Pós-Graduação em Ciência da Computação

Pedro Henrique Dreyer Leuchtenberg

Time Aware Sigmoid Optimization: A New Learning Rate Scheduling Method

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

Pedro Henrique Dreyer Leuchtenberg

Time Aware Sigmoid Optimization: A New Learning Rate Scheduling Method

A M.Sc. Dissertation presented to the Centro de Informáticaof Universidade Federal de Pernambuco in partial fulfillmentof the requirements for the degree of Master of Science inComputer Science.

Concentration Area: Computational Intelligence
Advisor: Cleber Zanchettin
Co-Advisor: David Macêdo

Recife
2019

Catalogação na fonte
Bibliotecária Mariana de Souza Alves CRB4-2105

L652t Leuchtenberg, Pedro Henrique Dreyer.
Time Aware Sigmoid Optimization: a new learning rate scheduling method/ Pedro Henrique Dreyer Leuchtenberg. – 2019.
61 f.: il., fig.

Orientador: Cleber Zanchettin.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação. Recife, 2019.
Inclui referências.

1. Inteligência computacional. 2. Aprendizagem de máquinas. 3. Redes neurais profundas. 4. Taxa de aprendizado. I. Zanchettin, Cleber (orientador). II. Título.

006.31 CDD (22. ed.) UFPE-CCEN 2020-149

Pedro Henrique Dreyer Leuchtenberg

“Time Aware Sigmoid Optimization: A New Learning Rate Scheduling Method”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 6 de setembro de 2019.

BANCA EXAMINADORA

_____

Prof. Dr. Adriano Lorena Inácio Oliveira
Centro de Informática/UFPE

_____

Profa. Dra. Aida Araújo Ferreira
Instituto Federal de Pernambuco/Campus Recife

_____

Prof. Dr. Cleber Zanchettin
Centro de Informática / UFPE
(**Orientador**)

ACKNOWLEDGEMENTS

To my mother, who always supported me in every single aspect. Your constant kindness and goodwill gave me the strength to finish this Masters Degree.

To Cleber, my advisor, who was always available. Thanks for not giving up on me; you deserved a better student.

To David, my co-advisor, thanks for your enthusiasm. No matter the results, you were always interested in discussing it and coming up with new ideas.

To my grandmother, sister, co-workers in GPRT, and all other people I had daily contact. Thanks for all the help during the last two and a half years.

ABSTRACT

The correct choice of hyperparameters for the training of a deep neural network is a critical step to achieve a good result. Good hyperparameters would give rise to faster training and a lower error rate, while bad choices could make the network not even converge, rendering the whole training process useless. Among all the existing hyperparameters, perhaps the one with the greatest importance is the learning rate, which controls how the weights of a neural network are going to change at each interaction. In that context, by analyzing some theoretical findings in the area of information theory and topology of the loss function in deep learning, the author was able to come up with a new training rate decay method called Training Aware Sigmoid Optimization (TASO), which proposes a dual-phase during training. The proposed method aims to improve training, achieving a better inference performance in a reduced amount of time. A series of tests were done to evaluate this hypothesis, comparing TASO with different training methods such as Adam, ADAGrad, RMSProp, and SGD. Results obtained on three datasets (MNIST, CIFAR10, and CIFAR100) and with three different architectures (Lenet, VGG, and RESNET) have shown that TASO presents, in fact, an overall better performance than the other evaluated methods.

Keywords: Machine learning. Deep neural networks. Learning rate.

RESUMO

A correta escolha dos hiper-parâmetros para o treinamento de uma rede neural profunda é um passo essencial para obter um bom resultado. Bons hiper-parâmetros vão levar a um treinamento rápido e a uma menor taxa de erro, enquanto que escolhas ruins podem fazer a rede não convergir, inutilizando todo o processo de treinamento. Dentre todos os hiper-parâmetros existentes, talvez o mais crítico seja a taxa de aprendizagem, que irá controlar a magnitude com qual os pesos da rede neural irá atualizar em cada interação. Nesse contexto, esse trabalho avaliou um novo método de mudança na taxa de aprendizagem denominado Training Aware Sigmoid Optimization(TASO), que propõe uma fase dupla de treinamento. O método proposto tem como objetivo melhorar o treinamento, obtendo uma melhor inferência em um menor tempo decorrido. Uma série de testes foi feitas de forma a validar essa hipótese, Comparando TASO com outros métodos de treinamento mais comuns como Adam, ADAGrad, RMSProp, e SGD. Resultados Obtidos em três datasets (MNITS, CIFAR10, e CIFAR100) e três diferentes arquiteturas (Lenet, VGG, e RESNET) mostraram que TASO apresenta uma melhor performance do que os outros métodos avaliados.

Palavras-chaves: Aprendizagem de máquinas. Redes neurais profundas. Taxa de aprendizado.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CPU	Central Processing Unit
MLP	Multi-Layer Perceptron
MLP	Graphics Processing Unit
NLP	Natural Language Processing
TASO	Training Aware Sigmoid Optimization

LIST OF SYMBOLS

$\nabla_x y$	Gradient of y with respect to x
$A \odot B$	Element-wise product of A and B
$L(x; \theta)$	Loss function of x samples with θ parameters
$x^{(i)}$	The i-th example from the dataset
$y^{(i)}$	The target associated with $x^{(i)}$

CONTENTS

1 INTRODUCTION

Humans like to automate things. We can probably describe any major invention and technological advancement as basically as a way of automating something.

- Electric light? An "automation" of the process of creating light using fire.
- Telephones? An "automation" of the process of communication across long distances.
- Automobiles? An "automation" of the process of locomotion.

We may argue that one of the significant differences between the human race now and from ten thousand years ago is how good we became in "automate" the different necessities of our basic needs. However few things can be done today that could not be achieved by humans with primitive technology. Of course, the amount of effort in some cases would be gigantic, but possible nevertheless. Just look at the Egyptians pyramids, to see what can be done using only very basic technology but quasi-infinite free human labor.

So, while in general terms automating things do not necessarily increase the boundaries of human achievements. In practical terms, this could not be further from the truth. Automation makes several activities more accessible, faster, and safer, and sometimes, the change brought forth is so massive that it can completely modify how human society works. In reality, the automation of human activities is the driving force behind the most considerable changes in the human way of life since pre-historic times. In the Neolithic revolution, human society started to live in large settlements thanks to the advancements of agriculture and animal husbandry, which can be thought of as automation of the activities of hunting and gathering. Following that, we have the industrial revolution, which automated human labor through the use of the steam engine. Even now, we are undertaking a new revolution originated from the developments in automation — namely, the advent of the electronic computer and how it can automate mental labor.

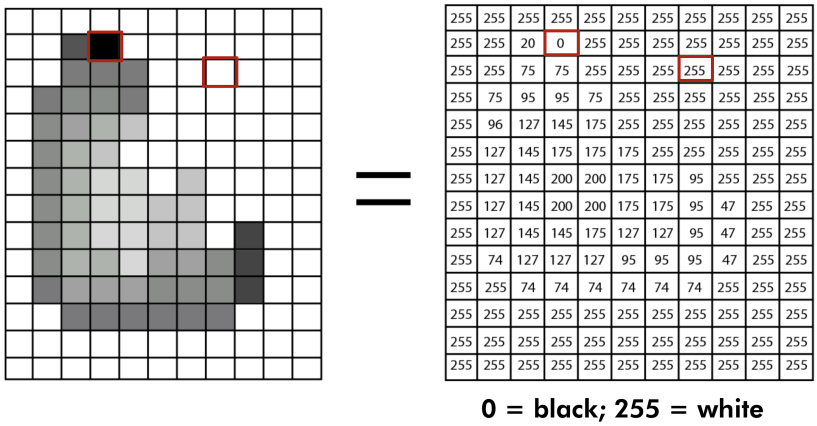
While the title of the first computer can be a topic of discussion (Does electromechanical computers count? Does it need to be programmable? What about Turing Complete?) the undeniable fact is that its advent made possible the automation of several high complex calculation that would take several human counterparts with significant mathematical knowledge a considerate amount of time. One of the classic examples of this era is the use of computers in the calculation of torpedoes trajectory in moving targets and cryptography analysis of ciphered messages during the second world war.

The following years were accompanied by a great interest in the scientific community on the new field of Artificial Intelligence (AI) (NEWELL; SIMON, 1956), (TURING, 1950), which consists on the study on how to implement human-like reasoning in a computer. It then became clear that computers and humans have a very different concept of complexity.

Remember the calculations made by the first computers mentioned above? While they seemed complex to us, they can be easily described in a series of simple mathematical equations and formal rules, which computers can readily calculate with high speed and perfect repeatability. However, "simple" tasks done every day by humans independent of specific domain knowledge, such as understanding spoken words and recognizing familiar faces, were incredibly tricky for computers until very recently. Let us take a look at the problem of image classification and the challenges one would have to tackle to create a hard-coded algorithm to get some insight on the matter.

Computers interpret images as depicted in Figure 1. For them, images are nothing more than a list of numbers representing luminosity intensity. The job of an image classification algorithm is to, by using those values, output what exactly is being depicted.

Figure 1 – Representation of how an image is interpreted by a computer.



Source:<http://edtech.engineering.utoronto.ca/files/2d-image-digital-representation>

Imagine how you would describe a cat to someone that never had seen such an animal before. You could say that it is an animal with four legs, long tail, covered with soft fur, and around half a meter in size. Seems pretty good, right? Now notice how many of those characteristics could also apply to a dog. Practically all of them. So we need to be more specific. We could mention that cats have long whiskers on their nose, that their head has a more rounded shape, or that its legs are longer, and so on. It is hardly a simple task. You could spend a whole day trying to explain what a cat looks like, and the other person could still not be 100% confident of detecting a cat when seeing one. However, once this person has seen a cat, subsequent identifications are going to be much easier. The person has "learned", via real-world examples, what a cat looks like.

Now imagine doing all this, but instead of a person, you need to come up with a cat identifier computer algorithm. Different than a human, a computer had no formal definition of softness, roundness of any other characteristics. They do not even comprehend underlying physical phenomena like changes of perceived size with distance, brightness, shadows, and occlusion, to name a few. The only thing the computer has is the image

representation with its pixels values, as shown in Figure 1. This is not a simple task. In fact, until the last decade, robust image classification similar to human levels was unheard of.

The problem was already mentioned. Computers only have the pixel values of the image but have no other information. Somehow is up to the computer's programmer job to codify all those real-world constructs in a virtual machine that only interprets 0's and 1's. However, how one describes soft fur to a computer? Long whiskers? Or even occlusion and shadow formation? There is not a definitive answer. For more than 40 years, researchers try to come up with handcrafted features for doing image classification with little success. Nevertheless, what if there is another way to build a cat classifier? Could not computers learn by example, like humans, and thus making the job easier?

Machine Learning is a subset of AI that tries to come up with solutions by presenting the computer with examples instead of relying on hard-coded solutions. Ideally, a machine learning algorithm would receive a series of cat images, and "learn" how to identify cats in subsequent images. While a fantastic principle, classical Machine Learning was not the solution to every problem. It worked well with tabular data. On the other hand, it struggled to deal with most problems that involved sensory input, such as computer vision, natural language processing, and speech recognition. The thing was that the raw input data was in a format too hard to be interpreted by the computer.

We were still dependent on creating handcrafted features. The difference now was that machine learning algorithms could "learn" how much of its feature each class had. For example, it could learn that cats have 0.8 soft fur and 0.9 head-roundness while dogs are 0.3 soft fur and 0.4 head-roundness.

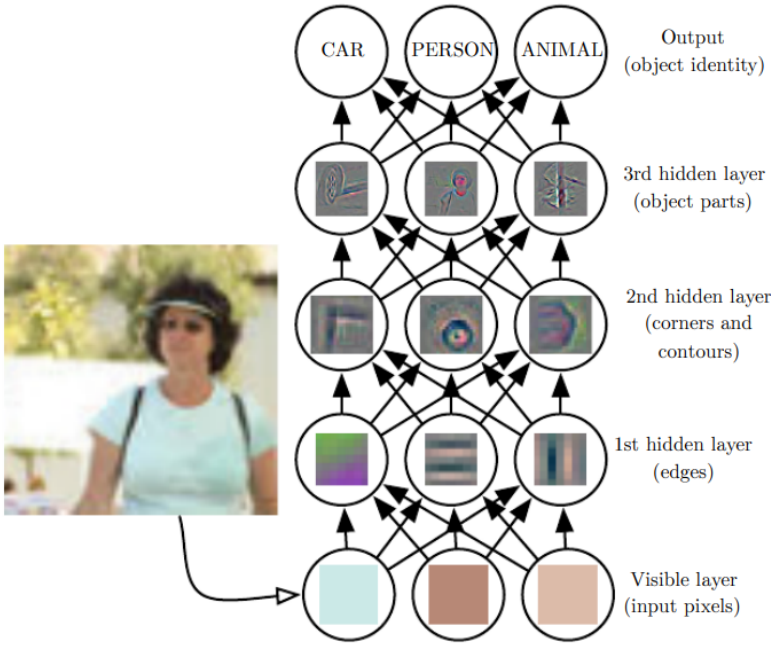
For many years feature engineering could be considered the principal bottleneck in the field of machine learning. Most of the more successful models relied heavily on handcrafted features to archive a good performance (GOODFELLOW; BENGIO; COURVILLE, 2016).

A possible solution for this problem is to have the machine learning algorithm itself trying to come up with the best features possible, a strategy called representation learning. While being a theoretically sound idea and being used in several areas of research, such as data compression and image classification, representation learning strategies such as dictionary learning were not as revolutionary as hoped.

It was only with the development of deep learning approaches that computers could reliably automate the creation of consistently high-quality representation and achieve inference performance close, or in most cases, higher than the features developed by human domain experts. Figure 2 shows how a deep learning architecture creates hierarchical features, increasing in complexity on each successive layer.

