0.78	4	0.3530	0.6608	0.6551
2	conv	71 h	13	1.98	2	0.3330	0.6692	0.6727
3	conv	74 h	13	2.02	5*	0.3470	0.7034	0.6995
1	conv exp	216 h	13	13.64	3	0.3980	0.7182	0.7051
2	conv exp	211 h	10	5.69	4	0.3960	0.7162	0.7019
3	conv exp	220 h	8	2.86	4	0.4010	0.6602	0.6562
1	residual	245 h	21	6.57	3	0.4080	0.6896	0.6848
2	residual	240 h	20	6.25	4	0.4070	0.7460	**0.7423**
3	residual	247 h	19	4.46	3	0.4140	0.7294	0.7177

Our new best accuracies outperform GeneticCNN (70.97%) and MetaQNN (72.86%) results, which were obtained by retraining the network found in the CIFAR-10 case. Compared to our CIFAR-10 outcomes, the test accuracy difference between runs of the same configuration is higher, although the fitness values are similar. A possible reason is the quality of the network performance estimation during evolution. With a more challenging dataset, the 50 epochs of training might not be sufficient to distinguish a considerably better structure. Furthermore, we are working with networks of much lower depth than the state-of-the-art models. To achieve higher accuracies, we might need to experiment with deeper architectures.

Observe that both ideas (increase training epochs and network depth) should raise the evolution time. Therefore, finding a satisfactory balance is a relevant topic for future investigation.

Figure 5.13 illustrates the final networks of the best runs in Table 5.21 for each function set. Observe that, in all structures, the convolutions with more filters predominate. In run *1-c +*, there are convolutions with all available kernel sizes, contrasting with well-known hand-designed models, such as Alexnet [6] or VGG [7].

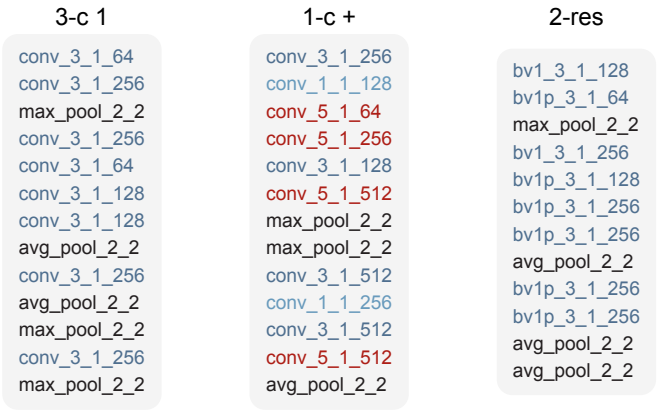

Figure 5.13: Final networks for the best runs of each function set.

We also applied the early-stopping mechanism described in Section 5.8 to the runs of Table 5.21 with accuracies above 69%; the results can be seen in Table 5.22. For all tested runs, except *2-c +*, the mechanism indicated to stop when the final network was available, which means no accuracy loss. Furthermore, there was a significant runtime decrease in almost all situations: in the best case (*3-res*), the early-stopped generation represents 27% of the total time. Run *2-res* had a more modest reduction: generation 235 represents 78% of the total runtime. Compared to CIFAR-10 results, the number of GPU days is in the same range for most cases.

Table 5.22: Results for early-stopping applied to the best function set runs on CIFAR-100: *c 1* (usual convolutional set), *c +* (expanded conv set), and *res* (residual set).

run -set	no early-stopping					with early-stopping				
	gen.	GPU days	L	param. (M)	test acc.	gen.	GPU days	L	param. (M)	test acc.
3-c 1	300	62	13	2.02	0.6995	107	21	13	2.02	0.6995
1-c +	300	180	13	13.64	0.7051	175	105	13	13.64	0.7051
2-c +	300	176	10	5.69	0.7019	103	62	10	8.58	0.6789
2-res	300	200	20	6.25	0.7423	235	156	20	6.25	0.7423
3-res	300	206	19	4.46	0.7177	85	55	19	4.46	0.7177

*gen. = generations; L = number of layers; param. = number of parameters

We highlight that Q-NAS could reach accuracy levels comparable to other methods for CIFAR-100, without any adjustments on parameter values or the early-stopping mechanism, indicating robustness in the algorithm.