Nowadays, deep learning is being used in many different fields with amazing results. Especially in areas that classical machine learning struggled historically. Image classification and segmentation (BULÒ; PORZI; KONTSCHIEDER, 2017), speech recognition (SAON

Figure 2 – Illustration on how a deep learning architecture receives a raw of pixel inputs and construct higher-level concepts such as gradient, edges and objects parts on each layer



Source:(GOODFELLOW; BENGIO; COURVILLE, 2016)

et al., 2017) and Natural Language Processing (NLP) (DEVLIN et al., 2018) are all areas where deep learning is state-of-the-art. However, while the results are auspicious, deep learning has its obstacles. While the problem of feature engineering may be in the past, we come up with new challenges, mainly the fields of architecture engineering and training optimization. Continued research and development are necessary to push further our understanding of these topics, which hopefully will bring even more promising results.

Deep learning networks are composed of several parameters called weights, which will, given the network input, produce a particular output. In other words, it is the weights that will determine if a given architecture is going to identify, for example, cats, dogs, or any other thing. Initially, the weights start in a random state, without much practical use. However, by showing new examples to the network, we can tweak the values of the weights in a process called "training". In simple terms, the training process is the following one: For each input presented to the network, it will give an output. In supervised learning cases, this output is then going to be compared to the expected result. The difference between them is used to update the weights' values in a manner that if the same input is presented again, it will give an output closer to the expected value.

The training algorithm controls the details of how each weight parameter is going to be updated. Similar to different algorithms in computer science, such as sorting or graph search algorithms, there are multiple flavors of training algorithms. Each kind of

training algorithm has its particularities like aiming to be faster, easier to implement, or promote a better inference performance. Most of the training algorithms have some hyperparameters which will tune how training is going to proceed and, different from the network parameters, are not changed throughout training but need to be set beforehand. The selection of proper hyperparameters values is not a deterministic process being usually guided by heuristics or past experiences.

A further particularity with all training algorithms is a hyperparameter called the learning rate, which dictates how fast the values of the weights are changing during the training process. While it seems to be a pretty simple issue, the correct tuning of the initial learning rate and how it is going to vary thought the training is a very complex problem. It is then, in the field of training optimization that this work tries to contribute.

Most training algorithms use a fixed training rate or vary it using a simple monotonic decaying function. In that sense, a relevant question to ask would be if there was not any better way to choose or decay the learning rate; one, not as simple as the already used methods, but which at the cost of some complexity improve the inference performance of the training process of a Deep Neural Network.

Following this line of thought and after some theoretical research and practical experiments, it was hypothesized that by using a two-phase training, where each training phase would focus on different optimization goals, we could see some improvement. Those studies and tests culminated with the proposal of a new learning rate decay method, created by the author, called Training Aware Sigmoid Optimization (TASO). Moreover, as the name implies, the learning rate is going to be varied following general sigmoid function, used here to archive a smooth transition between the two distinct learning rates used.

1.1 OBJECTIVES

This work has as principal objective to evaluate a novel learning rate decay method Called TASO, which aims to improve the overall training performance during deep learning tasks. We evaluate the method comparing its results to more commonly used learning algorithms, e.g., Adam, Rmsprop, and Adagrad.

A secondary objective is to analyze the process of hyperparameter selection in this new method and evaluate different possible heuristic to guide future uses.

1.2 DOCUMENT STRUCTURE

This Document is structured in the following manner:

- In Chapter 2, we have the **Theoretical Background**, which presents concepts used throughout the document, detailing the deep learning algorithm and its nuances. There is also an explanation on topics such as architectures, activation and loss functions, regularization, and training algorithms.

- Chapter 3 explain the **Proposed Method**, TASO, a type of learning rate schedule that complements the SGD algorithm. It contains a mathematical definition of the model, as well as an analysis of its hyperparameters and how to better select them.
- In Chapter 4, denominated **Experiments**, we have the reasoning behind the selection of databases and architectures and overall experiment design followed by the results and discussion of the tests.
- Finally, in Chapter 5, there is the **Conclusion** where we discuss the overall results and possible future works.

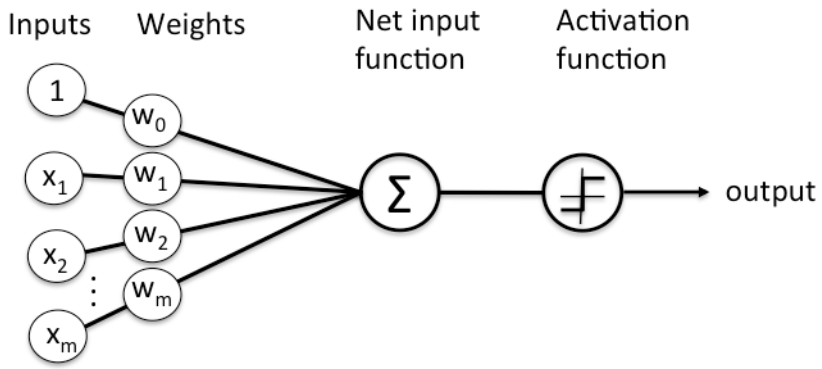
2 THEORETICAL BACKGROUND

2.1 NEURAL NETWORKS

While deep learning research started to get traction in the mid 2000s with the development of (HINTON; OSINDERO; TEH, 2006) and (BENGIO et al., 2006). Much of its theoretical basis comes from the field of Artificial Neural Network (ANN). ANN was developed during the second half of the last century, and as its name implies, takes great inspiration from the design of the human brain. The main idea behind ANN is that we have a series of units, which are interconnected between themselves, and depending on the input received will output a particular output. The ANN model tries to learn the mapping relation from the input to the output based on statistical patterns of the input data.

In Figure 3, we can see the classical representation of one of the most used units called, a perceptron, and some of its keys features, such inputs, weights, and activation function. It is easy to see the parallel between this and the human brain, where the perception is a single neuron, the weights represent the synapses and its strength, and the activation function simulates the behavior of biological neurons.

Figure 3 – Classical Perceptron's representation

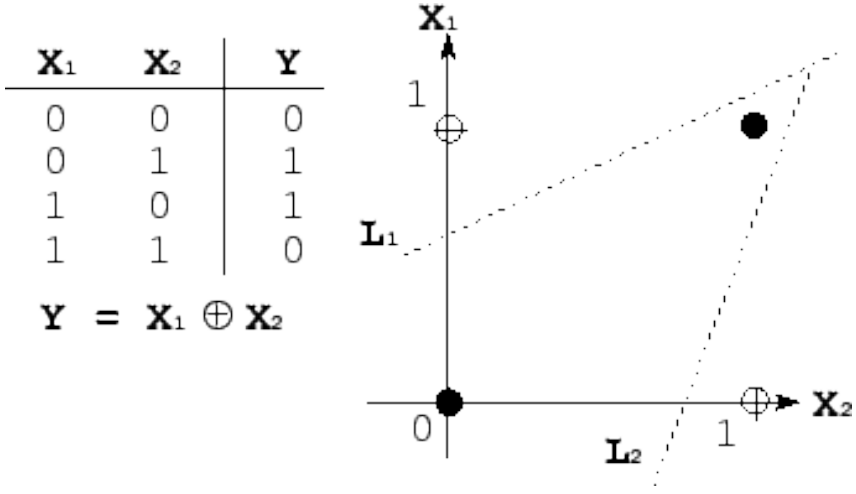


Source:<https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html>

While biological inspiration played a very important role during the earliest of ANN's research, it became clear that it was not enough to propel it forward. More specifically, (MINSKY; PAPERT, 1969) proved that ANN's designs used at the time could not come up with a correct generalization of the XOR problem, as shown in Figure 4. This single result was enough to practically put a halt in ANN research for more than a decade. It was only with the advent of backpropagation (LINNAINMAA, 1976) and its overall recognition as a useful tool with the work of (RUMELHART; HINTON; WILLIAMS, 1986) that the interest of ANN's started to pick up again.

The most important consequence of the development of backpropagation was that it made it possible to train a Multi-Layer Perceptron (MLP) networks. Different than the

Figure 4 – Visualization of the XOR problem. A perceptron is able to divide the input space using a straight line. However, there is no straight line that correctly divides the two different classes.



Source:<http://www.ece.utep.edu/research/webfuzzy/>

previous design, which has only an input and output layer, those new kinds of networks could have any number of arbitrary "hidden" layers. The results of employing more layers were immediate. Not only the new networks could solve the XOR problem, but it could also, theoretically, solve any other possible problem. This result is known as the Universal approximation theorem (CYBENKO, 1989), which in formal terms state that a multi-layer perceptron network with at least one hidden layer and a finite number of neurons can approximate any continuous function to any degree of precision.

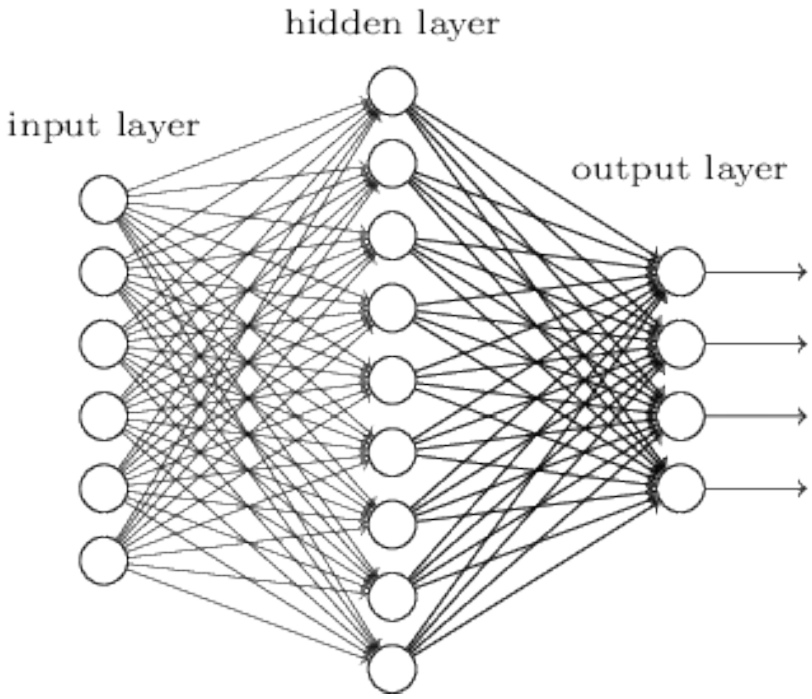
However, while backpropagation was an essential tool for ANN's, it not biological feasible (HUNSBERGER; ERIC, 2018). Meaning it cannot be considered a model on how our brains work. While for some researches (mainly in the area of computational neuroscience which studies how the brain works in an algorithmic level), this meant that ANN's were becoming not so much an interesting line of work for most of the scientific community it was a very welcomed addition. From this time on, biological inspirations and feasibility became more an afterthought than actually guiding principles.

From the development of backpropagation in the 80's, neural networks got quite a lot of traction as being one of the most researched topics in Machine Learning. But by the early 2000's it was clear that while multi-layer perceptrons were far from being able to achieve the kinds of results that the Universal Approximation Theorem stated. The problem was that, while a multi-layer neural network could approximate any arbitrary function most of the time, it was not possible to train it to reach the desired result. Let us look at the problem of image classification using classical multi-layer perceptrons to understand the issue better.

In the classic architecture of MLP's, every subsequent layer is fully connected to the previous one, as shown in Figure 5. In this Figure, we have six inputs, nine hidden

units, and four outputs, thus giving us 90 weight parameters. Imagine now dealing with a 1,000x1,000 image and a hidden layer with one million neurons. Suddenly, we have 10^{12} parameters that need to be adjusted the represent the input/output mapping. Considering that a 16-byte float point number represents each weight, we would need 16 terabytes only to hold its values. Even if space were not a problem, such a massive network would need a massive amount of data to be appropriately trained and to avoid underfitting.

Figure 5 – MLP fully connected structure.



Source: <https://bit.ly/2ONi52S>

One way to mitigate this problem is to use multiple hidden layers instead of one. Deeper architectures are intrinsically more efficient than shallow ones (PASCANU; MONTUFAR; BENGIO, 2013), so we would need less hidden units and consequently fewer parameters. Deep architectures would also be preferred since they could theoretically build up knowledge in a hierarchical fashion, as shown in Figure 2. Unfortunately, MLP models with multiple hidden layers suffer a problem called vanishing gradients (KREMER, 2001), where the backpropagation algorithm stops doing significant updates to the weight for each subsequent layer away from the output. This problem effectively stops the training process.

Those problems made MLP's very limited on the kind of problem they could solve. It functioned as a novel machine learning model in the 80's, but one that appeared to face similar issues as other machine learning methods (still depending on feature engineering, for example). By the early 2000's the MLP approach was already being surpassed by more newer developments such as Support Vector Machines (CORTES; VAPNIK, 1995) and random forests (BREIMAN, 2001).

Note that while there were different types of architectures of ANN's the rest of this work will focus on the ones presented so far, namely the MLP (also called feedforward networks) and its deep learning equivalents.

2.2 GENERAL WORKING PRINCIPLES

In general terms a neural network is a model that try to approximate an arbitrary function of the format $y = f(x)$ by an approximate function $y' = f'(x, w)$, with $y \approx y'$. On those equations, x represents the inputs, and w is the network parameters (also called weights and bias). In the beginning, the parameters are initialized from a random distribution, which makes f' pretty different than f. The job of the network is to update its parameters using as a basis the available problem information data.

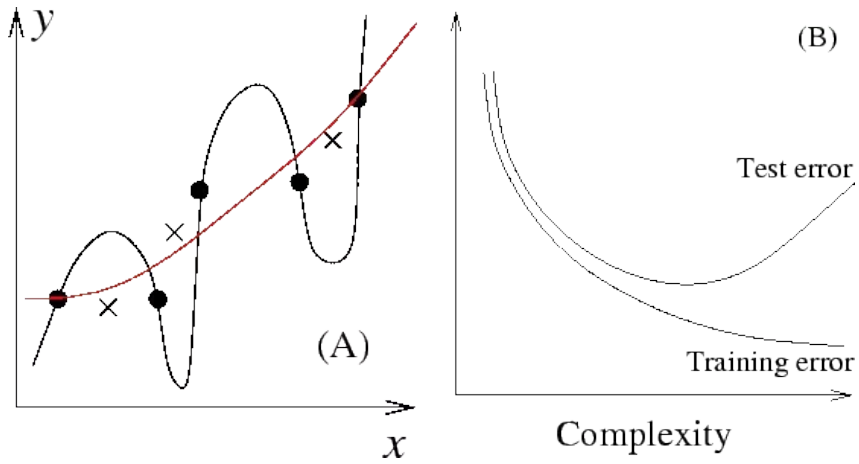
Every time a data sample is presented to the network, it will output a value. The network compares its output with the correct output using something called the loss function, which will quantify how 'wrong' our network output is from the desired one. There are different types of loss functions for different types of problems. For classification, the most used is the cross-entropy loss. The network will then calculate the gradient of the loss function concerning the weights of the previous layer. Thus, giving a general magnitude and direction that each weight must change to minimize the loss function. Finally, the backpropagation algorithm is used to calculate the gradient of the loss function to each subsequent layer. The weights are then updated, multiplying the value of the gradient by a scalar named the learning rate, which will dictate how fast the parameters are going to change. Once all weights are updated, one full interaction of the learning algorithm is done. Usually, thousands of those interactions are going to be needed to train a network properly.

While the minimization of the loss function is the primary goal of the network, it cannot follow this goal blindly. Most machine learning models could, if tweaked to do so, have an output that would perfectly match the desired output of the training data. However, as we are going to see, the resulting model would probably fare rather poorly when presented with new data that was not used during training. Just imagine a case where we would like to develop a model that would determine the volume of a cube given the length of one of its sides [1]. We could measure the cube side with a ruler and its volume with water, for example. Since we are getting real-world measures, there is bound to be some measurement error. So, for example, instead of measuring a side of $2cm$ and a volume of $2cm^3$ we could obtain a side of $2.1cm$ and a volume of $1.95cm^3$. An unrestricted neural network would become a direct mapping between the training data inputs and the desired output. Instead of "learning" that the volume of a cube is equal it is side length to the cube, it would come up with a much rather complex function, unrelated to the

[1] While this is an elementary problem which we know how to solve analytically, it is useful as a toy example

real world, but which would give perfect results on the training data. Figure 6 gives a visualization of what it would look like.

Figure 6 – Example of overfitting. The black dots are the measured training examples, the red line the real function and the black line the learned function given by the neural network.



Source: <https://puntomedionoticias.info/album/overfitting-large-dataset.htm>

In Figure 6 the available data points are the dots, the red line represents the real function, and the black is the function outputted by the network. Note that the black line passes perfectly across all the data points. On the other hand, it behaves erratically in every other place. Now, instead of training the network with all the data available, what if we divide it beforehand and only use part of it for training? In this case, we could use the remaining data points only to test our network, and see how well it is generalizing new inputs. Those two datasets are an integrating part of any machine learning algorithm and are known as training and test datasets. The objective of machine learning is then to minimize both the error of the test and training sets. When both errors are high, the network is underfitting, while when the training error is low, and the test error is high, the network is in overfitting.

While we have seen that it is not desirable to either underfit or overfit, and it is possible to verify that one or the other is happening by comparing the training the test error, what can be done to reduce its occurrence? As depicted in Figure 6, overfitting and underfitting are correlated to the complexity of the machine learning model. Simple models tend to underfit, while overcomplex models tend to overfit. In the case of neural networks, the model complexity is related to its architecture, weight values, and the number of training interactions. So, normally, to prevent underfitting, we need to pick a capable enough architecture (in the XOR case, one that has at least one hidden layer) and to train it long enough. The training algorithm is going to update the weights until our training and test errors are small. If training continues for more interactions, the test error is going to stop decreasing or even increase. This means that further training is not going to improve the model. In that sense, stopping the training at the right time can both reduce overfitting

and underfitting. Another strategy used to diminish overfitting is called regularization, which is a method to control further restrain model complexity.

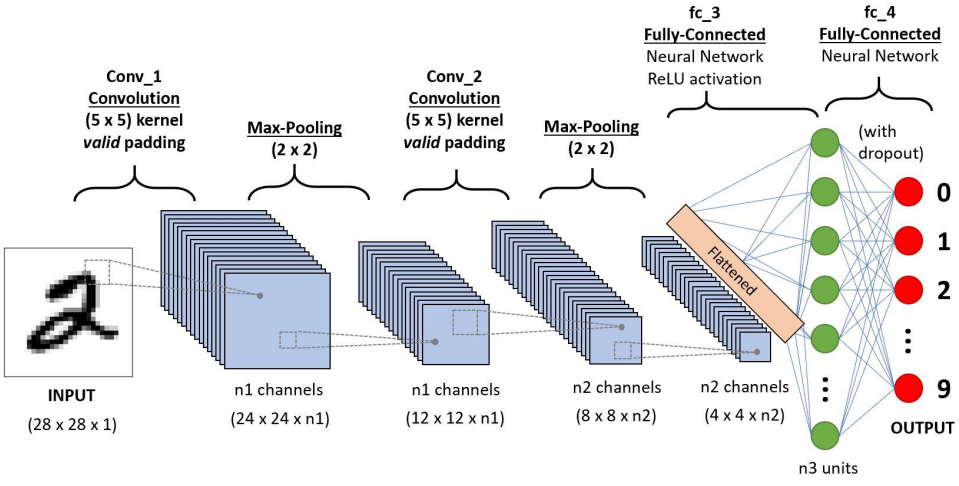
2.3 DEEP LEARNING APPROACHES

As it was mentioned in the previous section, Neural Networks based on the classical MLP had some shortcomings. However, a series of improvements, mainly in the areas of activation functions, architecture design, and regularization, allowed to improve the results by a significant degree, achieving results unmatched so far in the field of machine learning. Since those changes made possible the use of deep (more than one hidden layer) architectures, this new phase on ANN's research is called "Deep Learning".

2.3.1 Architectures

The fully connected architecture has a severe flaw of not scaling well with very high dimensional inputs such images. This was not the case with architectures, such as Convolutional Neural Network (CNN). In Figure 7, we can see the principal components of a CNN, which are its convolutional and pooling layers. The convolutional layers, as the name implies, perform a convolution operation with the input image. The result is an image roughly the same size but with some of its features highlighted. After that, it passes by the pooling layer, which will reduce the dimensionality of the image. The combined work of several of those layers is going produces hierarchical features exactly as the ones presented in Figure 2. Finally, in the last layer, we flatten the resulting image in a one-dimensional array and use a fully connected layer, much like the ones used in a classical MLP.

Figure 7 – Convolutional Neural Network model.

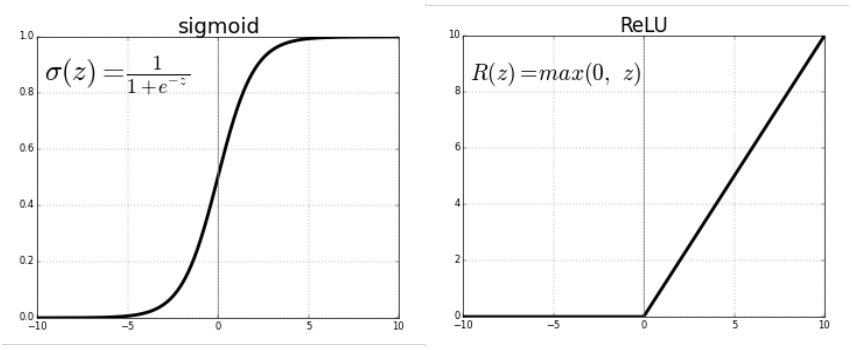


Source: <https://zhuanlan.zhihu.com/p/58882714>

2.3.2 Activation Functions

The usual activation functions used in MLP's were a logistic function of the format $\frac{1}{1+e^{-x}}$ which general shape seen in Figure 8 . They had nice mathematical properties such as being continuously derivable, having bottom and upper thresholds, and also having its derivative be an expression of itself. However, its derivative would tend to zero near its saturation values, making training slow sometimes and thus creating the vanishing gradient problem. Its non-symmetry around zero was known to raise some issues that would slow down training, as pointed out by (MONTAVON; ORR; MÜLLER, 2012) and (GLOROT; BENGIO, 2010). To avoid the symmetry problem, we can scale the sigmoid function around zero, given origin to the *tanh* function.

Figure 8 – Sigmoid and ReLU activation functions.



Source: <https://mlblr.com/includes/mlai/index.html>

Nowadays, most practical uses of Deep Learning uses the ReLU function (JARRETT et al., 2009), shown in Figure 8 and which have activation function equal to $max\{0, z\}$. As can be seen, those units have two regions of linear behavior: when the internal state of the neuron is negative, the output is zero, and when the internal state is positive, the output is the state itself. This brings a few interesting properties. First of all, its derivative is always equal to one when the unit is active, meaning we are always going to have some amount of training, solving the vanishing gradient problems. Second, its behavior is mostly linear, meaning that it should be easier to optimize (GOODFELLOW; BENGIO; COURVILLE, 2016). While the ReLU units still have some properties not wanted in the activation function, such not being symmetric around zero, much of the deep learning understanding come from general heuristics and tests results, which seems to indicate, so far, that the usage of a ReLU activation function improves training considerably.

2.3.3 Regularization

A central problem in machine learning is how to develop an algorithm that will not only perform well in the training data but as well in new data points. The set of strategies developed to reduce this test errors, sometimes at the expense of the training error is known as regularization strategies.

The most common regularization used in MLP's is called weight decay, and it consists of modifying the loss function to add a term dependent to the root square sum of the weights. In other words, if several of the weights are non-zero, our loss function is going to increase. This change makes the network try to compromise a small training error while maintaining most of the errors close to zero and thus limiting the complexity of the model.

Beyond weight decay, other samples of regularization methods are dataset augmentation, early stopping, and addition of noise (GOODFELLOW; BENGIO; COURVILLE, 2016), which were being used long before then the development of deep learning.

A new type of regularization, developed exclusively for deep learning architectures, is called dropout (SRIVASTAVA et al., 2014). In this method, for each training epoch, the network is going to be presented with a subset of its training samples, and each connection between neurons has a probability of being removed. Once training is done, the full network is used with its weights being scaled according to its probability of being removed. In other words, we are training in the same architecture many different networks in a subset of data, much like the assembly method called bagging (GOODFELLOW; BENGIO; COURVILLE, 2016). However, different from bagging, where all networks are independent, in dropout, the sub-networks all come from a single initial network and have its parameters shared. This makes it possible to train an exponentially higher number of sub-networks in a fraction of the time and space required for ensemble methods.

The sharing of parameters between the sub-networks also forces different combinations of hidden units to work equally well, which further increases the generalization capability of the overall network. This makes dropout not only a more efficient bagging approximation algorithm but also having a much more powerful regularization effect.

2.3.4 Data Normalization

A common practice used to avoid numerical instabilities in most machine learning models is to normalize the input variables. This process homogenizes the data, ensuring that each variable is going to have a zero mean and unity standard deviation. The same concept could be done for each layer of a deep network where the activation values are normalized across a minibatch of inputs. In other words, the output of each layer is going to have zero mean and unity variance. This method is called batch normalization (IOFFE; SZEGEDY, 2015), and while simple, in theory, it was responsible for improving training times of deep learning architectures significantly. This strategy can also be seen as a way of network regularization.

2.3.5 Other improvements

Trying to balance underfitting, overfitting, vanishing gradients, hyperparameters, and handcrafted features, to name a few, made the training very laborious from a practical point of view. The problem was so common that a popular concept was that the training of

a neural network was more an art than a science. However, the need for handcrafting and hyperparameter tuning is inverse proportional to the amount of data available. In other words, the higher the amount of data, the better a deep learning algorithm is going to perform. In that sense, Deep learning models need huge datasets to be adequately trained. However, for much of machine learning history, such datasets did not exist. Furthermore, even if they existed, to train them would take an incredible amount of time. It was only during the last decade with the creation of larger datasets such as Imagenet (DENG et al., 2009), that training deep learning models to achieve human-level performance became a possibility.

Another massive development was the use of Graphics Processing Unit (MLP) for training Deep Networks. Different than Central Processing Unit (CPU), which are designed to be able to do several different computation routines and more fitted to implement most machine learning algorithms, GPU is projected to do multiplications in a highly paralyzed way. Since most deep learning algorithms are nothing more than a vast series of multiplications, using GPU's to them made the process significantly more fast (20 times as fast in some cases). While one can argue that this reducing in training time was not strictly necessary to enable any of the results observed so far, I think it is not the case. Just imagine, instead of taking three weeks to train the winning submission for the Imagenet 2014 challenge, it took one year? Another important consideration is that Deep learning research, while backed up with theoretical formulation, is a field where empirical results are responsible for great leaps in performance. The capacity for testing more assumptions in less time is responsible much of the high speed of development in the last few years.

2.4 TRAINING ALGORITHMS

Training algorithms are probably the most crucial part of any Machine Learning model. They will directly control how the parameters of the model are going to adjusted. It is them no surprise that when ill applied, they can make even the state of the art deep learning model, with a correct chosen architecture, plenty of training samples, balanced classes, non-noisy data, in other words, a data scientist dream; utterly fail.

While most of the other components of a deep learning system, such as architecture, activation function, and loss function, have an evident selecting process, the same could not be said for training algorithms. The thing that differentiates them from the other components is that training algorithms have something called hyperparameters, which will dictate how they conduct the training process. Perhaps the most essential hyperparameters of all, and shared between every learning algorithm, is the learning rate, which controls the magnitude of how each parameter changes on every interaction.