5.10 Case study: seismic data

Researchers have also applied NAS to real data. A very recent example is to use NAS to create networks for emulating complex computer simulations of natural phenomena [78, 79]. We conclude our experiments with a case study on seismic image classification, applying Q-NAS to real datasets, as detailed below.

Geoscientists usually examine seismic images to study the subsurface of the Earth, as they represent the geologic structures beneath the ground. The process to generate these images starts with data acquisition: an intense sound source directs waves into the ground, which are reflected and then recorded by geophones. Finally, this signal is processed, producing seismic images that experts will interpret [37]. In a crucial step of the analysis, the expert separates layers of rock with different properties by looking for visual patterns that might distinguish them. Then, he marks the *horizons*, which are the division lines between layers [80]. This process can take months to complete, especially with large amounts of data. Both academia and industry have been investing in methods to accelerate this procedure [37, 80].

Figure 5.14 shows examples of seismic images from two real datasets: Penobscot 3D survey [81] and Netherlands F3 Block [82]. They both consist of a horizontal stack of 2D seismic images (*slices*) that creates a 3D volume (*seismic cube*). The vertical axis represents depth, and the other two perpendicular directions are called *inline* and *crossline* (see Figure 5.15 (a)). Penobscot contains 481 crossline and 601 inline slices, with dimensions 601×1501 and 481×1501 pixels, respectively. The Netherlands dataset has 951 crosslines of size 651×462 pixels, and 651 inlines of 951×462 pixels. Figure 5.14 also depicts the layers separated by an expert. From the machine learning point of view, the layers (areas between horizons) can be considered categories that a model can classify. In the ideal scenario, the expert would annotate a minimum amount of slices, the model would learn from this small set, and then classify the rest of the images in the cube.

Civitarese et al. [37] developed deep networks specifically for this classification task, focusing on efficiency. They carefully designed the models and used a particular learning rate schedule. We want to verify if Q-NAS can produce networks with equivalent performance to their best model *Danet-3*. To compare the networks' performance fairly, we followed the preprocessing method detailed in [37] to generate datasets from the seismic images of Penobscot and Netherlands. Then, we trained Danet-3 with these datasets, using the learning scheme described by the authors, in-

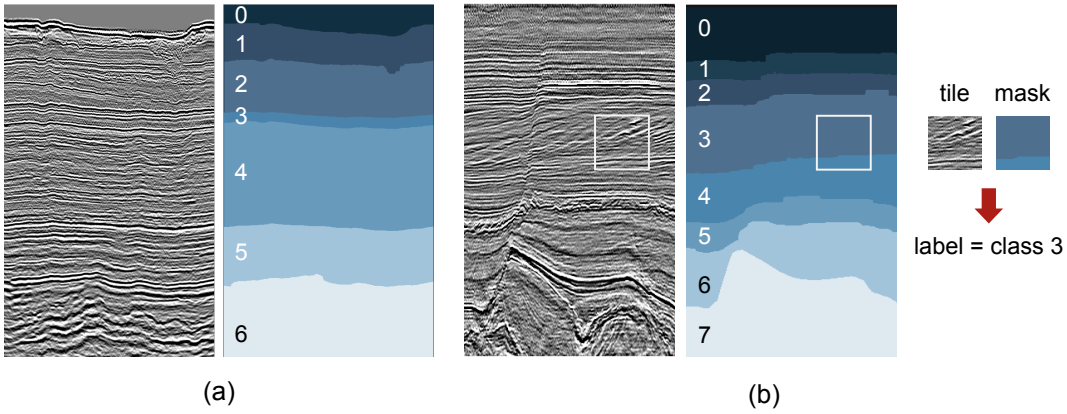

Figure 5.14: Examples of (cropped) inline slices and respective layers for (a) Penobscot and (b) Netherlands. In (b), we show a tile collected from the image and how it is labeled for a classification task.

cluding the values for the training parameters. We remark that the results in [37] were obtained with a different test set, so we could not compare directly with ours.

As detailed in [37], to get samples from each layer in the seismic images, it is necessary to break them into smaller parts (see Figure 5.14 (b)). The authors use a sliding window mechanism to get tiles of size 40×40 pixels. Note that the smaller the stride of the sliding window, the bigger is the number of generated tiles. They allow for a noise of 30%, that is, 70% of the tile must belong to a single layer; otherwise, the tile is discarded. Also, the original images have float ranges, which they rescale to the regular grayscale (0 to 255) before breaking them into tiles.

To generate the training set, we selected regularly spaced inline slices from the cubes (20 for Penobscot and 25 for Netherlands). For the test set, we randomly picked 50 slices from the blocks in between the training slices, guaranteeing similar numbers from each block (see Figure 5.15 (b)). Next, we applied the preprocessing procedure for each set. After the training and test tiles were ready, we randomly selected 20% of the training tiles to comprise the validation set. Also, we guarantee class balance by discarding tiles from the classes with more examples. Table 5.23 lists the characteristics of the generated tile datasets.

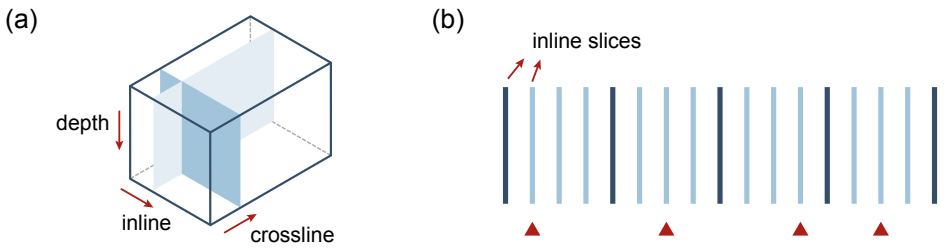

Figure 5.15: (a) Seismic cube with highlighted *inline* (dark blue) and *crossline* slices. (b) Selection scheme for train and test sets. The dark blue slices go to the training set. The test set is randomly selected from the areas between training slices (red triangles).

Table 5.23: Characteristics of each tile dataset.

dataset	tile size	stride	classes	train tiles	valid tiles	test tiles
Penobscot	40×40	5	7	3 420	381	9 030
Netherlands	40×40	10	8	4 737	527	10 336

Notice that, unlike the CIFAR cases, the challenge here is to work with limited amounts of data. Consequently, we did not reduce the datasets for evolution, as they have less than 5000 training examples, and we decreased the batch size to 64. Moreover, for a fair comparison with Danet-3, we did not apply data augmentation nor the mean subtraction we have used in all runs so far. The evolution parameters, on the other hand, are kept equal to the ones in Subsection 5.9.1 with a single exception: the number of classical individuals. We want to simulate a restricted resource scenario, with only a few GPUs available to run Q-NAS. Therefore, we set *repetition* to 1, so the five quantum individuals produce five structures per generation. Table 5.24 and Table 5.25 list the results of the runs for each dataset.

Table 5.24: Results for Penobscot dataset.

#	total time	GPU days	# of layers	param. (M)	# of pool layers	accuracy		
						fitness	retrain validation	retrain test
1	31 h	6.5	13	0.50	2	0.9869	0.9816	0.9730
2	32 h	6.6	14	1.05	2	0.9895	0.9816	0.9672
3	31 h	6.5	13	0.44	3	0.9895	0.9869	0.9812
4	33 h	6.8	13	0.34	3	0.9895	0.9869	0.9781
5	32 h	6.6	10	0.82	3	0.9895	0.9921	**0.9857**
							Danet-3 - test accuracy:	0.9745

Table 5.25: Results for Netherlands dataset.

#	total time	GPU days	# of layers	param. (M)	# of pool layers	accuracy		
						fitness	retrain validation	retrain test
1	41 h	8.5	13	0.53	3	0.9772	0.9848	0.9790
2	40 h	8.4	14	1.17	3	0.9791	0.9905	**0.9818**
3	38 h	7.9	11	0.46	3	0.9848	0.9810	0.9732
4	39 h	8.2	13	0.57	3	0.9772	0.9753	0.9761
5	38 h	7.8	11	0.33	3	0.9810	0.9867	0.9803
							Danet-3 - test accuracy:	0.9638

For both datasets, Q-NAS was able to generate networks that outperform Danet-3. Furthermore, these structures have significantly fewer parameters than Danet-3, which has 14.98 millions of weights. In the best case for Penobscot, the final network showed 98.57% of accuracy, and it has only 0.82 millions of parameters. For the Netherlands dataset, run *#5* reached 98.03% of accuracy, with a structure that has 0.33 million weights. Also, all runs spent less than 41 hours to complete (8.5 GPU days, considering the five GPUs that were used).