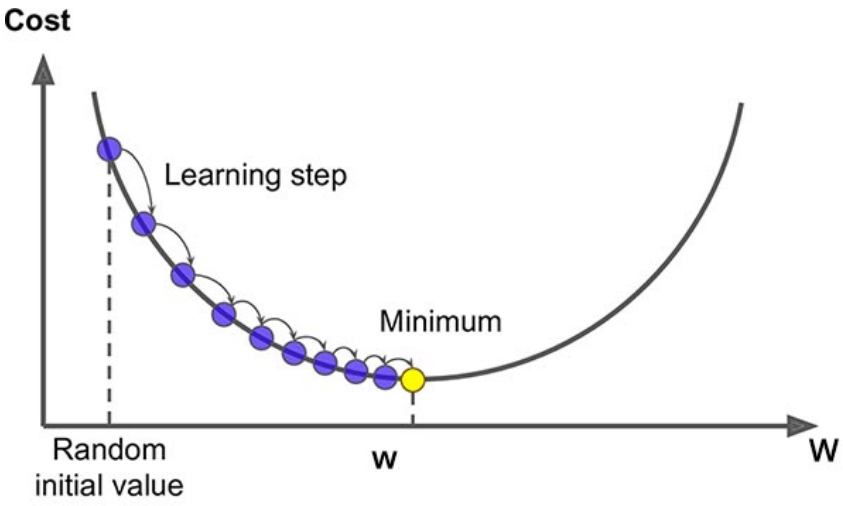
The main issue is that depending on the peculiarities of the training data and architecture, we have a different set of hyperparameters that would optimally train the network.

While some heuristics generally guide its selection, its usefulness is limited, and most of the time, some experiments are necessary to make a choice correctly. The following subsections will describe some of the most traditional training algorithms.

2.4.1 Vannila Gradient Descent

One of the oldest optimization techniques is the so-called Gradient Descent (CAUCHY, 1847). It finds the minimum of a function using an interactive process where at each interaction, the gradient of the function is evaluated, and a small step is taken in that direction. The visualization of such a process is presented in Figure 9

Figure 9 – Example of how a gradient descent algorithm would find a minimum of a function.



Source: <https://towardsdatascience.com/an-introduction-to-logistic-regression>

2.4.2 Stochastic Gradient Descent

The Stochastic Gradient Descent (SGD) is a form of gradient descent very similar to the vanilla Gradient Descent described above. The main difference being it uses a subset of the training samples, called mini-batches, instead of the whole training set. A formal definition of SGD can be seen in Algorithm 1.

The standard error of a given estimate varies according $\frac{\sigma}{\sqrt{n}}$ (JAMES et al., 2014), where σ is the standard deviation of the sample values and n is the number of samples. While, according to the above equation, the larger the batch size, the more accurate is the estimate of the gradient, the returns are not linear as we increase the batch size. So, for example, if we want to have an estimator ten times more precise, we need to use 100 times more data points. Even if smaller returns with larger batch size was not enough to justify not using the whole training set, most learning algorithms converge faster if they update more frequently, even if the estimate of the gradient is a bit off. The usage of small

Algoritmo 1: SGD algorithm

Require: Learning rate ϵ
Require: Initial parameters θ
while *stopping criterion not met* **do**
  Sample a minibatch of n examples from the training set $\{x^{(1)}, \ldots, x^{(n)}\}$ with corresponding targets $y^{(i)}$.
  Compute gradient estimate: $g \leftarrow +\frac{1}{n}\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  Apply update: $\theta \leftarrow \theta - \epsilon g$

mini-batches also seems to have a regularization effect, as shown in (WILSON; MARTINEZ, 2003).

For its simple implementation, and excellent properties, enabling training massive deep learning data sets in smaller batches, the SGD algorithm is still one of the most used algorithms for training.

2.4.3 SGD with Momentum

While pure SGD is a sound strategy, it is typically used with a momentum term, first proposed by (POLYAK, 1964), which usually accelerate the learning on cases of small or noisy gradients, and high curvatures. The term momentum comes from the physical analogy of this algorithm in which the gradient can be considered as an external force acting on a particle traveling on the Loss function space. In more mathematical terms, we are now applying a moving exponential average on the gradient term, which makes the gradient term to take into consideration its past values. The formal definition of the SGD with momentum can be seen in Algorithm 2.

Algoritmo 2: SGD with momentum

Require: Learning rate ϵ, momentum parameter α
Require: Initial parameters θ, initial velocity v
while *stopping criterion not met* **do**
  Sample a minibatch of n examples from the training set $\{x^{(1)}, \ldots, x^{(n)}\}$ with corresponding targets $y^{(i)}$.
  Compute gradient estimate: $g \leftarrow +\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  Compute velocity update: $v \leftarrow \alpha v - \epsilon g$
  Apply update: $\theta \leftarrow \theta + v$

The momentum hyperparameter α is going to dictate how much the previous values of the gradient are going to affect the current direction with higher values making the previous values have a stronger influence. α will also determine the highest update value a step can archive; if the gradient has a constant direction and value equal to g, its highest terminal velocity is given by $\frac{\epsilon g}{1-\alpha}$.

Nesterov Momentum, a variant of the momentum algorithm, was developed by (SUTSKEVER et al., 2013). The formal definition of Nesterov momentum is given in Algorithm 3. The difference between Nesterov and the original variation is that the gradient is evaluated after the current velocity is applied. It was proved (GOODFELLOW; BENGIO; COURVILLE, 2016) that this modification increases the rate of convergence for a convex problem using the whole training set to calculate each gradient step. However, since deep learning training is a non-convex problem, and we usually are using mini-batches, its theoretical properties are not granted.

Algoritmo 3: SGD with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α
Require: Initial parameters θ, initial velocity v
while *stopping criterion not met* **do**
  Sample a minibatch of n examples from the training set $\{x^{(1)}, \ldots, x^{(n)}\}$ with corresponding targets $y^{(i)}$.
  Apply interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$
  Compute gradient (at interim point): $g \leftarrow +\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  Compute velocity update: $v \leftarrow \alpha v - \epsilon g$
  Apply update: $\theta \leftarrow \theta + v$

2.4.4 Adaptive Learning Methods

CChoosing the correct learning rate can have a very profound effect on the speed and quality of training. Trying to reduce this effect, there were a series of adaptive learning methods created in the last few years, that try to calculate a particular learning rate for each set of parameters. The basic idea behind them is that the global learning rate is going to be scaled by each parameter, past gradient absolute values. So, if a parameter has consistently big gradients, its learning rate is going to have a more significant decrease, while parameters with constantly small gradients will have a smaller reduction in comparison. The expected results for those kinds of algorithms is to transverse small sloped regions in the Loss function space more rapidly.

2.4.4.1 Adagrad

The Adagrad algorithm (DUCHI; HAZAN; SINGER, 2011) scales the global learning rate by the inverse square root of the sum of all squared values of the gradient. While having some good theoretical properties for the convex optimization case, AdaGrad does not perform so well for deep learning training tasks. One of the main issues seems to be that the accumulative term is a monotonic increasing function, meaning that it is always getting larger. This can lead to an excessive decrease in the learning rate during later parts of the training. The Adagrad Algorithm can be seen in Algorithm 4

Algoritmo 4: Adagrad algorithm

Require: Global learning rate ϵ
Require: Initial parameters θ
Require: Small constant δ, normally 10^{-7}, for numerical stability
while *stopping criterion not met* **do**
  Sample a minibatch of n examples from the training set $\{x^{(1)}, \ldots, x^{(n)}\}$ with corresponding targets $y^{(i)}$.
  Compute gradient estimate: $g \leftarrow +\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  Accumulate squared gradient: $r \leftarrow r + g \odot g$
  Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$
  Apply update: $\theta \leftarrow \theta + \Delta\theta$

2.4.4.2 Rmsprop

The RMSProp algorithm (TIELEMAN; HINTON, 2012) was proposed as a modification of AdaGrad by changing the accumulation of gradients into a weighted moving average, similar to the SGD Momentum algorithm. The idea behind this change is that ideally, we want to arrive in local minima, which could be seen as having a quasi-convex neighborhood. Since Adagrad accumulative term is always increasing by time, the algorithm would have arrived in a convex region; it could have passed for many non-convex ones, which would reduce the learning rate considerably and thus diminish its convergence capabilities. If we used an exponential mean, older gradients would have a smaller influence in the accumulative term. Meaning that once we reached a convex structure, the good theoretical properties shared by the RMSProp with the Adagrad algorithm would make it converge faster. The Rmsprop algorithm can be seen in Algorithm 5

Algoritmo 5: RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ
Require: Initial parameters θ
Require: Small constant δ, normally 10^{-6}, for numerical stability
while *stopping criterion not met* **do**
  Sample a minibatch of n examples from the training set $\{x^{(1)}, \ldots, x^{(n)}\}$ with corresponding targets $y^{(i)}$.
  Compute gradient estimate: $g \leftarrow +\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  Accumulate squared gradient: $r \leftarrow \rho r + (1 - \rho) g \odot g$
  Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\sqrt{\delta r}} \odot g$
  Apply update: $\theta \leftarrow \theta + \Delta\theta$

Another variation of RMSProp was proposed in (GRAVES, 2013). In this version, the gradient is normalized by an estimation of its variance. It seems that this modification improved the results for recurrent network structures.

2.4.4.3 Adam

The Adam algorithm (KINGMA; BA, 2014) is a combination of RMSPRop and the momentum used in SGD. The two methods are called first-moment term and second-moment term in the Adam algorithm. It also includes a bias correction term to account for the initialization of the momentum terms at zero. The formal Adam algorithm can be seen in the Algorithm 6.

Algoritmo 6: Adam algorithm

Require: Learning rate ϵ
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$
Require: Small constant δ, normally 10^{-8}, for numerical stability
Require: Initial parameters θ
while *stopping criterion not met* **do**
  Sample a minibatch of n examples from the training set $\{x^{(1)}, \ldots, x^{(n)}\}$ with corresponding targets $y^{(i)}$.
  Compute gradient estimate: $g \leftarrow +\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  $t \leftarrow t + 1$
  Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1)g$
  Update biased second moment estimate: $t \leftarrow \rho_2 r + (1 - \rho_2)g \odot g$
  Correct bias in first moment: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$
  Correct bias in second moment: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
  Compute update: $\Delta\theta \leftarrow -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$
  Apply update: $\theta \leftarrow \theta + \Delta\theta$

Recent research (REDDI; KALE; KUMAR, 2019) has found some theoretical shortcomings of the Adam algorithm, where the usage of exponential moving average caused the non-convergency on a convex toy-problem. An alternative method, called Amsgrad, was then developed to overcome this deficiency.

2.4.5 Second-Order Methods

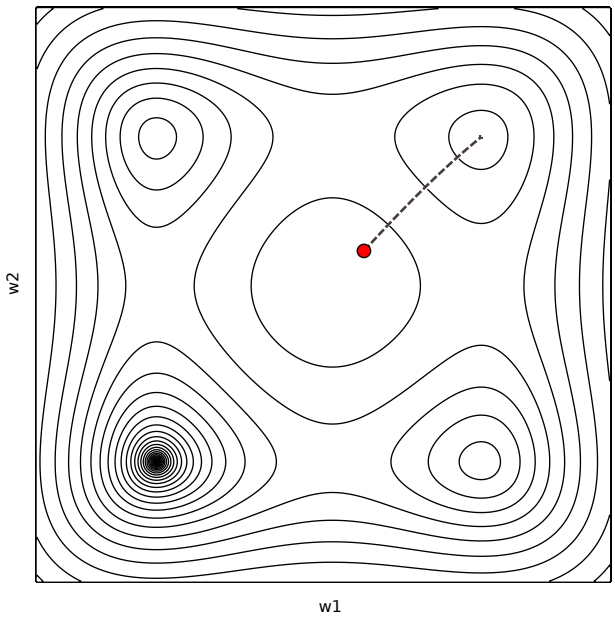
The methods mentioned above all use the gradient of the loss function to update the weights. Since they use the first derivative, they are called first-order methods. We also have the second-order methods (GOODFELLOW; BENGIO; COURVILLE, 2016), such as the Newton method, Conjugate gradients, and BFGS, which use the second derivative to improve optimization. While second-order methods tend to work better than first-order in more general optimization problems, the same cannot be said for the deep learning case. First of all, there is the computational cost of applying such methods. The basis of all those methods is the Hessian matrix, composed by the partial derivatives of all the network parameters. So if the network has k parameters, the Hessian is going to have $k \times k$ elements; a considerable number of variables. This Hessian matrix will also need to be calculated at each interaction since the parameters change at each step. Because of

those limitations, only small networks can be trained with such algorithms. The second problem is that most second-order methods are attracted to saddle points and points of local maxima, arriving at non-ideal solutions, making the results worse than first-order methods. For these factors, second-orders methods are generally not used for deep learning optimization.

3 PROPOSED METHOD

The loss function space is highly dimensional (the dimension is equal to the number of parameters of the network) with quite complex geometry. It is filled with local minima, and saddle points that would slow down or halt training. Training methods that use some sort of gradient descent are attracted to local minima, meaning they will converge for one and stay there if no other action is made. An example of such case is presented of Figure 10 where we have a bi-dimensional loss function for different values of two parameters w_1 and w_2. In Figure 10, the parameter initialization made the initial point closer to one particular local minimum. By using a learning algorithm based on gradient descent, the direction of greater descent was dominated by this local minimum, thus making the parameters converge to it. Note that we have a smaller minimum value, but since gradient descent only calculates the gradient locally, it is oblivious of the general shape of the loss function.

Figure 10 – Example of how an algorithm using gradient descent would find a local minimum of a function.