Figure 5.16 illustrates a quantum individual and the final network for run *#5* of the Netherlands dataset. Notice the preference for functions with a kernel size of 5 and low probabilities for *conv_3_1_256*. This behavior was also observed in other runs of this experiment, which is reasonably different from the CIFAR results. This contrast is interesting since, in both cases, the input images have similar sizes, but a notably different visual aspect.

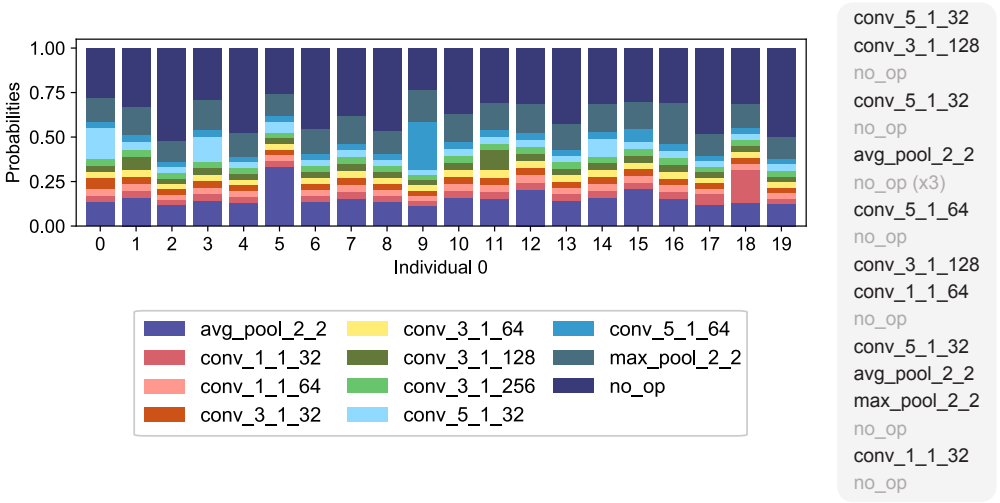

Figure 5.16: Quantum individual 0 and final network for run *#5* of the Netherlands dataset.

To close this section, we remark that Q-NAS was applied to seismic datasets that represent real problems in the geoscience domain. Based on our results, it is possible to conclude that Q-NAS could automatically find structures to solve the seismic classification task in less than 8.5 GPU days, surpassing specially designed models in accuracy and efficiency.

6
Conclusions

In this work, we introduced a new quantum-inspired algorithm to search for deep neural network structures. The proposed quantum representation allows for Q-NAS to evolve both the architecture and some numerical hyperparameters.

Our new quantum representation for deep networks provides considerable flexibility to the user, as he/she can choose how complex will be the search space of node functions. Furthermore, the probabilistic quality of the quantum individual offers another degree of flexibility, as initial probabilities can be defined to favor some functions over the others.

Using the CIFAR-10 dataset, we varied the values of the Q-NAS' parameters, covering a wide range of options for many of them. We observed that some parameters affect more the evolution process than others, but the final accuracy is similar in most of the cases. Our results are promising regarding the robustness of Q-NAS, as satisfactory outcomes were obtained with many different parameter setups.

We also developed an experiment to verify if Q-NAS can evolve both the network structure and some hyperparameters. We selected a few numerical hyperparameters to optimize and compared them to the case with fixed values. In both situations, Q-NAS was able to produce similar results, but we did not observe a clear benefit in optimizing the selected hyperparameters.

Additionally, we investigated the impact of different retraining schemes on the final test accuracy. We observed that, although special learning schedules can lead to better results, the cosine schedule provides similar accuracies (less than 0.5 percentage points of difference). Also, it requires only the definition of an initial learning rate, with no further tuning. We incorporated the cosine scheme in our retrain phase, which improved our test accuracies.

The study on penalization showed that our method could not decrease the number of invalid structures generated during evolution, but it helped to remove such networks from the saved population. In this way, it reduced the chances of invalid individuals affecting the final results. We believe that correcting not only the network at training time but also its description in Q-NAS may be an alternative that could help reduce the generation of invalid structures.