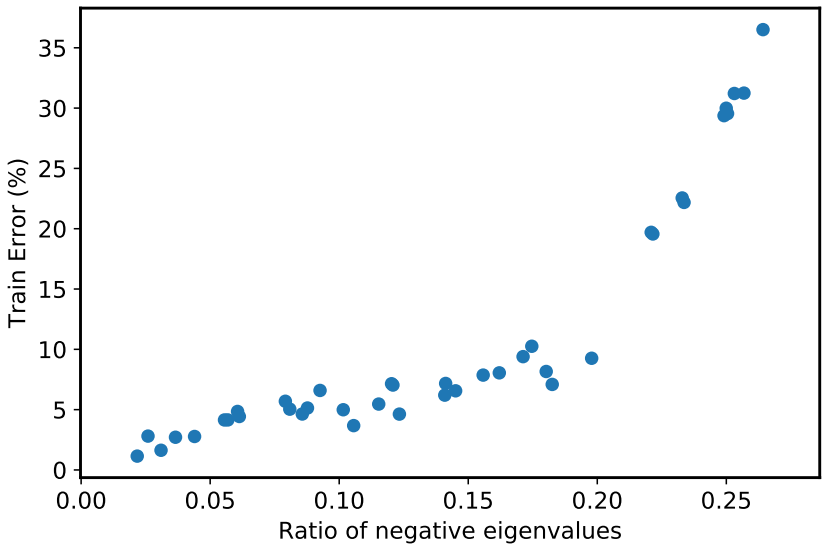


Source: Adapted from <http://www.deeplearning.ai/ai-notes/optimization/>

For a long time, this was deemed to be a shortcoming on gradient training methods since the local minimum loss value could be much higher than the true global minimum, and the algorithm would not know better. However, new research on the field (DAUPHIN et al., 2014) seems to indicate that the higher the dimension of an optimization space, the smaller the chance of local minima appearing, with saddle points being much more common. Another observed behavior was that the error of critical points was correlated with the fraction of negative eigenvalues (local minima have only positive eigenvalues,

local maxima only negative, and saddle points have both) on its hessian matrix; meaning that if a critical point has an error much bigger than the global minimum it's is highly probable it is a saddle point or a local maxima. This behavior is depicted in Figure 11, where we have the training error for different critical in a typical loss function space of a deep learning network. There we can see that the higher the fraction of negative eigenvalues, the greater the error. Similarly, when we have only positive eigenvalues (zero on the x-axis of Figure 11) the error is very close to zero.

Figure 11 – Plot of the training error of a critical point regarding its ratio of negative eigenvalues in its Hessian matrix.



Source: Adapted from (DAUPHIN et al., 2014).

The two points above could be summarized as such: in cases of a high dimensional space (which is the case of deep learning optimization), local minima are rare and with loss values close to the global minimum. Saddle points are also usually surrounded by regions of small curvature called plateaus, where the learning process can become very slow because of small gradients. So one can argue that there are two different phases on the correct training of a deep learning network: First correctly move between the much more present saddle points, avoiding unnecessary slowdowns, and then once close to a local minimum, converge to it. In that sense, in the initial portion of training, I propose that a higher learning rate would be more advisable to make traversing the plateaus easier. Once we arrive near the vicinity of local minima, a lower learning rate is more suited to converge to this critical point. With these arguments, I propose a scheme of learning rate decay with two distinct phases [1], suited for the two phases of training optimization laid out above. While there are other training methods which propose a learning rate decay, the novelty regarding this one is the fact that it takes into consideration the total

[1] Another line of research that corroborate that are two distinct phases during training was done by (SHWARTZ-ZIV; TISHBY, 2017), who presents arguments using information theory background.

number of epochs, and escalates the transition point between the two phases of training accordingly. The hyperparameters also control when and how fast the change between the two phases in training occur, which could also be of interest.

The proposed method is called Training Aware Sigmoid Optimization (TASO). The mathematical formulae of TASO is shown in Equation 3.1, where ϵ_i and ϵ_f are the initial and final learning rate respectively, k is the current epoch, k_t is the total number of epochs, and α, β are hyperparameters. The formal definition of TASO can be seen in Algorithm 7. The formula is a sigmoid function in the form of $\frac{1}{1+\exp(x)}$, with the hyperparameters added to fine control the sigmoid behavior.

$$\epsilon = \frac{\epsilon_i}{1 + \exp(\alpha(\frac{k}{k_t} - \beta))} + \epsilon_f \tag{3.1}$$

Algoritmo 7: TASO algorithm

Require: initial learning rate ϵ_i and final learning rate ϵ_f
Require: hyperparameters α and β
Require: Initial parameters θ
while *stopping criterion not met* **do**
  Sample a minibatch of n examples from the training set $\{x^{(1)}, \ldots, x^{(n)}\}$ with corresponding targets $y^{(i)}$.
  Compute gradient estimate: $g \leftarrow +\frac{1}{n}\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  Compute new learning rate: $\epsilon \leftarrow \frac{\epsilon_i}{1+\exp(\alpha(\frac{k}{k_t}-\beta))} + \epsilon_f$
  Apply update: $\theta \leftarrow \theta - \epsilon g$

An example of the function behavior is shown in Figure 12, where we have an initial learning rate that is maintained for the initial part of training and after a certain number of epochs decrease to a lower learning rate for the remaining of training. With this approach, we can attain the desired behavior intended for the learning rate, with two distinct phases with a simple mathematical definition.

The idea is that to obtain a function that at the beginning of training has a learning rate equal to ϵ_i and during later parts of training has a learning rate equal to ϵ_f. Equation 3.2 and Equation 3.3 shows how TASO learning rate evaluates in the first and last epoch respectively.

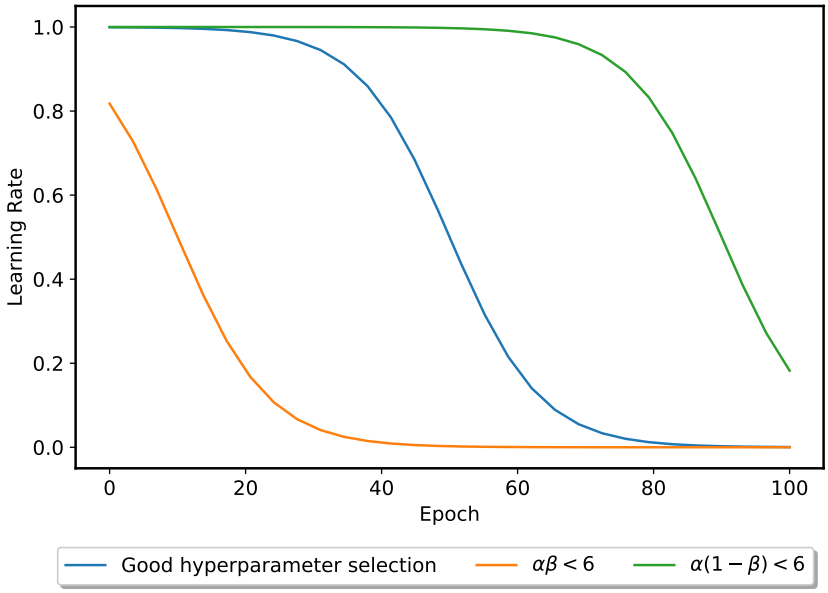
$$\epsilon_{k=0} = \frac{\epsilon_i}{1 + \exp(\alpha(\frac{0}{k_t} - \beta))} + \epsilon_f = \frac{\epsilon_i}{1 + \exp(-\alpha\beta)} + \epsilon_f \approx \epsilon_i + \epsilon_f \approx \epsilon_i \tag{3.2}$$

$$\epsilon_{k=k_f} = \frac{\epsilon_i}{1 + \exp(\alpha(\frac{k_t}{k_t} - \beta))} + \epsilon_f = \frac{\epsilon_i}{1 + \exp(\alpha(1 - \beta))} + \epsilon_f \approx \epsilon_f \tag{3.3}$$

Note that the hyperparameters ϵ_i and ϵ_f were characterized as the initial and final learning rates. However, this is not the case, since for both equations, as seen above, there is an approximation sign. Thus, by restricting the choices of α and β, we can make the

difference small enough to be negligible. As a useful heuristic, both $\alpha\beta$ and $\alpha(1-\beta)$ to be higher than 6 as to maintain errors bellow 5%. These values were empirically defined. An example of how the choice of non-conforming values of α and β can create degenerative cases can be seen if Figure 12

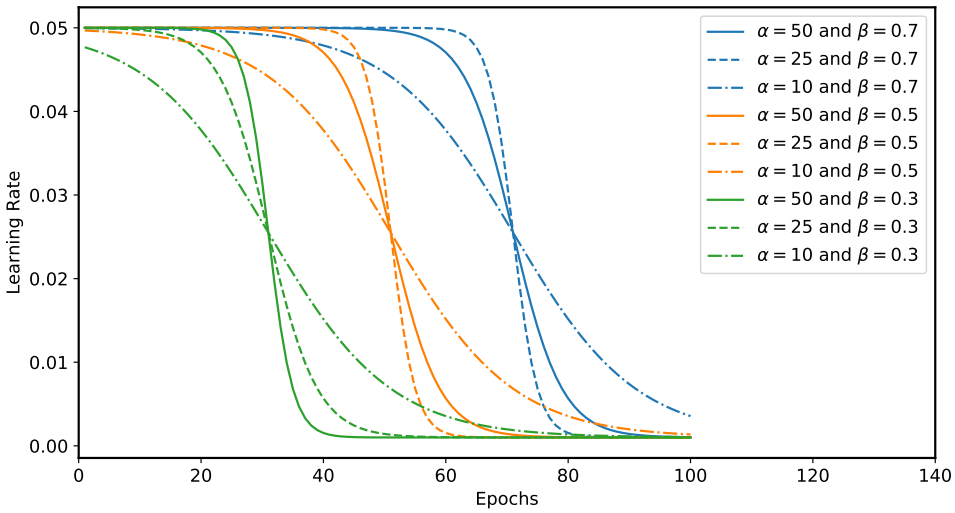
Figure 12 – Example of how the hyperparameter choice can create a degenerate case where the two-phase training is not well implemented.



Source: The Author

Apart from the learning rates, the hyperparameters of the TASO method are α, β. Where α controls how steep the decrease in the logistic function occurs, and β dictates when does the learning rate reaches its halfway point. The change of α and β on the overall shape of the learning rate can be seen in Figure 13. Note what happens when the heuristics for the selection of α and β are not respected. In the first case with $\alpha = 10$ and $\beta = 0.3$, we have $\alpha\beta$ equal to 3, making the initial learning rate not close enough then our expected value of 0.05. In the second case with $\alpha = 10$ and $\beta = 0.7$, we have $\alpha(1-\beta)$ also equal to 3, which makes the final learning rate not reach the expected value of 0.0025.

Figure 13 – Effects on hyperparameters changes on TASO with ϵ_i equal to 0.05, ϵ_f equal to 0.0025 and 100 total epochs.



Source: The Author

4 EXPERIMENTS

In the experiments, I would like to compare TASO with the more commonly used training algorithms in different architectures and datasets. In the first pair of architecture-dataset experiments, there is going to be a grid search to find the best learning rate and other hyperparameters. Once the best set of hyperparameters is found, they are going to be re-used on further experiments with different datasets and architectures. The motive behind this choice is to first, reduce the amount of training time to find the best hyperparameters for each architecture-dataset pair, and second, to test the robustness [1] of the training algorithms being tested with sub-optimal hyperparameters.

4.1 TRAINING ALGORITHMS

For the selection of the training algorithms, it was used the ones tested by (WILSON et al., 2017), which are SGD and its momentum version, RMSProp, Adam, and Adagrad. This work also tries to compare different learning algorithms and have a robust evaluation procedure, making o good pick as a starting point. The hyperparameter selection was also guided by it. Thew framework I used for the tests was Pytorch version 1.1, which has all these methods already implemented. The Computer used to run the tests had the following configuration:

- **CPU**: intel Core i7-7700K
- **GPU**: NVidia TitanXP
- **RAM**: 16GB

The learning rate tested for each algorithm was the following ones:

- **SGD**: $[2, 1, 0.5, 0.25, 0.05, 0.01, 0.001]$
- **RMSProp**: $[0.01, 0.005, 0.001, 0.0005, 0.0003, 0.0001]$
- **Adagrad**: $[0.1, 0.05, 0.01, 0.0075, 0.005]$
- **Adam**: $[0.005, 0.001, 0.0005, 0.0003, 0.0001, 0.000005]$

For the RMSProp algorithm, I opted to use the default value of 0.99 for the exponential decay rate and not to test other values since it seems to have little influence on the overall

[1] Ideally we want a algorithm that perform well with a large set of hyperparameters and not only with the best hand-picked ones after several initial tests. The concept of robustness characterizes such behaviour. So when we said that given algorithm is robust it means that its final result doesn't depends so much on the set of hyperparameters chosen

result. Still, for RMSprop, I also choose to test the alternate version proposed by (GRAVES, 2013). In the tests, it is named RMSProp centered. Together with Adam, I will evaluate its alternative version, AMSGrad. For both Adam and AMSGrad, the exponential decay terms ρ_1 and ρ_2 were left at the default values of 0.9 and 0.99 as similar to the RMSProp case, changes on their values do not seem to impact training in a very significant manner. Finally, for TASO I will be testing the values of hyperparameters showed in Figure 13 which are $\alpha = [10, 25, 50]$ and $\beta = [0.3, 0.5, 0.7]$=, which seems to encompass a good range of possible configurations. For the initial learning rate, I am going to use the best one to be found in the SGD experiment, and the final one is going to be 20 times smaller than the initial one, empirically defined. Note that are two cases where the choices of α and β fall out of the recommended zone defined in section 3.1. This was done to check if not following the determined heuristic would be accompanied with worse results.

4.2 DATABASES

The tests were done using three different databases the MNIST, CIFAR10, and CIFAR100. All of them are image classification datasets but with different degrees of complexity and size.

MNIST (LECUN et al., 1998) is a dataset of handwriting digits composed of 70,000 greyscale images. It has a training set of 60,000 examples and a test set of 10,000 images. The images have a size of 28 pixels by 28 pixels with the digits being normalized and centralized. Figure 14b show a small subset of the MNIST dataset. While this dataset is quite old, originating in 1998, it is still useful. Its simplicity makes it very easy and fast to properly train, making it very useful to evaluate if novel algorithms are appropriately tuned.

CIFAR10 and CIFAR100 are both subsets of the Tiny image dataset (TORRALBA; FERGUS; FREEMAN, 2008) differing from each other regarding the number of classes. CIFAR10 has 10 classes and CIFAR100 has 100 classes. They both have 60,000 color images divided between 50,000 training images and 10,000 test images. The images have a size of 32 pixels by 32 pixels. Figure 14a show an example of images comprising the CIFAR10 dataset.

Figure 14 – Examples of the used datasets.

(a) CIFAR10 database (b) MNIST database

Source:<https://ai-coordinator.jp/mnist-ng>

4.3 ARCHITECTURES

Three CNN architectures are going to be used in experiments. Lenet5 (LECUN et al., 1998) VGGNet19 (SIMONYAN; ZISSERMAN, 2014) and Resnet18 (HE et al., 2015). Lenet5 was developed in 1998 with the purpose of identifying handwriting digits. It has seven-layer in total, three convolutional layers, two subsampling layers, one fully connected layer, and an output layer formed with Euclidean Radial Basis Functions (to give the classes probabilities). It is one of the first CNN models to present good overall results. While it is pretty old, and applying some practices not in use anymore, such as Radial basis functions in the output layer and hyperbolic tangent activation function, it is a reliable option to make fast and simple tests in older databases such as the MNIST.

VGGNet was created in 2014 and was one of the runner ups of the ILSVRC 2014 competition. It has 19 weight layers(16 convolutional layers plus three fully connected layers). All the convolutional layers use 3x3 filters, a strategy applied to reduce the number of trainable parameters and to make it possible to increase the number of layers.

Resnet was created in 2015 and won several image classification competitions (ILSVRC 2015 image detection and localization, 2015 COCO detection and segmentation). It is characterized by the use of skip connections and massive batch normalization. It uses only a fully connected layer in the output, reducing the number of parameters to be trained and making it possible to archive much higher depths (the network used to won ILSVRC 2015 used 152 layers).

4.4 EXPERIMENT DESIGN

When designing the experiments, we want to come up with a test scheme that would allow us to compare the different learning algorithms. One that via analyzing its results,

we could infer if any of the available algorithms are better then the other. In that sense, the items bellow were used as guiding principles to the design of the experiment.

- **Reproducible**: By the intrinsic nature of deep learning algorithms, we can arrive at different results, even when working with the same dataset, architecture, and learning algorithm. Things such as weight initialization, batch size, and data ordering influence the final result. In that sense, we must present a way to us (or any other researcher) to reproduce our findings. This was done by precisely describing our experimental protocol and fixing a seed for the random number generator, which will make every random step deterministic.
- **Statistical Significance**: As was mentioned above, there are a series of factors that can influence the final result of the deep learning training task. Usually, this results in differences in the range of 1-2% in the loss function across multiple runs of the training algorithm. Nevertheless, however small the difference may be, it is a good practice to do some statistical testing to evaluate the performance of different algorithms correctly. A heuristic usually applied is to do 30 runs of each test to have enough data to make educated assumptions. Unfortunatelly, using such a high number of repetitions was not viable in most deep learning approaches since the training time is very long. In that case, we compromised to doing five runs, as it would be enough at least to calculate the mean and standard deviation, and have a partial notion of the statistical distribution of the results.
- **Information Tracking:** At each test, we need to select what kind of parameters we would like to track across the experiment as to later be able to analyze them. We opted to track the accuracy and loss function across each interaction and epoch from the training and test sets.
- **Wide Hyperparameters Search**: Each training algorithm has a series of hyperparameters that will change the training result. Thus it is necessary to test several configurations to see how they interfere with the algorithm performance.

In summary, we are doing a test with a fixed seed (reproducible), with five runs (statistical significance), tracking the loss function and accuracy value across each epoch varying some hyperparameters (hyperparameters search) for each learning algorithm.

Regarding hyperparameters choices, ideally, we should perform a grid or random search, across all hyperparameters available. However, this approach is usually not doable, considering the amount of time necessary to train a deep learning network. For this reason, we decided on some tests to evaluate each hyperparameter individually, maintaining all the other ones constant. While not an ideal approach, since we can always have a combination of hyperparameters not tested with better performance, it presents a good compromise between reducing test time and finding a better result.

Another important topic regarding hyperparameters is that the ideal set is dependent on the choice of architecture and dataset. However, as the grid search argument, to search for a different set of hyperparameters each time you face a new problem is counterproductive. A good training algorithm, beyond having an ideal set of hyperparameters, that would result in good inference performance, is one which default hyperparameters could be used across different experiments and give a good result. In that sense, I decided to do a full hyperparameter search only in the first dataset and architecture, and use the best results found in this test on subsequent tests. With this approach, we are not only testing the best possible result in each learning algorithms but also its robustness, which is a more important measure in real-world usability than the ideal best performance.

Lastly, with the intent of improving transparency and also to help further research on the topic, I made public the code necessary to run all the tests presented here. It can be found in <https://github.com/pedro-dreyer/masters_code>

4.5 RESULTS

4.5.1 VGG10 and CIFAR10

4.5.1.1 SGD

For the SGD algorithm, we tested seven learning rates starting from 2 until 0.001. We opted to use a moment of 0.9, with and without the Nesterov version. We also tested the same learning rates with no moment at all. The overall results can be seen in Table 1 [2].

For further comparison, we pick up the best results of each approach and compare the training progression. The results can be seen in Figures 15. While the final value of the three algorithms is very similar, the non-Nesterov version has a better overall performance. Other values of moments were tested as well their results can be seen in Table 2 From those tests, we can conclude that the best parameters were a learning rate of 0.05 with a Non-Nesterov moment of 0.9.

[2] All values presented on tables in this document are referent to the results obtained in the test set

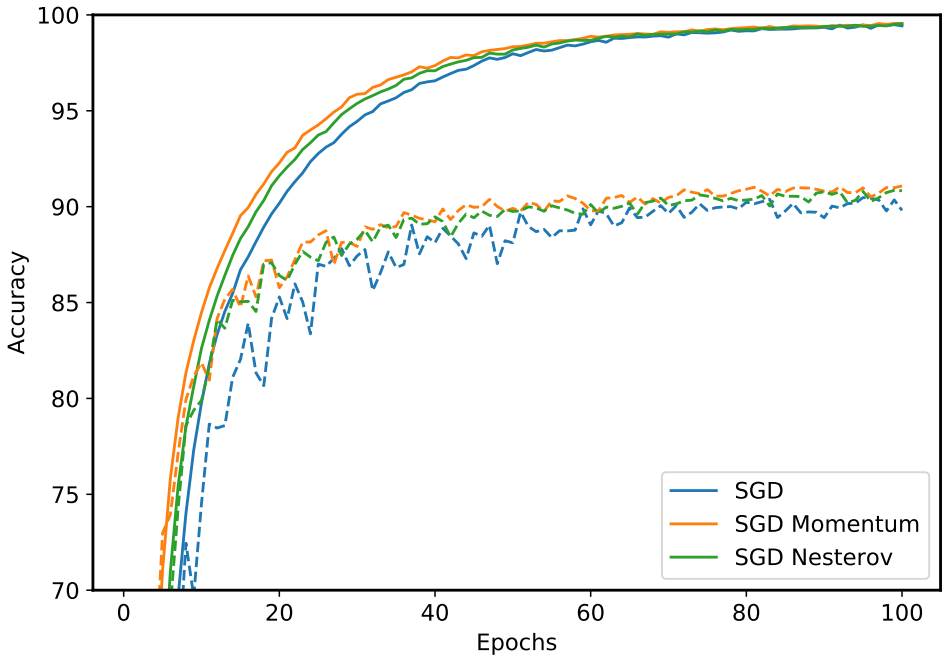
Table 1 – Different learning rates and type of moments tested for the SGD algorithm. CIFAR10 dataset and VGG19 architecture. The moment is equal to 0.9 for both non-Nesterov and Nesterov versions.

Method	Learning Rate	Accuracy	Loss
SGD	2	35.91 (± 36.64)	1.69 (± 0.87)
	1	88.94 (± 0.18)	0.47 (± 0.01)
	0.5	89.72 (± 0.42)	0.42 (± 0.03)
	0.25	**90.39** (± 0.29)	0.41 (± 0.01)
	0.05	89.66 (± 0.18)	0.44 (± 0.02)
	0.01	86.00 (± 0.14)	0.52 (± 0.01)
	0.001	78.78 (± 0.10)	0.62 (± 0.01)
SGD Momentum	2	10.05 (± 0.03)	2.48 (± 0.15)
	1	10.04 (± 0.03)	2.33 (± 0.01)
	0.5	25.82 (± 22.38)	1.94 (± 0.53)
	0.25	55.32 (± 33.12)	1.33 (± 0.74)
	0.05	**91.01** (± 0.13)	0.37 (± 0.00)
	0.01	90.55 (± 0.11)	0.40 (± 0.01)
	0.001	86.38 (± 0.13)	0.50 (± 0.01)
SGD Nesterov	2	10.14 (± 0.10)	2.32 (± 0.00)
	1	10.08 (± 0.09)	2.31 (± 0.00)
	0.5	10.02 (± 0.02)	2.30 (± 0.00)
	0.25	28.70 (± 26.22)	1.93 (± 0.54)
	0.05	90.00 (± 0.71)	0.41 (± 0.01)
	0.01	**90.49** (± 0.28)	0.40 (± 0.01)
	0.001	86.56 (± 0.22)	0.51 (± 0.00)

Table 2 – Different moments values test for SGD non-Nesterov algorithm. CIFAR10 dataset and VGG19 architecture. Learning rate equal to 0.05.

Momentum	Acurracy	Loss
0.1	90.01 (± 0.08)	0.44 (± 0.03)
0.3	90.32 (± 0.42)	0.42 (± 0.01)
0.5	90.67 (± 0.06)	0.40 (± 0.02)
0.7	90.90 (± 0.44)	0.38 (± 0.01)
0.9	**91.01** (± 0.13)	0.37 (± 0.00)
0.99	54.64 (± 32.65)	1.34 (± 0.75)

Figure 15 – Comparison between the SGD algorithms training the VGG19 architecture in the CIFAR10 dataset. Solid lines are from the training set and dashed lines from the test set.



Source: The Author

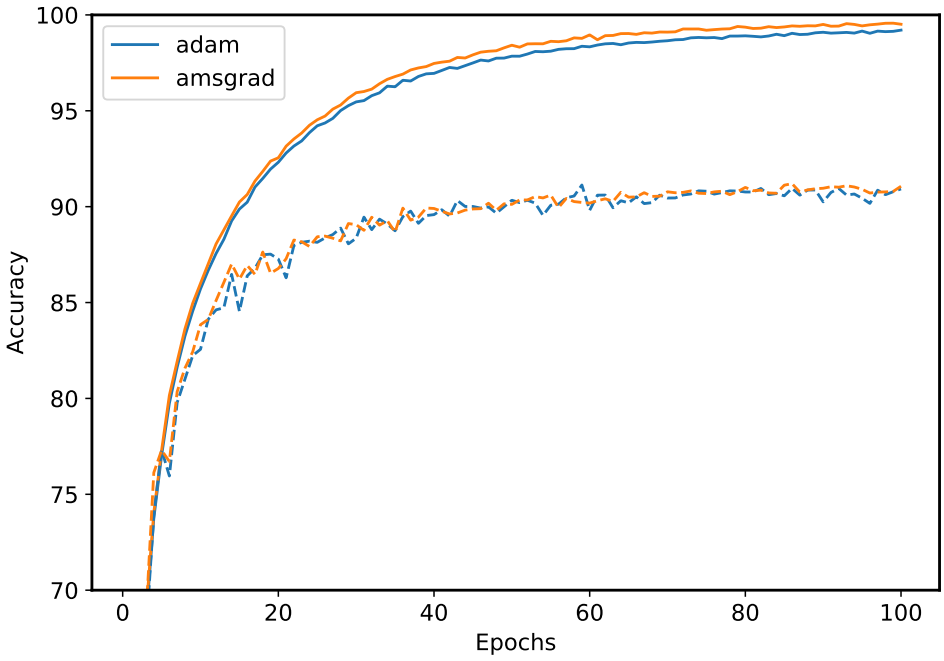
4.5.1.2 Adam

For the Adam algorithm, we tested six learning rates varying from 0.005 to 0.00005. We also compared two versions of the algorithm, the Amsgrad and the original version. A summary of the results is shown in Table 3. A plot comparing the best result of each experiment can be seen in Figure 16 From the experiments, we note that the learning rates 0.001, 0.0005, and 0.0003 have very close results. While the Amsgrad version presented a better training accuracy, its test accuracy appears to be very similar to the original Adam algorithm. For further tests, the Amsgrad Adam algorithm is going to be used with learning rate 0.0005, since it was the one with the highest accuracy.

Table 3 – Different learning rates tested for the Adam Algorithm. CIFAR10 dataset and VGG19 architecture.

Method	Learning Rate	Accuracy	Loss
ADAM	0.005	90.173 (± 0.050)	0.397 (± 0.014)
	0.001	90.933 (± 0.347)	0.381 (± 0.003)
	0.0005	**91.027** (± 0.042)	**0.374** (± 0.013)
	0.0003	90.773 (± 0.188)	0.378 (± 0.004)
	0.0001	88.967 (± 0.094)	0.446 (± 0.006)
	5e-05	86.540 (± 0.237)	0.522 (± 0.007)
ADAM amsgrad	0.005	90.387 (± 0.146)	0.401 (± 0.005)
	0.001	91.013 (± 0.343)	0.376 (± 0.013)
	0.0005	**91.333** (± 0.157)	**0.369** (± 0.013)
	0.0003	91.063 (± 0.273)	0.370 (± 0.004)
	0.0001	88.667 (± 0.115)	0.458 (± 0.004)
	5e-05	86.403 (± 0.097)	0.536 (± 0.004)

Figure 16 – Comparison between the best Adam algorithms for the CIFAR10 dataset and VGG19 architecture. Solid lines are from the training set and dashed lines from the test set.