We obtained our best accuracies for CIFAR-10 when exploring a search space containing convolutions with more filters and also when searching in the space of

residual networks. For these runs, we investigated the effect of adding a simple early-stopping mechanism, which improved the runtimes significantly. Our best residual network achieved 93.85% of test accuracy on the CIFAR-10 task, with a runtime of 67 GPU days considering the early-stop. The best convolutional network reached 93.70% of accuracy, requiring 48 GPU days with early-stopping. These accuracy values overcome the performance of hand-designed models and also other NAS works. Furthermore, these results indicate that Q-NAS is competitive regarding the balance between accuracy, runtime efficiency, and automation. We emphasize that our algorithm does not rely on meta-architectures, which represent strong human bias in the search, or specially designed learning schemes that require tuning.

We extended our investigation by applying Q-NAS to the more challenging dataset CIFAR-100, and we were able to outperform the hand-designed models Maxout [54] and NiN [8]. Our best accuracy of 74.23% from a residual network with 24 layers is comparable to that of a ResNet with 164 layers and only 2.8 percentage points worse than the NAS method in [19]. We highlight that in the CIFAR-100 experiments, we only applied Q-NAS with previously defined configurations, i.e., we did not perform any parameter adjustment.

Finally, we applied Q-NAS to real datasets to solve the seismic classification task, which lies in the image recognition context. In a restricted resource scenario, Q-NAS' networks were able to outperform a hand-designed model (Danet-3 [37]), specially developed for this task. For the Penobscot dataset, our best network (with 0.82 million parameters) reached an accuracy of 98.57%, while Danet-3, which has more than 14 million weights, achieved 97.45%. In the Netherlands dataset, the outcomes were similar: Danet-3 showed 96.38% of accuracy, and our best result was 98.18%, with a network of 1.17 million parameters. For the two real datasets used in this work, Q-NAS could automatically find structures to solve the seismic classification task in less than 8.5 GPU days.

As future work, we intend to study further the invalid networks issue. More specifically, we want to experiment with the correction scheme introduced in Section 5.6: when fixing the networks at training time, we can also modify the individual description accordingly. We believe that this method might reduce the generation of invalid networks faster (in terms of evolution iterations) than the penalization scheme. Also, it is possible to combine the description correction with penalization, to intensify the rejection of invalid structures.

Additionally, we plan to extend our parameter analysis, studying their impact when working with other datasets. Although we obtained satisfactory results with a similar configuration for all tested datasets, a more in-depth investigation is needed. This analysis is also important to select a configuration that is appropriate for many datasets.

Another subject we plan to investigate is the analysis of the generated structures. We believe that the thousands of networks we produce in the evolutionary process can provide interesting insights about the topologies. It is possible to verify if specific sequences of functions frequently appear so we can encapsulate them in future runs. Also, we can compare the structures generated for different datasets, to investigate a potential relation between the topology and the characteristics of the data.

In this work, we framed Q-NAS in the image classification context to run our experiments and compare them to the literature results. However, Q-NAS is not restricted to this task, and we want to extend our analysis to other problems suitable for neural networks, such as regression tasks.

Finally, we intend to conduct a more comprehensive study on the evolution of hyperparameters. We want to verify which ones should be evolved, that is, which parameters notably improve the results when optimized. Note that it is possible to extend this idea to the dataset preprocessing, which usually has many options and parameter values to select.

Bibliography

[1] F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., *Automatic Machine Learning: Methods, Systems, Challenges*. Springer, 2018.

[2] B. Zoph and Q. Le, "Neural architecture search with reinforcement learning," *arXiv:1611.01578 [cs]*, November 2016.

[3] G. Hinton, S. Osindero, and Y. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[4] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, January 2009.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, ser. Adaptive Computation and Machine Learning series. MIT Press, 2016.

[6] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012, pp. 1097–1105.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556 [cs]*, September 2014.

[8] M. Lin, Q. Chen, and S. Yan, "Network In Network," *arXiv:1312.4400 [cs]*, December 2013.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv:1512.03385 [cs]*, December 2015.

[10] P. Angeline, G. Saunders, and J. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 54–65, January 1994.

[11] K. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[12] M. Vellasco, A. Cruz, and A. Pinho, "Quantum-inspired evolutionary algorithms applied to neural modeling," *IEEE World Conference on*

Computational Intelligence, Plenary and Invited Lectures, pp. 125–150, 2010.

[13] M. Silva, M. Vellasco, and E. Cataldo, “Evolving Spiking Neural Networks for Recognition of Aged Voices,” *Journal of Voice*, 2017.