Source: The Author

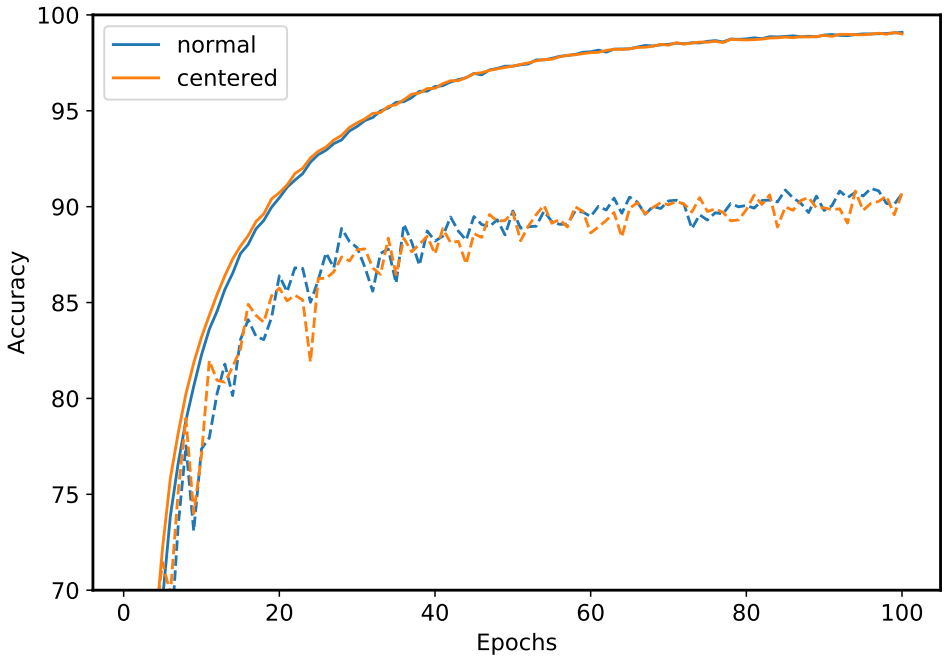
4.5.1.3 Rmsprop

For the Rmsprop algorithm, we evaluated six learning rates ranging from 0.01 to 0.0001. We also evaluated the opting of using a centered and non-centered version of the method. Table 4 presents the best results for each hyperparameter set. Figure 17 compares the best results for centered and non-centered versions. The learning rates from 0.0003 to 0.001 presented similar results across both algorithms, and there is to appear little difference between the centered and non-centered version of the algorithm. For further tests, the non-centered version with a learning rate of 0.00005 is going to be used, since it presented the best overall results.

Table 4 – Different Learning rates and centered parameter for the Rmsprop algorithm. CIFAR10 dataset and VGG19 architecture.

Method	Learning Rate	Accuracy	Loss
RMSPROP	0.01	89.947 (± 0.130)	0.442 (± 0.022)
	0.005	90.107 (± 0.321)	0.426 (± 0.012)
	0.001	90.743 (± 0.012)	0.407 (± 0.014)
	0.0005	**90.773** (± 0.074)	**0.392** (± 0.020)
	0.0003	90.547 (± 0.152)	0.402 (± 0.012)
	0.0001	88.753 (± 0.399)	0.462 (± 0.014)
RMSPROP centered	0.01	89.967 (± 0.174)	0.433 (± 0.019)
	0.005	90.00 (± 0.177)	0.425 (± 0.010)
	0.001	90.653 (± 0.262)	0.390 (± 0.030)
	0.0005	**90.760** (± 0.214)	**0.388** (± 0.020)
	0.0003	90.580 (± 0.161)	0.387 (± 0.031)
	0.0001	88.720 (± 0.380)	0.467 (± 0.011)

Figure 17 – Comparison between the best Rmsprop algorithms for the CIFAR10 dataset and VGG19 architecture. Solid lines are from the training set and dashed lines from the test set.



Source: The Author

4.5.1.4 Adagrad

For the Adagrad method, we tested five learning rates ranging from 0.1 to 0.005. From the results, a learning rate of 0.05 was chosen for further experiments.

Table 5 – Results of the tests for the Adagrad algorithm comparing multiple learning rates. CIFAR10 dataset and VGG19 architecture.

Learning Rate	Accuracy	Loss
0.1	89.183 (± 0.163)	0.453 (± 0.017)
0.05	89.52 (± 0.311)	0.451 (± 0.006)
0.01	88.772 (± 0.091)	0.462 (± 0.019)
0.0075	88.092 (± 0.020)	0.471 (± 0.011)
0.005	**87.805** (± 0.095)	**0.452** (± 0.006)

4.5.1.5 TASO

The TASO experiments were done using the best learning rate of the SGD experiments (0.05). The tests analyzed how two sets of hyperparameters, α and β, would interfere in training. As mentioned in the section of Experiment Design, three values of α (10, 25, and 50) and three values of β (0.3, 0.5, 0.7) were tested.

The results of training across the hyperparameters configurations can be seen in Table 6, where the accuracy and loss function of the test set is shown considering 100 epochs. According to the results, we see a small difference varying α and β with the best performance being archived by a α of 25 and a β of 0.7. There was also two tests were the hyperparameters were out of the proposed range, namely $\alpha = 10$, $\beta = 0.3$ and $\alpha = 50, \beta = 0.7$. While the first case presented the lowest accuracy, the second one performed well. While these are only preliminary results, the heuristic of maintaining both $\alpha\beta$ and $\alpha(1 - \beta)$ greater than six is still recommended. It was also verified that the new calculations added in the TASO algorithm did not interfere with the training time. A expected result since the backpropagation algorithm itself is much more computationally intensive than the calculation of the new learning rate each interaction added by TASO.

4.5.1.6 Overall results

Finally the comparison between the best result for each method can be seen in Figure 18 and Table 7. Observing the graphics, we can note that around epoch 40, the training accuracy continues to increase where the test accuracy starts to plateau. Those signs are indications that an increase of training time would bear ever-diminishing returns. In other words, we can assume that 100 epochs were time enough to converge all the training algorithms.

Table 6 – Comparison among multiple sets of hyperparameters of the TASO algorithm.

α	β	Acurracy	Loss
10	0.3	90.96 (± 0.12)	0.37 (± 0.01)
10	0.5	91.66 (± 0.31)	0.38 (± 0.00)
10	0.7	91.97 (± 0.19)	0.38 (± 0.02)
25	0.3	90.73 (± 0.17)	0.37 (± 0.01)
25	0.5	91.61 (± 0.27)	0.37 (± 0.00)
25	0.7	**91.98** (± 0.19)	**0.35** (± 0.01)
50	0.3	90.85 (± 0.30)	0.36 (± 0.01)
50	0.5	91.94 (± 0.04)	0.37 (± 0.00)
50	0.7	91.95 (± 0.25)	0.37 (± 0.00)

The tests were also re-run using the best parameters for each training method for only 25 epochs. The aim is to verify how the algorithms would fare having a limited training time to make weights adjustment. The result can be found in Table 8 and Figure 19. On both epochs choices, the TASO method achieves a better result, while being more significant in the 25 epochs case. Also, in both experiments, we can easily visualize a bump in the accuracy close to the 20 and 70 epoch mark, where the learning rate of the TASO algorithm started to decrease.

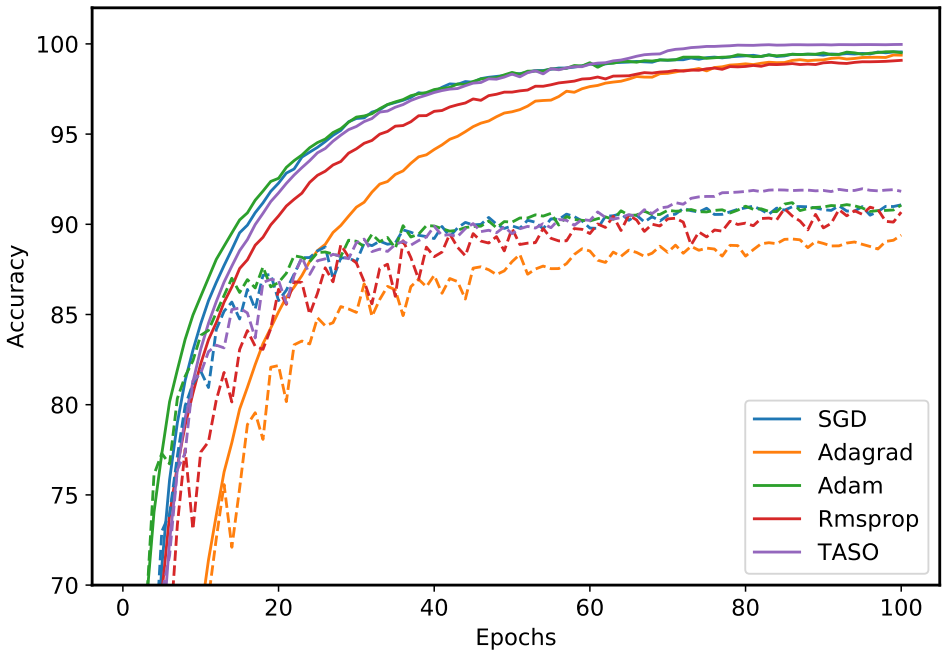
Table 7 – Results for each training algorithm using the best set of hyperparameters. CIFAR10 dataset and VGG19 architecture.

Training method	Acurracy	Loss
SGD	91.08 (± 0.13)	0.36 (± 0.00)
Adagrad	89.40 (± 0.31)	0.42 (± 0.01)
Adam	91.33 (± 0.16)	0.37 (± 0.01)
RMSProp	90.77 (± 0.01)	0.41 (± 0.01)
TASO	**91.98** (± 0.19)	0.36 (± 0.02)

Table 8 – Results of the best run of each algorithm for 25 epochs.

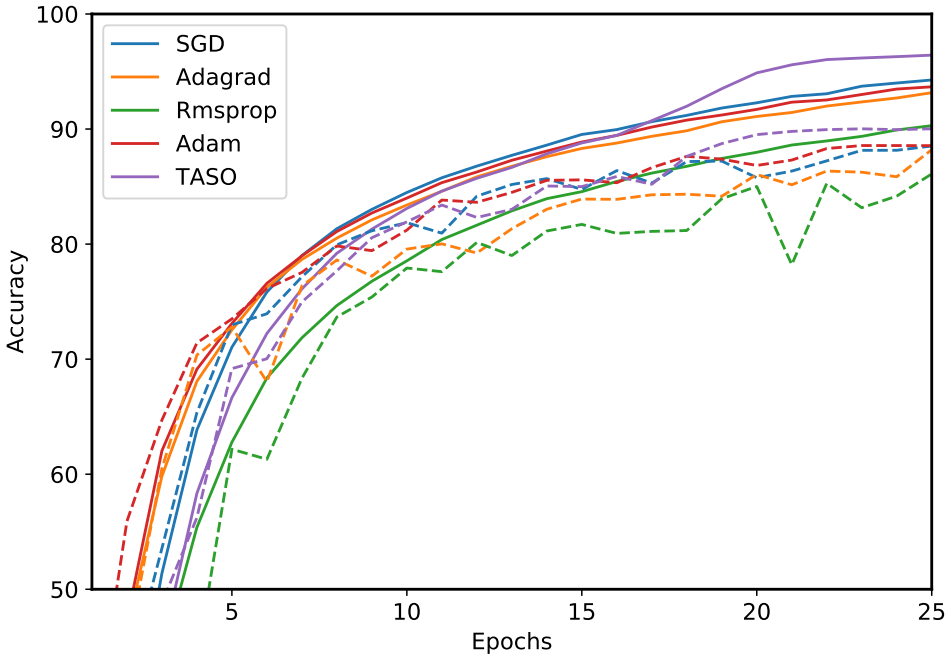
Training method	Acurracy	Loss
SGD	88.52 (± 0.02)	0.38 (± 0.01)
RMSProp	88.20 (± 0.33)	0.41 (± 0.01)
Adagrad	86.10 (± 0.27)	0.44 (± 0.01)
Adam	88.56 (± 0.16)	0.38 (± 0.00)
TASO	**90.02** (± 0.41)	**0.34** (± 0.01)

Figure 18 – Comparison between the best set of hyperparameters for each algorithm on the CIFAR10 dataset and VGG19 architecture. Solid lines are from the training set and dashed lines from the test set.



Source: The Author

Figure 19 – Comparison between the best set of hyperparameters for each algorithm on the CIFAR10 dataset and VGG19 architecture for 25 epochs. Solid lines are from the training set and dashed lines from the test set.



Source: The Author

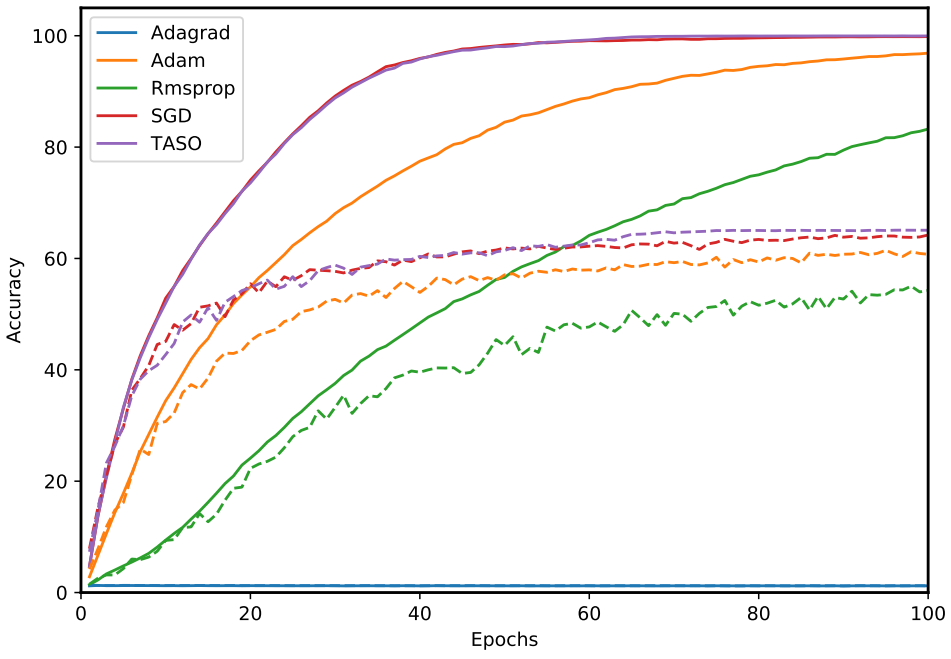
4.5.2 VGG19 and CIFAR100

Using the best hyperparameters from the previous experiment, the same architecture but now using the CIFAR100 dataset, was now evaluated. The results can be seen in Table 9 and Figure 20. We note once again that the TASO has the best overall result. A more profound case of overfitting is happening here since the training accuracy is very close to 100% while the test accuracy is much smaller. Probably if we increased the weight decay, the result would improve. Note as well that the Adagrad failed to converge, indicating the non-reliability of the algorithm.

Table 9 – Results using the best hyperparameters found using the VGG19 architecture and CIFAR10 dataset. CIFAR100 dataset and VGG19 architecture.

Training method	Acurracy	Loss
Adagrad	1.330 (± 0.064)	4.770 (± 0.179)
Adam	61.463 (± 0.138)	1.902 (± 0.040)
RMSProp	55.020 (± 0.546)	2.043 (± 0.054)
SGD	64.213 (± 0.538)	1.772 (± 0.046)
TASO	**65.083** (± 0.479)	**1.768** (± 0.012)

Figure 20 – Comparison between the best set of hyperparameters found in the VGG19 architecture and CIFAR10 dataset. CIFAR100 dataset and VGG19 architecture. Solid lines are from the training set and dashed lines from the test set.



Source: The Author

4.5.3 RESNET18 and CIFAR10

The results of changing the architecture but maintaining the CIFAR10 datasets can be seen in Table 10. Those are so far the best results, where the TASO method archives more than 2% from the second-best performing algorithm. This more significant improvement could come from the fact that Resnet has a much deeper architecture than VGG19, which could contribute to having a more complex loss function space, and thus making it harder to optimize without changing the learning rate. Note as well that Adagrad, while did converge this time, have much lower accuracy than the other methods.

Table 10 – Results using the best hyperparameters found using the VGG19 architecture and CIFAR10 dataset. CIFAR10 dataset and Resnet18 architecture.

Training method	Acurracy	Loss
Adagrad	19.033 (± 0.784)	2.197 (± 0.007)
Adam	92.357 (± 0.298)	0.353 (± 0.019)
RMSProp	92.130 (± 0.286)	0.372 (± 0.009)
SGD	92.550 (± 0.159)	0.364 (± 0.014)
TASO	**93.150** (± 0.086)	**0.358** (± 0.012)

4.5.4 MNIST and LENET5

The last test, comparing the LENET5 Architecture and MNIST dataset, is shown in Table 11. The results showed that all the methods except for Adagrad managed to archive near-perfect accuracy and thus could be considered equivalent to the task of best training this particular dataset and architecture. In that sense, we can not really use its results to make any particular claim regarding which algorithm was the best one.

Table 11 – Results using the best hyperparameters found using the VGG19 architecture and CIFAR10 dataset. MNIST dataset and Lenet5 architecture.

Training method	Acurracy	Loss
Adagrad	75.160 (± 8.010)	1.152 (± 0.369)
Adam	99.030 (± 0.036)	0.030 (± 0.001)
RMSProp	99.097 (± 0.037)	0.030 (± 0.001)
SGD	99.083 (± 0.046)	0.030 (± 0.002)
TASO	99.093 (± 0.038)	0.030 (± 0.002)

5 CONCLUSION

The problem of correctly training a machine learning algorithm is an old one, existing since the development of the first algorithms on the field. While the last years we saw a significant improvement in the deep neural networks field, with results never before seen, the old dilemma on how to properly train the network still remains. The crux of the matter is that the deep learning networks have millions, if not billions of parameters, giving rise to a multidimensional loss space with extremely complex non-convex geometry, making the finding of global minima not intuitive. In that sense, backed up theoretical discoveries, and empirical tests, this document proposed a novel learning rate decay method named TASO. TASO conducts training in two distinct phases using as a basis a sigmoid function and has a new approach compared to other training algorithms on how it uses the information of the total number of epochs to properly tune how the learning rate decay is going to take place.

Several tests were done to evaluate the overall performance of TASO, using three architectures encompassing old (LENET5) and more recent (VGG19, RESNET18) architectures on three image classification datasets(CIFAR10, CIFAR100, and MNIST) with different complexity levels. The performance of TASO was compared with other training algorithms (SGD, ADAM, RMSProp, and Adagrad) that are used in many tasks of deep learning across multiple other papers. The results have shown that even with minimal hyperparameter tuning, TASO was able to perform better or equal than all the other methods compared on tasks related to image classification without any increase in the overall training time. Overall the proposed method seems to be robust since it works on different datasets and architectures using the same set of hyperparameters, appearing to be well suited for training a general deep learning network.

5.1 FUTURE WORKS

While TASO appears to work reliably in the datasets and architectures evaluated so far, further tests are necessary. First, instead of five repetitions, it would be advised to do at least thirty runs to get some more samples. This would enable us to do some statistical tests such as Kruskal–Wallis, which would evaluate with some degree of statistical significance if TASO is better than the other training methods.

Second, a more diverse task set would be advisable. Deep learning is being used across multiple fields, but yet the tests realized only cover image classification. Future tests should include NLP, handwriting analysis, speech recognition to name a few.

Finally, there are also interesting new discoveries such as proposed by (SMITH; TOPIN, 2017) that, together with transfer learning, can get state of the art results with as little

as four epochs. It would be interesting to know if TASO could help improve even further those results.

REFERENCES

BENGIO, Y.; LAMBLIN, P.; POPOVICI, D.; LAROCHELLE, H. Greedy layer-wise training of deep networks. In: *Proceedings of the 19th International Conference on Neural Information Processing Systems*. Cambridge, MA, USA: MIT Press, 2006. (NIPS'06), p. 153–160. Disponível em: <http://dl.acm.org/citation.cfm?id=2976456.2976476>.

BREIMAN, L. Random forests. *Mach. Learn.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 45, n. 1, p. 5–32, out. 2001. ISSN 0885-6125. Disponível em: <https://doi.org/10.1023/A:1010933404324>.

BULÒ, S. R.; PORZI, L.; KONTSCHIEDER, P. In-place activated batchnorm for memory-optimized training of dnns. *CoRR*, abs/1712.02616, 2017. Disponível em: <http://arxiv.org/abs/1712.02616>.

CAUCHY, A. Méthode générale pour la résolution des systèmes d'équations simultanées. In: *Compte rendu des séances de l'académie des sciences*. [S.l.: s.n.], 1847. p. 372–379.

CORTES, C.; VAPNIK, V. Support-vector networks. *Mach. Learn.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 20, n. 3, p. 273–297, set. 1995. ISSN 0885-6125. Disponível em: <https://doi.org/10.1023/A:1022627411411>.

CYBENKO, G. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, Springer London, v. 2, n. 4, p. 303–314, dez. 1989. ISSN 0932-4194. Disponível em: <http://dx.doi.org/10.1007/BF02551274>.

DAUPHIN, Y. N.; PASCANU, R.; GÜLÇEHRE, Ç.; CHO, K.; GANGULI, S.; BENGIO, Y. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR*, abs/1406.2572, 2014. Disponível em: <http://arxiv.org/abs/1406.2572>.

DENG, J.; DONG, W.; SOCHER, R.; LI, L.-J.; LI, K.; FEI-FEI, L. ImageNet: A Large-Scale Hierarchical Image Database. In: *CVPR09*. [S.l.: s.n.], 2009.

DEVLIN, J.; CHANG, M.; LEE, K.; TOUTANOVA, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. Disponível em: <http://arxiv.org/abs/1810.04805>.

DUCHI, J.; HAZAN, E.; SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, JMLR.org, v. 12, p. 2121–2159, jul. 2011. ISSN 1532-4435. Disponível em: <http://dl.acm.org/citation.cfm?id=1953048.2021068>.

GLOROT, X.; BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*. [S.l.: s.n.], 2010.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>.

GRAVES, A. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013. Disponível em: <http://arxiv.org/abs/1308.0850>.

HE, K.; ZHANG, X.; REN, S.; SUN, J. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. Disponível em: <http://arxiv.org/abs/1512.03385>.

HINTON, G. E.; OSINDERO, S.; TEH, Y.-W. A fast learning algorithm for deep belief nets. *Neural Comput.*, MIT Press, Cambridge, MA, USA, v. 18, n. 7, p. 1527–1554, jul. 2006. ISSN 0899-7667. Disponível em: <http://dx.doi.org/10.1162/neco.2006.18.7.1527>.

HUNSBERGER; ERIC. *Spiking Deep Neural Networks: Engineered and Biological Approaches to Object Recognition*. UWSpace, 2018. Disponível em: <http://hdl.handle.net/10012/12819>.

IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. Disponível em: <http://arxiv.org/abs/1502.03167>.

JAMES, G.; WITTEN, D.; HASTIE, T.; TIBSHIRANI, R. *An Introduction to Statistical Learning: With Applications in R*. [S.l.]: Springer Publishing Company, Incorporated, 2014. ISBN 1461471370, 9781461471370.

JARRETT, K.; KAVUKCUOGLU, K.; RANZATO, M.; LECUN, Y. What is the best multi-stage architecture for object recognition? In: *2009 IEEE 12th International Conference on Computer Vision, ICCV 2009*. [S.l.: s.n.], 2009. p. 2146–2153. ISBN 9781424444205.

KINGMA, D. P.; BA, J. *Adam: A Method for Stochastic Optimization*. 2014. Cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. Disponível em: <http://arxiv.org/abs/1412.6980>.

KREMER, S. C. *Field Guide to Dynamical Recurrent Networks*. 1st. ed. [S.l.]: Wiley-IEEE Press, 2001. ISBN 0780353692.

LECUN, Y.; BOTTOU, L.; BENGIO, Y.; HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, v. 86, n. 11, p. 2278–2324, Nov 1998. ISSN 0018-9219.

LINNAINMAA, S. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, v. 16, n. 2, p. 146–160, Jun 1976. ISSN 1572-9125. Disponível em: <https://doi.org/10.1007/BF01931367>.

MINSKY, M.; PAPERT, S. *Perceptrons*. Cambridge, MA: MIT Press, 1969.

MONTAVON, G.; ORR, G. B.; MÜLLER, K. (Ed.). *Neural Networks: Tricks of the Trade - Second Edition*. Springer, 2012. v. 7700. (Lecture Notes in Computer Science, v. 7700). ISBN 978-3-642-35288-1. Disponível em: <https://doi.org/10.1007/978-3-642-35289-8>.

NEWELL, A.; SIMON, H. The logic theory machine–a complex information processing system. *IRE Transactions on Information Theory*, v. 2, n. 3, p. 61–79, Sep. 1956. ISSN 0096-1000.

PASCANU, R.; MONTUFAR, G.; BENGIO, Y. On the number of response regions of deep feed forward networks with piece-wise linear activations. *arXiv preprint arXiv:1312.6098*, 2013.

POLYAK, B. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, v. 4, p. 1–17, 12 1964.

REDDI, S. J.; KALE, S.; KUMAR, S. On the convergence of adam and beyond. *CoRR*, abs/1904.09237, 2019. Disponível em: <http://arxiv.org/abs/1904.09237>.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning Representations by Back-propagating Errors. *Nature*, v. 323, n. 6088, p. 533–536, 1986. Disponível em: <http://www.nature.com/articles/323533a0>.

SAON, G.; KURATA, G.; SERCU, T.; AUDHKHASI, K.; THOMAS, S.; DIMITRIADIS, D.; CUI, X.; RAMABHADRAN, B.; PICHENY, M.; LIM, L.; ROOMI, B.; HALL, P. English conversational telephone speech recognition by humans and machines. *CoRR*, abs/1703.02136, 2017. Disponível em: <http://arxiv.org/abs/1703.02136>.

SHWARTZ-ZIV, R.; TISHBY, N. Opening the black box of deep neural networks via information. *CoRR*, abs/1703.00810, 2017. Disponível em: <http://arxiv.org/abs/1703.00810>.

SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

SMITH, L. N.; TOPIN, N. Super-convergence: Very fast training of residual networks using large learning rates. *CoRR*, abs/1708.07120, 2017. Disponível em: <http://arxiv.org/abs/1708.07120>.

SRIVASTAVA, N.; HINTON, G.; KRIZHEVSKY, A.; SUTSKEVER, I.; SALAKHUTDINOV, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, JMLR.org, v. 15, n. 1, p. 1929–1958, jan. 2014. ISSN 1532-4435. Disponível em: <http://dl.acm.org/citation.cfm?id=2627435.2670313>.

SUTSKEVER, I.; MARTENS, J.; DAHL, G.; HINTON, G. On the importance of initialization and momentum in deep learning. In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. JMLR.org, 2013. (ICML'13), p. III–1139–III–1147. Disponível em: <http://dl.acm.org/citation.cfm?id=3042817.3043064>.

TIELEMAN, T.; HINTON, G. *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. 2012. COURSERA: Neural Networks for Machine Learning.

TORRALBA, A.; FERGUS, R.; FREEMAN, W. T. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 30, n. 11, p. 1958–1970, Nov 2008. ISSN 0162-8828.

TURING, A. M. Computing machinery and intelligence. *Mind*, Oxford University Press on behalf of the Mind Association, v. 59, n. 236, p. 433–460, 1950. ISSN 00264423. Disponível em: <http://www.jstor.org/stable/2251299>.

WILSON, A. C.; ROELOFS, R.; STERN, M.; SREBRO, N.; RECHT, B. The marginal value of adaptive gradient methods in machine learning. In: GUYON, I.; LUXBURG, U. V.; BENGIO, S.; WALLACH, H.; FERGUS, R.; VISHWANATHAN, S.; GARNETT, R. (Ed.). *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017. p. 4148–4158. Disponível em: <http://papers.nips.cc/paper/7003-the-marginal-value-of-adaptive-gradient-methods-in-machine-learning.pdf>.

WILSON, D.; MARTINEZ, T. R. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, v. 16, n. 10, p. 1429 – 1451, 2003. ISSN 0893-6080. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0893608003001382>.