[14] P. Paiva, M. Vellasco, and J. Amaral, “Quantum-Inspired Optimization of Echo State Networks Applied to System Identification,” in *2018 IEEE Congress on Evolutionary Computation*, July 2018, pp. 1–8.

[15] A. Krizhevsky, “Learning multiple layers of features from tiny images,” University of Toronto, Tech. Rep., 2009.

[16] B. Baker, O. Gupta, N. Naik, and R. Raskar, “Designing neural network architectures using reinforcement learning,” in *International Conference on Learning Representations (ICLR)*, April 2017.

[17] B. Zoph, V. Vasudevan, J. Shlens, and Q. Le, “Learning transferable architectures for scalable image recognition,” in *2018 Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018, pp. 8697–8710.

[18] Z. Zhong, J. Yan, W. Wu, J. Shao, and C. Liu, “Practical block-wise neural network architecture generation,” in *2018 Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018, pp. 2423–2432.

[19] E. Real *et al.*, “Large-scale evolution of image classifiers,” in *Proceedings of the 34th International Conference on Machine Learning*, vol. 70, June 2017, pp. 2902–2911.

[20] M. Suganuma, S. Shirakawa, and T. Nagao, “A genetic programming approach to designing convolutional neural network architectures,” in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2017, pp. 497–504.

[21] E. Real, A. Aggarwal, Y. Huang, and Q. Le, “Regularized evolution for image classifier architecture search,” *arXiv:1802.01548 [cs]*, February 2018.

[22] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. Xing, “Neural architecture search with bayesian optimisation and optimal transport,” in *Advances in Neural Information Processing Systems 31*, 2018, pp. 2020–2029.

[23] A. Brock, T. Lim, J. Ritchie, and N. Weston, “SMASH: One-shot model architecture search through hypernetworks,” in *International Conference on Learning Representations (ICLR)*, 2018.

[24] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, “Path-level network transformation for efficient architecture search,” in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

[25] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, “Efficient architecture search by network transformation,” in *AAAI Conference on Artificial Intelligence*, 2018.

[26] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, “Hierarchical representations for efficient architecture search,” in *International Conference on Learning Representations (ICLR)*, 2018.

[27] M. Platel, S. Schliebs, and N. Kasabov, “Quantum-inspired evolutionary algorithm: A multimodel EDA,” *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 6, pp. 1218–1232, December 2009.

[28] K. Han and J. Kim, “Quantum-inspired evolutionary algorithm for a class of combinatorial optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 6, pp. 580–593, December 2002.

[29] A. da Cruz, M. Vellasco, and M. Pacheco, “Quantum-inspired evolutionary algorithm for numerical optimization,” in *2006 IEEE International Conference on Evolutionary Computation*, July 2006, pp. 2630–2637.

[30] ——, “Quantum-inspired evolutionary algorithms applied to numerical optimization problems,” in *IEEE Congress on Evolutionary Computation*, July 2010, pp. 1–6.

[31] M. Cardoso, M. Silva, M. Vellasco, and E. Cataldo, “Quantum-inspired features and parameter optimization of spiking neural networks for a case study from atmospheric,” *Procedia Computer Science*, vol. 53, pp. 74–81, 2015.

[32] F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro, “DENSER: deep evolutionary network structured representation,” *Genetic Programming and Evolvable Machines*, vol. 20, no. 1, pp. 5–35, March 2019.

[33] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, April 1980.

[34] Y. LeCun, Y. Bengio, and G. Hinton, “Deep Learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[35] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, “Handwritten digit recognition with a back-propagation network,” in *Advances in Neural Information Processing Systems 2*, 1990, pp. 396–404.

[36] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, November 1998.

[37] D. Chevitarese, D. Szwarcman, E. Brazil, and B. Zadrozny, “Efficient classification of seismic textures,” in *2018 International Joint Conference on Neural Networks (IJCNN)*, July 2018, pp. 1–8.

[38] O. Russakovsky *et al.*, “ImageNet large scale visual recognition challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[39] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015, pp. 448–456.

[40] P. Luo, X. Wang, W. Shao, and Z. Peng, “Towards understanding regularization in batch normalization,” in *International Conference on Learning Representations (ICLR)*, 2019.

[41] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, p. 529, 2015.

[42] R. Sutton and A. Barto, *Reinforcement learning: an introduction*, 2nd ed. MIT Press, 2018.

[43] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv:1707.06347 [cs]*, 2017.

[44] A. Eiben and J. Smith, *Introduction to Evolutionary Computing*, ser. Natural Computing Series. Springer Berlin Heidelberg, 2007.

[45] Z. Michalewicz, *Genetic algorithms + data structures*, ser. Artificial intelligence. Springer-Verlag, 1992.

[46] L. Xie and A. Yuille, “Genetic CNN,” in *2017 IEEE International Conference on Computer Vision (ICCV)*, October 2017, pp. 1388–1397.

[47] B. Wang, Y. Sun, B. Xue, and M. Zhang, “Evolving Deep Neural Networks by Multi-objective Particle Swarm Optimization for Image Classification,” in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2019, pp. 490–498.

[48] H. Jin, Q. Song, and X. Hu, “Auto-Keras: An Efficient Neural Architecture Search System,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 1946–1956.

[49] H. Liu, K. Simonyan, and Y. Yang, “DARTS: Differentiable Architecture Search,” in *International Conference on Learning Representations (ICLR)*, 2019.

[50] X. Dong and Y. Yang, “Searching for a Robust Neural Architecture in Four GPU Hours,” in *2019 Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[51] T. Elsken, J. Metzen, and F. Hutter, “Simple and efficient architecture search for convolutional neural networks,” *arXiv:1711.04528 [cs]*, November 2017.

[52] Z. Luo, P. Wang, Y. Li, W. Zhang, W. Tang, and M. Xiang, “Quantum-inspired evolutionary tuning of SVM parameters,” *Progress in Natural Science*, vol. 18, no. 4, pp. 475–480, 2008.

[53] N. Nayman, A. Noy, T. Ridnik, I. Friedman, R. Jin, and L. Zelnik, “XNAS: Neural Architecture Search with Expert Advice,” in *Advances in Neural Information Processing Systems 32*, 2019, pp. 1975–1985.

[54] I. Goodfellow, D. Farley, M. Mirza, A. Courville, and Y. Bengio, “Maxout networks,” *arXiv:1302.4389v4 [cs]*, 2013.

[55] E. Rieffel and W. Polak, “An introduction to quantum computing for non-physicists,” *ACM Computing Surveys*, vol. 32, no. 3, pp. 300–335, September 2000.

[56] A. da Cruz, “Quantum-inspired evolutionary algorithms for problems based on numerical representation,” PhD Dissertation, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, 2007.

[57] M. Moore and A. Narayanan, “Quantum-inspired computing,” Department of Computer Science, University of Exeter, Tech. Rep., 1995.

[58] G. Zhang, “Quantum-inspired evolutionary algorithms: a survey and empirical study,” *Journal of Heuristics*, vol. 17, no. 3, pp. 303–351, June 2011.

[59] A. Narayanan and M. Moore, “Quantum-inspired genetic algorithms,” in *Proceedings of IEEE International Conference on Evolutionary Computation*, May 1996, pp. 61–66.

[60] Kuk-Hyun Han and Jong-Hwan Kim, “Genetic quantum algorithm and its application to combinatorial optimization problem,” in *Proceedings of the 2000 Congress on Evolutionary Computation.*, vol. 2, 2000, pp. 1354–1360.

[61] A. da Cruz, M. Vellasco, and M. Pacheco, *Quantum-Inspired Evolutionary Algorithm for Numerical Optimization*. Springer Berlin Heidelberg, 2007, pp. 19–37.

[62] L. da Silveira, R. Tanscheit, and M. Vellasco, “Quantum inspired evolutionary algorithm for ordering problems,” *Expert Systems with Applications*, vol. 67, pp. 71–83, 2017.

[63] J. Xiao, Y. Yan, Y. Lin, L. Yuan, and J. Zhang, “A quantum-inspired genetic algorithm for data clustering,” in *2008 IEEE Congress on Evolutionary Computation*, June 2008, pp. 1513–1519.

[64] A. Ramos and M. Vellasco, “Quantum-inspired evolutionary algorithm for feature selection in motor imagery EEG classification,” in *2018 IEEE Congress on Evolutionary Computation*, July 2018, pp. 1–8.

[65] C. Pereira, D. Dias, M. Vellasco, F. Viana, and L. Martí, “Crude oil refinery scheduling: Addressing a real-world multiobjective problem through genetic programming and dominance-based approaches,” in *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO)*, 2018, p. 1821–1828.

[66] D. Dias and M. Pacheco, “Quantum-Inspired Linear Genetic Programming as a Knowledge Management System,” *The Computer Journal*, vol. 56, no. 9, pp. 1043–1062, 08 2012.

[67] C. da Silva, D. Dias, C. Bentes, M. Pacheco, and L. Cupertino, “Evolving gpu machine code,” *Journal of Machine Learning Research*, vol. 16, no. 22, pp. 673–712, 2015.

[68] M. Silva, A. Koshiyama, M. Vellasco, and E. Cataldo, “Evolutionary features and parameter optimization of spiking neural networks for unsupervised learning,” in *2014 International Joint Conference on Neural Networks (IJCNN)*, July 2014, pp. 2391–2398.

[69] O. Travis E, “A guide to numpy,” USA, 2006.

[70] D. Dias, “Quantum-inspired Linear Genetic Programming,” PhD Dissertation, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, 2010.

[71] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification,” in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034.

[72] D. Szwarcman, D. Civitarese, and M. Vellasco, “Q-nas revisited: Exploring evolution fitness to improve efficiency,” in *2019 Brazilian Conference on Intelligent Systems (BRACIS)*, October 2019, pp. 509–514.

[73] R. Istrate, F. Scheidegger, G. Mariani, D. Nikolopoulos, C. Bekas, and A. C. I. Malossi, “TAPAS: Train-less accuracy predictor for architecture search,” *arXiv:1806.00250 [cs]*, June 2018.

[74] Y. Sun, B. Xue, and M. Zhang, “Automatically evolving CNN architectures based on blocks,” *arXiv:1810.11875 [cs]*, October 2018.

[75] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv:1412.6980 [cs]*, 2014.

[76] M. Wistuba, “Finding competitive network architectures within a day using UCT,” *arXiv:1712.07420 [cs]*, December 2017.

[77] K. He, X. Zhang, S. Ren, and J. Sun, “Identity mappings in deep residual networks,” in *European Conference on Computer Vision*, 2016, pp. 630–645.

[78] M. Kasim *et al.*, “Up to two billion times acceleration of scientific simulations with deep neural architecture search,” *arXiv:2001.08055 [cs]*, 2020.

[79] M. Hutson, “Ai shortcuts speed up simulations by billions of times,” *Science*, vol. 367, no. 6479, pp. 728–728, 2020.

[80] D. Civitarese, D. Szwarcman, E. V. Brazil, and B. Zadrozny, “Semantic segmentation of seismic images,” *arXiv:1905.04307 [cs]*, 2019.

[81] L. Baroni, R. Silva, R. Ferreira, D. Chevitarese, D. Szwarcman, and E. Vital Brazil, “Penobscot interpretation dataset,” July 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1341774

[82] ——, “Netherlands f3 interpretation dataset,” September 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1471548

[83] M. Abadi *et al.*, “Tensorflow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org.

[84] L. Dalcín, R. Paz, and M. Storti, “MPI for Python,” *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108 – 1115, 2005.

[85] E. Gabriel *et al.*, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” in *11th European PVM/MPI Users’ Group Meeting*, September 2004, pp. 97–104.

A
Implementation details

Q-NAS was developed in Python 3.6 [1]. The network assembling and training code made use of Tensorflow 1.9 [83] open source deep learning library.

Q-NAS runs in a multi-process environment, via MPI messages (with the help of MPI4Py library [84]). The master node runs Q-NAS and distributes the evaluation tasks to the slaves with non-blocking send operations. It also collects the results with non-blocking receives. Each worker (including the master) evaluates one individual per generation, that is, the total number of processes is equal to the number of classical individuals to be evaluated.

The experiments were executed in a multi-computer environment, which contains NVIDIA K80 [2] GPUs and Power8 processors running Linux (Red Hat Enterprise Linux 7.4 [3]). OpenMPI [85] 3.1.1 distribution was installed.

For the retraining phase, however, we run the jobs on a single machine from the environment described above. We use 1 GPU and 1 CPU in the single machine.

[1] https://docs.python.org/3.6/
[2] https://www.nvidia.com/en-gb/data-center/tesla-k80/
[3] https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux