POLITECNICO DI TORINO

MASTER'S DEGREE PROGRAMME IN MECHATRONIC ENGINEERING

A.a 2021/2022

July 2022

MASTER'S DEGREE THESIS

Software compressive optimization of deep neural networks

AUTHOR

Diego García González

SUPERVISORS

Carla Fabiana Chisserini

Claudio Ettore Casetti

Acknowledgements

This project means a full stop in my academic life. All these years have involved a great effort that finally brings the rewards and comes to an end.

Firstly I want to thank Professor Carla Fabiana Chiasserini, Doctor Franceso Malandrino and Giuseppe Di Giacomo their implication in this work and their help. Without them this thesis would have been impossible to finish.

Thanks to my new and old friends. The Erasmus experience has allowed me to meet incredible people that I would like to keep in my life. Thanks Jamal for all those funny moments in Torino, thanks Felix for the talks and the nice moments at home, thanks Victor for making me feel like one more of the team and thanks Giulio for you generosity and your friendship. On the other hand, I also need to mention my people from Spain. My university friends, Vela, Miguel and David, you have been a great help during the pandemic while I was in Torino. And thanks to my second family, Mery, Marta, Cris and above all, Raúl and Dani, you have made me strong and feel capable of achieving anything, so a part of this thesis is also yours.

Last but not least, thanks to my real family. Thank you Álvaro for your advises, your help and your mentorship. A new time is coming for you and you need to enjoy in to the full. Thank you dad for all your support. Your help, commitment and love have made that, after eight years of university studies and some bad moments, I can get his wonderful achievement. And finally, thank you mum. You have always been the greatest support in my life, so I owe you everything. This thesis is dedicated to my family, because without them, I wouldn't be writing these words.

Abstract

Software compressive optimization techniques are based on the need of deploying complex and large deep neural network models in devices with low processing capacity. Their high efficiencies allow deep neural networks to reduce the cost of resources while maintaining a good performance and reliable results. In this project, two optimization techniques are studied, analysed and combined in order to maximize the compression of a state-of-the-art deep neural network model. The results achieved show the different approaches that can be followed and the great impact that these techniques have in the downsizing of large models.

Keywords: *deep neural networks; knowledge distillation; pruning; optimization; compression.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning [Mitchell, 1997] and more specifically, Deep Learning [Schulz and Behnke, 2012] appeared for the first time by the hand of Paul Werbos, who described for the first time the process of training an artificial neural network through backpropagation of errors in 1974 [Werbos, 1994]. Ever since then, researchers have focused on the creation and development of new deep neural networks that could accomplish better predictions or classifications [Li et al., 2021]. This improvement in performance always came together with larger and sometimes more complex neural networks. Although the accuracy of these neural networks was in many cases astonishing, its huge size often made them impossible to be used in common applications.

With the time, many new applications have been field of research in the deep learning field. Computer vision [Sinha et al., 2018], speech recognition [Roger et al., 2020], natural language processing [Torfi et al., 2020], machine translation [Yang et al., 2020] or medical image analysis [Litjens et al., 2017] are only examples of some of the fields that have experience the application of neural networks with surprising results. However, the use of deep neural networks is subject to severe constraints. A large deep neural network can have more than a hundred billion parameters that must be trained in data processing centres for days or even weeks. Therefore, common users, companies or researchers cannot afford such a huge amount of resources.

One of the technological fields that intrinsically has a lack of resources is the self-driving vehicles field [Huang and Chen, 2020]. Despite cars can have powerful onboard computers, the computations they must do usually require a very short time and thus, the use of large deep neural networks is non-functional. Nowadays, self-driving and non-self-driving car are supposed to have a fast response in the performance of image recognition and data computation, and still, maintain a good accuracy. Thus, the use of small and rapid neural networks is so important in this fields.

Along with self-driving vehicles, edge computing [Zhang et al., 2020] is becoming more important each day. This new paradigm, based on the location of small processing data centres in near locations, allows processing data much faster than the cloud paradigm thanks to the proximity of the servers. However, again, the resources that edge computing provide are limited and hence, the operations it treats must be very optimized.

More and more applications, paradigms and devices need to process data in an efficient and fast way, but common state-of-the-art deep neural networks cannot satisfy this need, since, despite being very accurate, they need a lot of resources.

Because of this problem, the researching community has put its effort in the optimization of large deep neural networks. Its aim is to reduce that size of the large state-of-the-art neural networks and their computing time while maintaining the accuracy. In this regard, it is possible to take advantage of the fact that these deep neural networks are usually over-parametrized.

Different compression techniques have appeared in the last years [Matsubara et al., 2021]. These techniques try to solve the problem of excessive size and computing time by optimizing the cost of models. Even though many compression methods are coming to light, the most important ones that have recently emerged are known as *quantization*, *neural network pruning* and *knowledge distillation*.

Quantization [Gholami et al., 2021] is a technique based on the change of the formats of the variables used by neural networks. It leverages the fact that most systems use `float32` variables to perform their computations, and converts these variables into the `int8` format. Doing this, neural networks experience a significant reduction in size since each variable occupies 4 times less in memory, and computations are faster.

Pruning [Molchanov et al., 2016] is a technique based on the removal of the least important parameters or nodes of a neural network. This technique leverages the fact that over-parametrized neural networks accept removing its most superfluous parts without having an impact on its performance.

Knowledge distillation [Gou et al., 2021] is a method based on the training of a small neural network from the results of the performance of a larger one. With this technique is possible to obtain reduced neural networks capable of mimicking the behaviour of a bigger one.

This work is focused on the study and analysis of the neural network pruning and knowledge distillation techniques. Several experiments will be done in order to obtain results of the performance of the methods and finally, both will be combined in an attempt to achieve an effective compression of a neural network along with a reliable performance.

Along this project a thorough explanation of Deep Neural Networks and some possibilities for their compressive optimization will be shown. In Chapter 2, an overview of the structure and functioning of Deep Neural Networks (DNNs) is given. Furthermore, it is included an introduction of the software DNN compressive optimization techniques that are studied throughout the project. Chapter 3 is divided in two main sections that explain the two techniques analysed in this work. In Section 3.1 the technique of Knowledge Distillation is explained along with different experiments and in Section 3.2 the technique of Neural Network Pruning is presented and analysed along with its experiments. In the Chapter 4 the results of the experiments and their explanations are discussed. Finally, in the Chapter 5 a summary of the conclusions of the project is given and approximation to a future work that can mean a continuation of this project is introduced.

Chapter 2

State of the Art

In this Chapter all the different elements that are related to this project are described. First of all, we are explaining the definition and composition of Artificial Neural Networks (ANNs) and Deep Neural Networks (DNNs). Afterwards, we are going to deepen from DNNs to Convolutional Neural Networks (CNNs) and their importance in image recognition. Then, we are going to explain the software environment that we have used to perform this project, this is, Pytorch, and the libraries in which we have focused the most. Finally, we are going to explain the Simplify library of Python, which is a key part of this project.

2.1 Artificial Neural Networks

Over the last twenty years great investments and continuous researching on Artificial Intelligence (AI) have lead to an astonishing development of this field. Artificial Intelligence can be defined as the combination of algorithms whose final purpose is the performance of tasks that normally would require human intelligence. AI can be splitted into many different branches among which we find the so called Artificial Neural Networks (ANNs) or Neural Networks (NNs). ANNs are computing systems that try to mimic the behavior of human neural networks. NNs are based on the connection of individual neurons (also called nodes or perceptrons) which are generally formed by four different elements: the input, the weights, the bias and the outputs (Figure 2.1). In fact, they are called neurons because they copy the form of the human neurons, where the inputs are based on the dendrites, the node is based on the nucleus and the output is based on the axon.

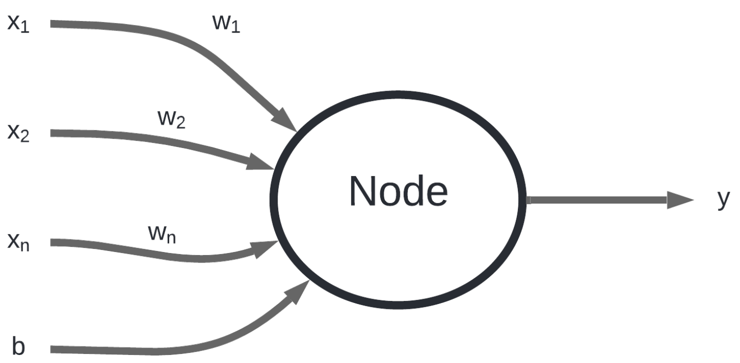


Figure 2.1: Basic structure of a node where *xi* are the inputs, *wi* are the weights, *b* is the bias and *y* is the output.

The basic structure of a node is compounded by two parts:

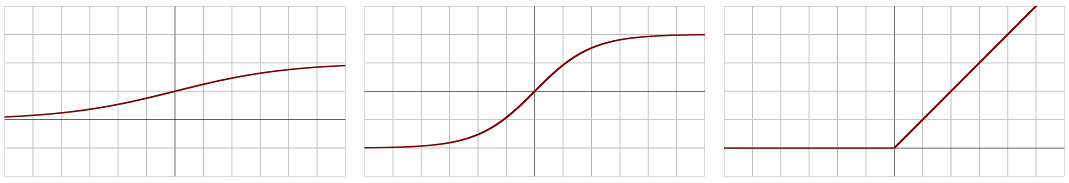
1. Firstly, a linear combination with the form of the equation 2.1, in which y is the output, x_i are the inputs, w_i are the weights and b is the bias. The output of this equation is called *logit* or *score*.

$$logit = (w_1x_1) + (w_2x_2) + ... + (w_nx_n) + b \tag{2.1}$$

2. Secondly, the output obtained in the previous step is passed through a function, called activation function, in order to achieve a non-linear output in the continuous space. Therefore, the final output of a node has the form of the equation 2.2.

$$y = g((w_1x_1) + (w_2x_2) + ... + (w_nx_n) + b) \tag{2.2}$$

The most typical activation functions used in Neural Networks are the sigmoid function (Figure 2.2a), the hyperbolic tangent function (Figure 2.2b) and the Rectified Linear Unit (ReLu) function (Figure 2.2c).



(a) Sigmoid function. (b) Hyperbolic tangent function. (c) ReLu function.

Figure 2.2: Most typical activation functions.

The connection of the output of different neurons to the input of others creates the Artificial Neural Network. When different neurons receive the same inputs and their outputs are the inputs of same other nodes, we can call this group of neurons as a layer. The first layer of the ANN is always called the input layer, since it represents the inputs, and the last layer is called the output layer. This last layer must have the same number of nodes as possible targets our NN can have. The rest of the layers between the input and the output layers are called *hidden layers*. Therefore, the minimal representation of an ANN have at least one input layer, one hidden layer and one output layer, as we can see in Figure 2.3.

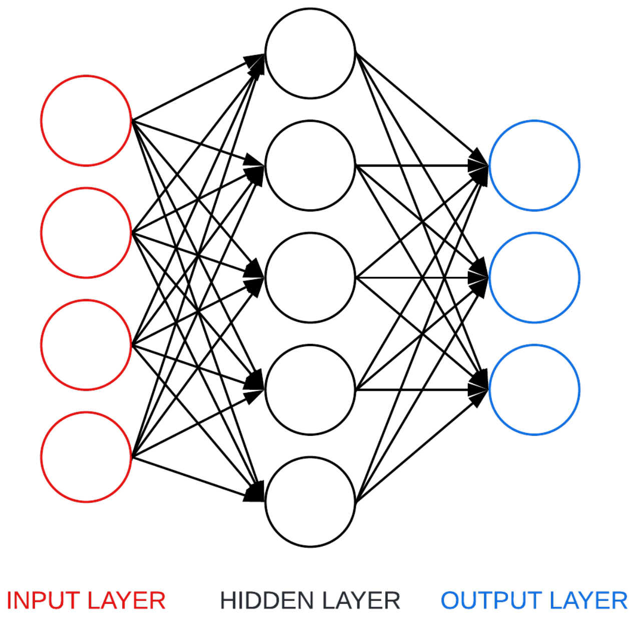


Figure 2.3: Representation of an Artificial Neural Network.

Depending on the number of nodes that the output layer has, the ANN can be meant to solve two kind of problems:

1. If the output layer contains only one node, the NN will be focused on a prediction problem.
2. If the output layer contains more than one node, the NN will solve a classification problem.

It is important to mention that in classification problems, the activation function used by the nodes of the output layer must take into account the contribution of all the output nodes, since the output probabilities of each node will be dependent each another. Thus, we cannot use a Sigmoid or ReLu functions.

In this case, the activation function used by the output layer is called *Softmax function* and it has the form of the equation 2.3, in which z_i are the logits of the different output nodes and y_i is the final output probability.

$$y_i = \frac{e^{z_i}}{\sum_{i=1}^{n} e^{z_i}} \tag{2.3}$$

Because of computation requirements, Neural Networks usually have more than one hidden layer. In this case, they are called Deep Neural Networks (DNNs). In the next subsection we are explaining the capabilities and power of DNNs.

2.1.1 Deep Neural Networks

As we explained in the last section, Deep Neural Networks are ANNs which have more than one hidden layer (Figure 2.4). The number of hidden layers is a parameter defined by the programmer to get the best performance in a computation process. There are DNNs with a very reduced number of layers and nodes, as the famous LeNet [LeCun et al., 1989], and huge DNNs with a large amount of parameters, for example, VGG16 [Simonyan and Zisserman, 2015].

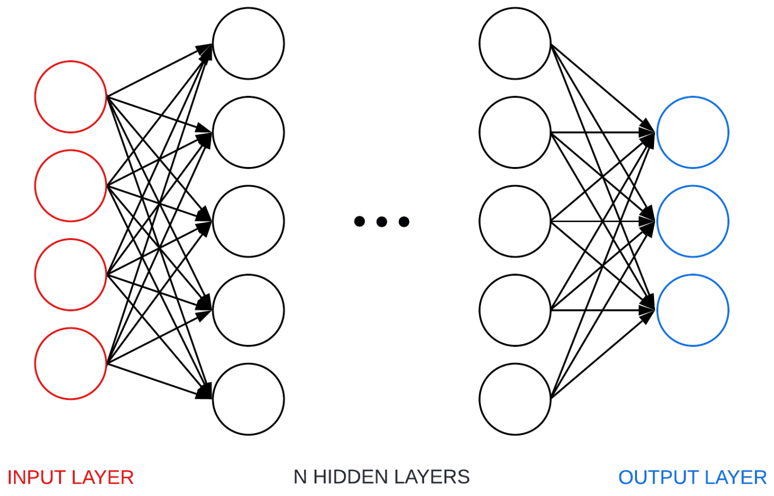


Figure 2.4: Representation of an Deep Neural Network, with N hidden layers.

The size of the DNN depends on the requirements that the computation process has. In the case of having a reduced amount of available memory or limited computation time we will choose a smaller DNN. However, when requirements are related to accuracy in the performance of the DNN we will normally choose a larger DNN.

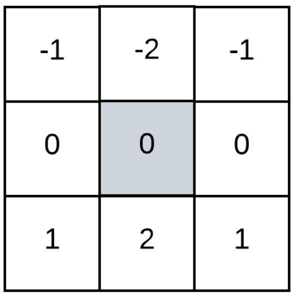
There are several kinds of DNNs that are used for different applications. This is, some DNNs work better in some applications than others. For example, Recurrent Neural Networks are frequently used for the field of Natural Lenguage Processing. But in this project, we are going to focus on Convolutional Neural Networks (CNNs), which are used in a wide range of applications mostly related to image recognition.

2.1.1.1 Convolutional Neural Networks

Convolutional Neural Networks are NNs specialized in extracting features from related data, most typically pixels of an image. They are normally composed of two parts:

1. The first part of the CNNs are the convolutional layers, which are in charge of extracting the image features.
2. The second part are the fully connected layers, whose purpose is to obtain relations from the image features previously extracted.

Convolutional layers work in a way in which each data (pixel) of an image is related to the ones next to it. This is done thanks to the so called *kernel*. The kernel assigns weights to the surrounding data in the way that is shown in the Figure 2.5. In order to obtain an output related to the data of the middle of the kernel (the grey square) we would multiply the weights of each data by its value. As we can see, the kernel shown in the Figure 2.5 is designed to find and enhance horizontal edges and lines in an image.



-1	-2	-1
0	0	0
1	2	1

Figure 2.5: Representation of the weights assigned in the kernel.

The kernel passes through all the pixels of an image obtaining an output in which the features of an image can be found out.

However, for each convolutional layer we do not have only one kernel. It is the job of the programmer to choose the number of times that different kernels passes though and image to obtain new features. The obtained outputs create new sets of data that are called *filters*. Therefore, we can not talk of nodes in convolutional layers, but of filters. Each filter is responsible of finding a pattern in the image. After computing the outputs, which in reality are the logits, these values are usually passed through a ReLu function.

After each convolutional layer the data must be filtered and for that, we use the *pooling layers*. As previously said, pooling layers take convolutional layers as input. Their aim is to reduce the size of feature maps (filters). There are two types of pooling layers: the max pooling layer and the average pooling layer. However, in this project we will focus on the max pooling layer, that works by taking a stack of feature maps as input and reduces its size by removing the least important parameters. We can see a representation of this process in the Figure 2.6.

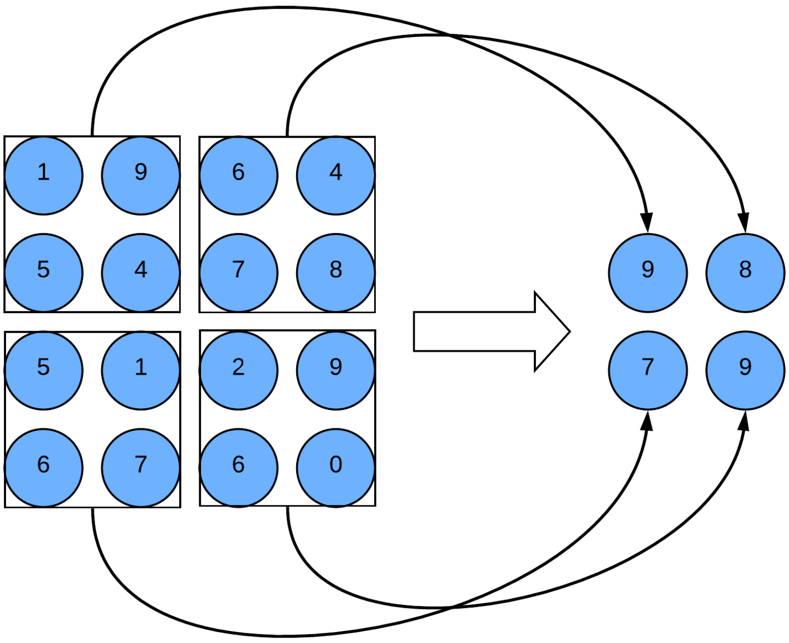


Figure 2.6: Representation of the max pooling process.

After the convolutional and pooling layers, as said, the data goes through one or more than one fully connected layers. These layers (that have been slightly explained in the Section 2.1) are formed by nodes that receive the data as inputs from all the nodes of the previous layer and send their outputs to all the nodes of the following layer.

Therefore, convolutional, pooling and fully connected layers together form the so-called CNN as the Figure 2.7 shows.

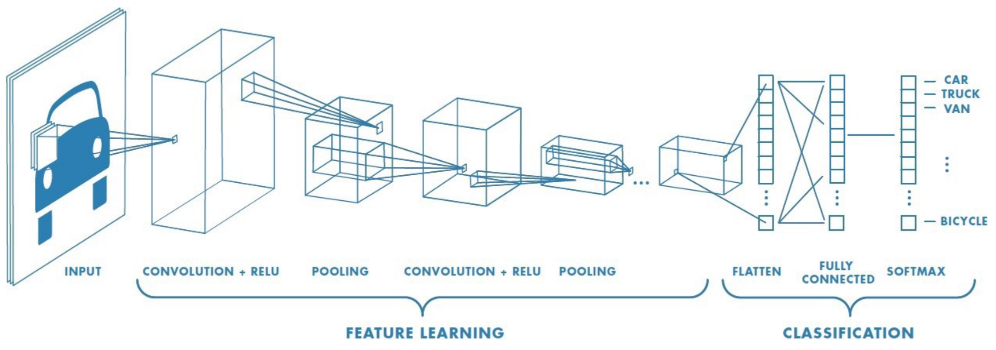


Figure 2.7: Representation of a CNN.

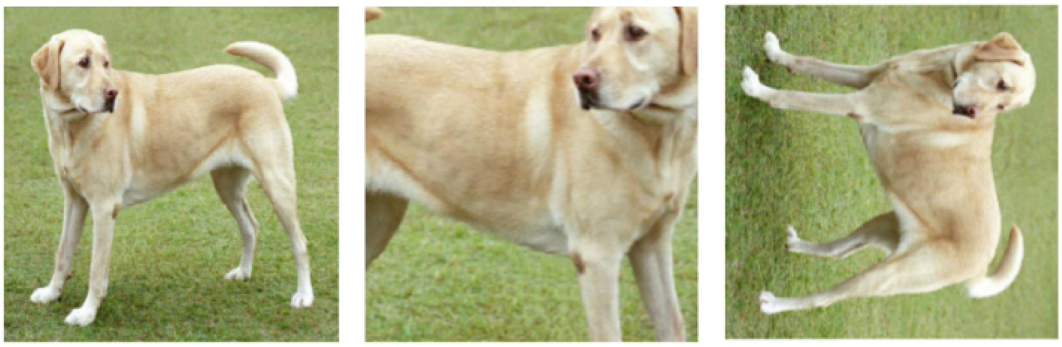
After seeing the structure of the NNs we are going to focus on how they work. As we have said, the ultimate aim of NNs is usually to predict or classify. However, before being able to do this, they must be trained. Next, we are going to explain the training process of the Neural Networks.

2.2 Training process of Neural Networks

The training process of NNs is usually composed of two or three stages, which are the training optimization, the validation and optionally, the testing. In this project we are focusing on the first two processes, but we will also briefly explain the testing process.

The first step in this process is the treatment of the data that we are using to train our model. Depending on the kind of data (images, text characters, voice inputs) there are different techniques to treat them. For this work, we will focus on the treatment of images as data.

Image data can have many different ways of treatment. The most common are normalization and augmentation. Normalization is a technique that requires an input with two arguments: the mean and the standard deviation. Its main purpose is to standardize the modules of all the data of an image. On the other hand, augmentation is a technique basically used in the training of the Neural Network. The purpose of this technique is to randomly modify the images by inverting, rotating or moving them. This will avoid the so-called *overfitting*, which is a problem that we will explain next. There are other techniques as cropping and resizing that as well as being used to process the image, they standardize the size of images that are the inputs to a NN model. In the Figures 2.8 we can see some of the different techniques to process the input data.



(a) Original image. (b) Cropped, resized and inverted image. (c) Rotated image.

Figure 2.8: Different techniques of image processing.

After processing the input data, the following process is the called feed-forward or inference. In this process, the data passes through all the defined operations in the NN model. From this process, in a classification problem (after the softmax function), we obtain the probabilities of each output to fit with the target. These probabilities, of course, are values from zero to one.

The objective in the training is to maximize the product of all the output probabilities that fit with the required target. However, this product is an operation that may cause problems in its computations, since it deals with very small floating point values. To manage these problems we use a powerful method called crossed entropy. It is based on the use of logarithms to turn the probability products into sums, which greatly facilitates the computation process. Therefore, using the minus logarithm of the probabilities, the new objective is to minimize the sum of the cross entropy.

Since cross entropy is the new operation to minimize, we can also treat it as an error function.

In the Figure 2.9 we can see how this process works. The values of the operations are the probabilities of the appropriate outputs to fit with the targets. To obtain the error function (Cross Entropy) we apply the minus logarithm to each probability. Finally, our objective is to minimize this function.

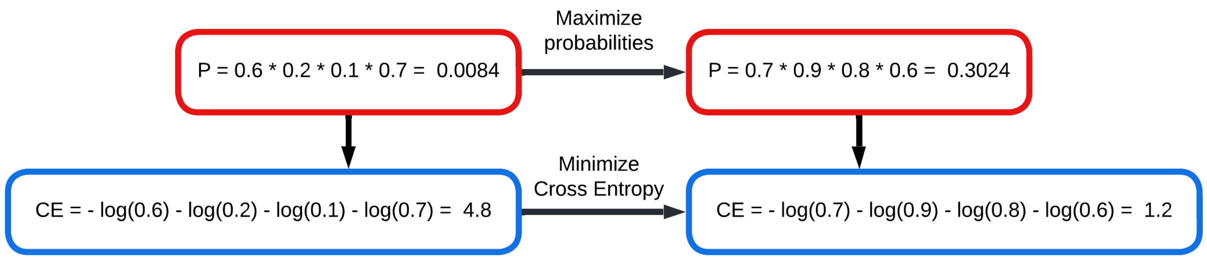

Maximize probabilities
P = 0.6 * 0.2 * 0.1 * 0.7 = 0.0084
P = 0.7 * 0.9 * 0.8 * 0.6 = 0.3024
Minimize Cross Entropy
CE = - log(0.6) - log(0.2) - log(0.1) - log(0.7) = 4.8
CE = - log(0.7) - log(0.9) - log(0.8) - log(0.6) = 1.2

Figure 2.9: Computation and minimization of the Cross Entropy.

The next step of the training is the minimization of the error function. There are different algorithms to do it, but the most common is the so-called *Gradient Descent*. Using this algorithm, our goal is to calculate the gradient of the error function E, at a point $x = (x_1, x_2, ..., x_n)$, given the partial derivatives of the equation 2.4.

$$\nabla E = (\frac{\partial}{\partial w_1}E, \frac{\partial}{\partial w_2}E, ..., \frac{\partial}{\partial w_n}E, \frac{\partial}{\partial b}E) \tag{2.4}$$

In which w_i, b are the weights and biases (parameters) of the NN model.

Taking the appropriate derivatives, t can be easily demonstrated that the computation of the Gradient Descent algorithm leads to an update of the model parameters of the form showed in the equations 2.5 and 2.6.

$$w_i\prime = w_i + \alpha[(y - \hat{y})x_i] \tag{2.5}$$

$$b\prime = b + \alpha(y - \hat{y}) \tag{2.6}$$

In the equations 2.5 and 2.6 we introduce the new parameter α, called learning rate, which is the constant in charge of the amplitude of each Gradient Descent step. It is also important to mention that these equations are specifically related to NNs with sigmoid activation functions, since it is well known that the derivative of this function has the form of the equation 2.7.

$$\sigma\prime = \sigma(x)(1 - \sigma(x)) \tag{2.7}$$

Having a first insight of the updating of parameters with the Gradient Descent algorithm we must understand how these derivatives can propagate through the NN. This process is called *backpropagation*. Backpropagation is a powerful tool that allows updating all the parameters of the NN by using the Gradient Descent algorithm and the chain rule.

Having the example of a NN as the one shown in the Figure 2.10, we can understand the error function as a function of all the weights of the model.

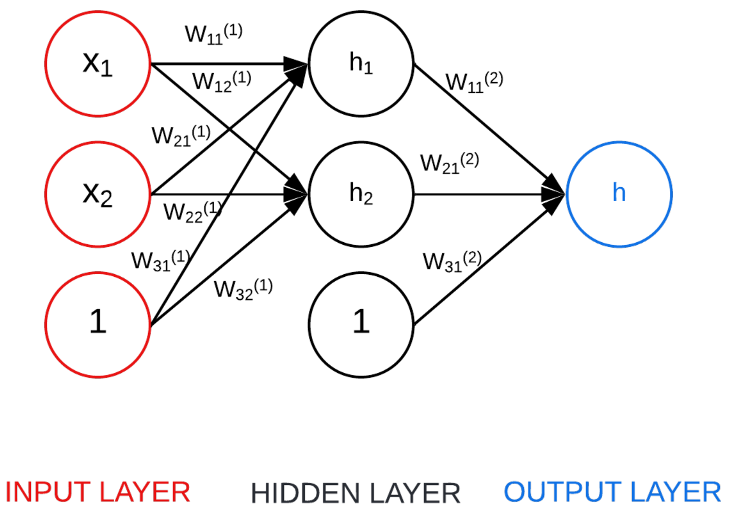


Figure 2.10: Representation of a NN with all its weights.

Knowing this, to minimize this function we need to calculate its gradient, which has the form of a matrix as the one shown in the equation 2.8.

$$\nabla E = \begin{pmatrix} \frac{\partial E}{\partial W_{11}^{(1)}} & \frac{\partial E}{\partial W_{12}^{(1)}} & \frac{\partial E}{\partial W_{11}^{(2)}} \\ \frac{\partial E}{\partial W_{21}^{(1)}} & \frac{\partial E}{\partial W_{22}^{(1)}} & \frac{\partial E}{\partial W_{21}^{(2)}} \\ \frac{\partial E}{\partial W_{31}^{(1)}} & \frac{\partial E}{\partial W_{32}^{(1)}} & \frac{\partial E}{\partial W_{31}^{(2)}} \end{pmatrix} \tag{2.8}$$

Although the element of the matrix 2.8 in which ∂W has superscript 2 can be computed by using the Gradient Descent algorithm, the rest of the element cannot. To calculate these elements we need to use a mathematical tool called the *chain rule*.

The chain rule states that if we have a function h that is a composition of two functions of x, where $z = f(x)$ and $h = g(f(x))$ or $h = g \circ f$, then, its derivative can be defined as the equation 2.9.

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial z}\frac{\partial z}{\partial x} \tag{2.9}$$

Therefore, with the Gradient Descent algorithm and the chain rule, it is possible to compute all the elements of the matrix 2.8. We can find out the variations that each parameter must have in our NN model in order to minimize the error function. As an example, the equation 2.10 would calculate the optimal variation of E with respect to $W_{11}^{(1)}$. In this equation, E represents the error function, $\hat{y}$ represents the prediction or output of the model, h and h_1 represent the outputs of the respective activation functions (see Figure 2.10) and $W_{11}^{(1)}$ represents a weight.

$$\frac{\partial E}{\partial W_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial W_{11}^{(1)}} \tag{2.10}$$

After the first feed-forward and backpropagation processes we obtain an optimized model. At this point, it is time to validate our model. The validation is a process in which an inference of new data is performed through the optimized model with the purpose of calculating the accuracy of the model. There are two ways of computing this accuracy:

1. Calculating the average loss (from the error function) of all the outputs corresponding to all the data.
2. For each data, comparing the most probable output with the correct target.

The training optimization and the validation processes must be repeated until the model reaches a good performance, this is, the training and validation losses are low and the validation accuracy is high. Each iteration that this process repeats is called an *epoch*. The programmer must find the perfect number of epochs to iterate the training process. Too many epochs would lead to a phenomenon that has been mentioned in the section 2.1, called *overfitting*.

Overfitting is a phenomenon for which the NN model behaves too good for the training data set but does not fit properly the general data. There are some techniques to prevent overfitting. We have already mentioned one, that is the data augmentation. However, the most common techniques are the so-called *Dropout* and *Early Stopping*. We are explaining them next.

Dropout

Sometimes one part of the network has very large weights and it ends up dominating all the training, while another part of the network does not get trained. Dropout [Srivastava et al., 2014] is a technique that consists of turning the dominating part off and letting the rest of the network train.

More thoroughly, as we go through the epochs, we randomly turn some nodes off avoiding the feed forward process to pass through them. Doing this, the other nodes take more part in the training.

The Figure 2.11 represents a DNN with two hidden layers of four nodes in which there is a 50% of probability of dropout. As we can see, two random nodes of each layer are turned off each epoch.

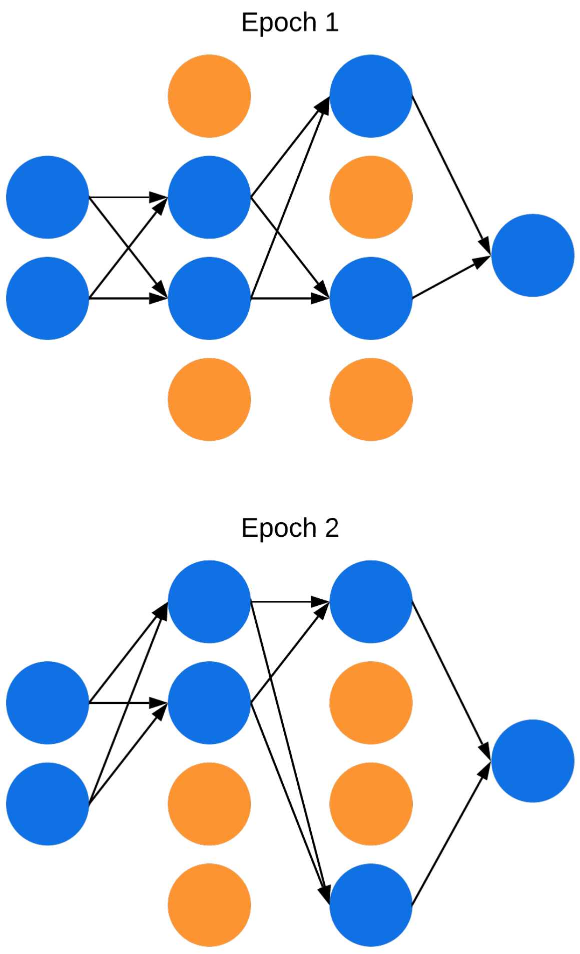


Figure 2.11: Inference process of a DNN with two hidden layers and 50% of dropout probability.

Early Stopping
Usually the training process is started with random weights in the first epoch, having a terrible behavior with very large validation and training losses. When we keep going through the epochs, the model is trained and fits the data much better. However, there is a point in which the more epochs we train the model, the better it fits the training data, but generalizes horribly. As a result, the training losses will continuously decrease but the

validation losses will increase.

The Figure 2.12 represents the so called model complexity graph, which measures the training and validation errors with respect to the epochs.

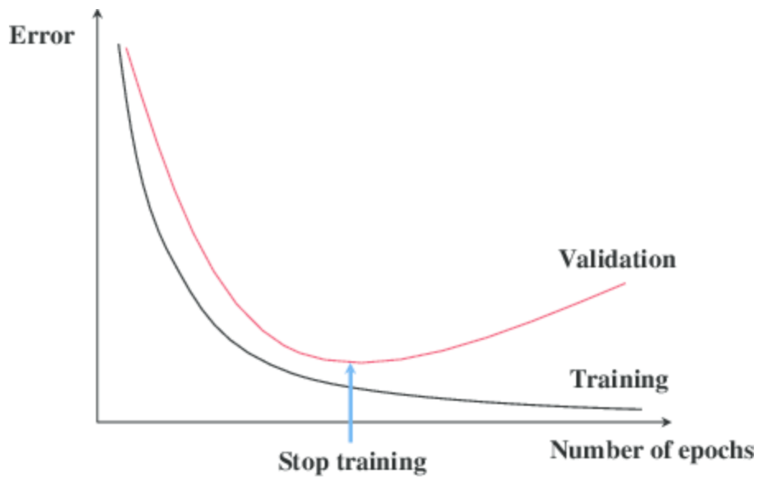


Figure 2.12: Example of model complexity graph.

As we can see in the Figure 2.12, the training curve is always decreasing, since, as we train the model, we keep fitting the training data better and better.

On the other hand, we can observe that at the beginning of the training the validation error is very large because the model is not exact (we call this *underfitting*). Then, it decreases as the model generalizes well until it gets to a minimum point, the so called *Goldilocks spot*. Finally, once we pass that spot, the model starts overfitting since it stops generalizing and just starts 'memorizing the training data'.

Therefore, the model complexity graph divides the training process into two parts: the underfitting part, when the training and validation losses decrease; and the overfitting part, when the validation loss starts increasing. These two parts are separated by the Goldilocks spot, that is the point where the model generalizes the best. Thus, this determines the number of epochs we must use in our training process.

This algorithm is called Early Stopping [Girosi et al., 1995] and it is very widely used to train neural networks.

2.3 PyTorch

Neural Network computation are linear algebra operations on *tensors*, a generalization of matrixes. For example, an RGB image is based on a 3-dimensional tensor.

PyTorch [Paszke et al., 2019] is an open source machine learning framework used for deep learning applications such as image recognition. It was primarly developed by the Facebook AI Research team. Although PyTorch interface is mainly developed and optimized for Python, it also has an interface for C++. Pytorch tensors are similarly used as Numpy [Harris et al., 2020] arrays but with some great benefits, as the graphics processing units (GPU) acceleration.

PyTorch provides different packages that facilitate the work with neural networks. Next we are explaining the process of the most important:

1. The package *torch* of Pytorch provides different modules such as *nn*, that allows building efficiently large neural networks; *autograd*, which records the operations accomplished in the inference process and replies them backwards to compute the gradients; and *optim*, which implements several optimization algorithms in order to perform the backpropagation of the NN models.

2. The package *torchvision* provides different data sets for the training, validation and testing processes of a NN model. With this package is also possible to process the downloaded data in order to adjust it to our neural networks needs.

As previously said, PyTorch is capable of using GPU acceleration thanks to the *Compute Unified Device Architecture (CUDA).* CUDA is a parallel computation platform that includes a compiler and a set of of development tools created by Nvidia.

CUDA leverages the large parallelism and high band-width of GPU memories in applications with high arithmetic cost. It is designed in a scalable way in order to be able to increase the number of computational kernels. This design contains three key points, which are the string group hierarchy, the shares memory and the synchronization barriers.

In practice, in PyTorch it is necessary to use the package *torch.cuda.* This package supports the so-called CUDA tensor types, which implement the same functions as CPU tensors but running on the GPU.

2.4 Transfer Learning

Once an overall review of the core of this project (DNNs) and the used programming environment (PyTorch) have been exposed, we are going to present the different methods and algorithms that have been used in the development of this work.

The most famous and common DNNs (VGG16, VGG19, Densenet121, AlexNet...) are typically trained in a massive data set with over one million images labelled in 1000 different categories. These DNNs are based on convolutional neural network architectures and once they are trained, they work astonishingly well as feature detectors for images that they have not been trained on.

The training process of these DNN models normally takes very long times, in the order of days, weeks or months, so the development of daily and common applications cannot afford it.

Transfer Learning [West et al., 2007] is a method that consists of the use of pre-trained DNN models to work on data sets in which they have not been trained on. As previously said , the pre-trained DNNs usually work very well as feature detectors. As we have seen in the section 2.1, feature detection is performed by the convolutional layers of the model. The first convolutional layers of the pre-trained model extract more general features from the data set and, this extraction of general features can also be applied to different data sets. Therefore, usually it is not necessary to train the firt layers of a pre-trained model.

However, when using pre-trained models it is normally necessary to adjust the numbers of nodes of the last layer in order to obtain the same number of possible outputs as targets has our data set.

A good practice to train pre-trained DNN models for our own data sets is to leave all the parameters of the convolutional layers of the DNN unchangeable and train the classification layers, this is, all the fully connected layers. Then, checking the results obtained by the first training of the pre-trained model we can adjust the number of layers to train to get a better performance.

In PyTorch, transfer learning can be done by the *torchvision.models* package. This package has multiple models that can be easily loaded and used in a project. With this package it is also possible to load pre-trained models in order to train them for a specific data set.

2.5 Knowledge distillation

Large machine learning and deep learning models are very common. They are mainly used for researching purposes, so training these models allows improving the state of the art knowledge. However, when it comes to a practical use, such big and complex models are difficult to deploy in edge devices, what prevents them from obtaining real world data sets.

Moreover, the majority complex deep learning modelling work focuses in the development of one or several machine learning models whose objective is to achieve a good performance in validation data sets, which in the end is not representative of the real world data.

Therefore, this friction between training and validation data leads to models that have a good accuracy on validation data sets but do not perform well when it comes to real world data.

Knowledge distillation [Hinton et al., 2015] is a technique that tries to overcome these difficulties by transferring the knowledge from a large model or collection of models to a smaller one that can be deployed on an edge device without significant loss in performance. This technique was primarily demonstrated by Bucilua and collaborators in 2006 [Bucilua et al., 2006].

Knowledge distillation is mainly based on two models:

1. The teacher, which corresponds to the large and complex model from which we want to extract the knowledge.
2. The student, which is a small model whose purpose it to mimic the behaviour of the teacher leveraging its knowledge to obtain similar accuracy.

There are different types of knowledge distillation that can be categorized in two main forms: *Response-based knowledge* and *Feature-based knowledge*. We are going to briefly define all of them, but in this project we will mainly focus on the Response-based knowledge.

1. The **Response-based knowledge** focuses on the output of the teacher model. This is, it compares the scores of the teacher model with the scores of the student model, getting a distillation loss function. In this way, the student model improves its performance by trying to make the same predictions as the teacher model.

A basic diagram of the Response-based knowledge distillation can be seen in the Figure 2.13.

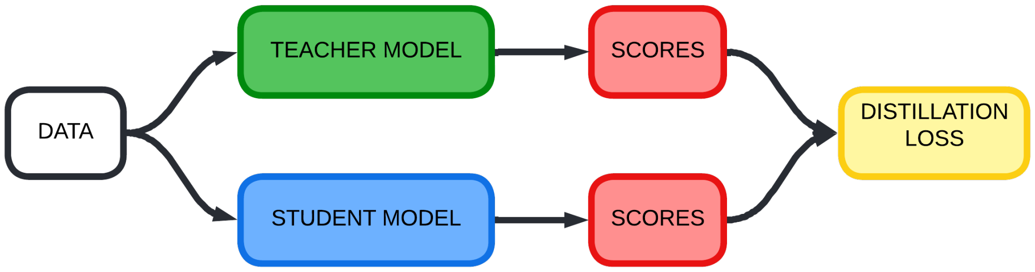


Figure 2.13: Diagram of the Response-based knowledge distillation process.

Regarding image recognition and computer vision tasks, as for example image classification, the inputs to compute the distillation loss are comprised of soft targets and not the scores. As seen in section 2.1, soft targets are usually calculated using the softmax function on all the possible scores, obtaining a probability distribution of the targets.

2. **Feature-based knowledge** is particularly suitable for deep neural networks. As we have seen in the section 2.1 each hidden layer of a model deals with specific features of the data. The final aim of this type of knowledge distillation is to make the student model mimic the feature activations of the teacher model.

   As we can see in Figure 2.14, in this case the distillation loss in computed by means of the outputs of specific layers.

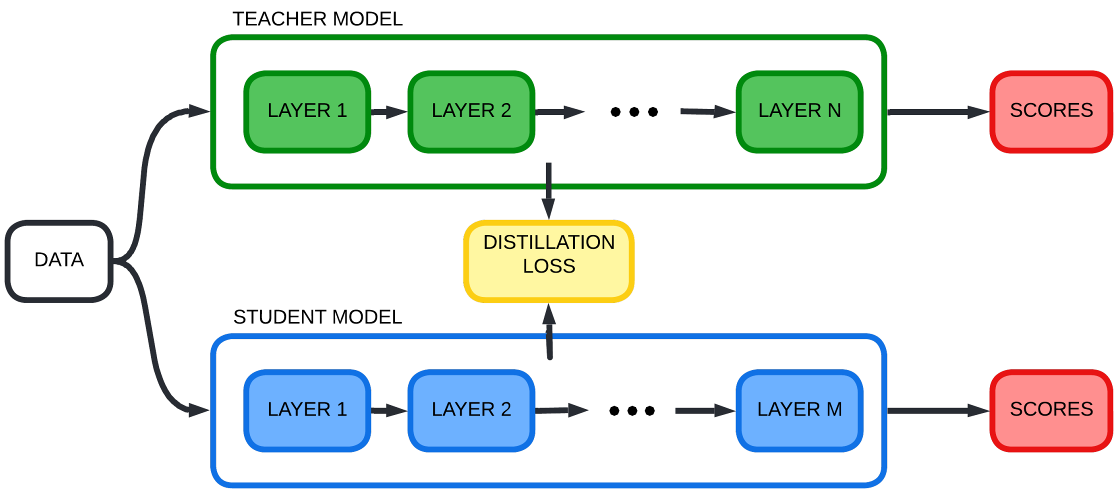


Figure 2.14: Diagram of the Feature-based knowledge distillation process.

As well as the different techniques of computing the distillation loss, there are different methods to train the teacher and student models.

Specifically, there are two main methods for training, which are called the *offline* and *online distillation*. Next we are explaining both, but in this project, the method that is used is the offline distillation.

1. **Offline distillation** is the most common method of training in knowledge distillation. In this technique, firstly, the teacher model must be completely trained, and afterwards, the student is trained on the basis of the knowledge of the teacher. Usually, both models are trained with the same training data set.

   Offline distillation is a well known technique in deep learning, since it is can be implemented very easily.

   In Figure 2.15 it is shown a simple diagram of the offline distillation training process, in which, firstly, the teacher model is trained and then, the knowledge is passed to the student.

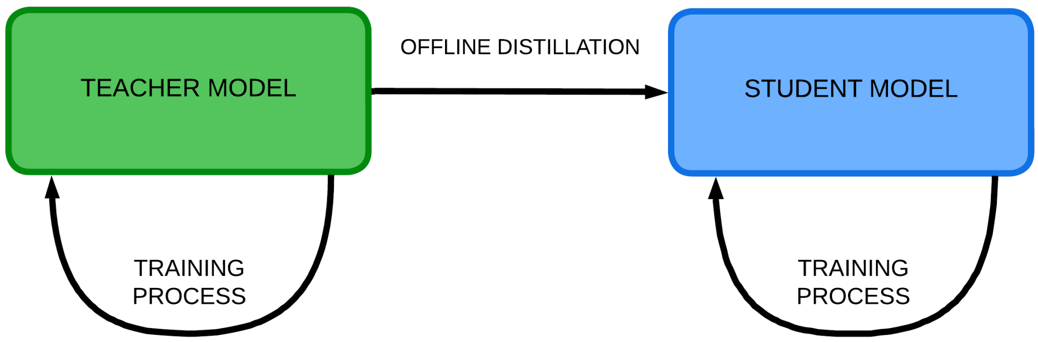


Figure 2.15: Diagram of the offline distillation training process.

2. **Online distillation**, on the other hand, is a technique in which both, the teacher and the student models are trained simultaneously in a single end-to-end training process.

   This way of training is particularly suitable when a pre-trained teacher model is not available for offline distillation. Moreover, it can be a highly effective technique thanks to the use of parallel computing.

   In Figure 2.16 it can be seen a diagram that shows the process of online distillation training. As we can observe, every epoch of the training process the state dictionaries (the parameters) of the teacher and student models are updated.

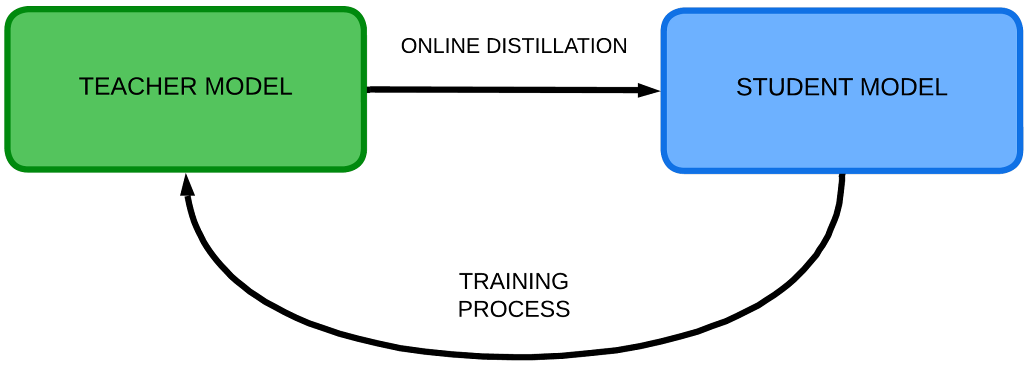


Figure 2.16: Diagram of the online distillation training process.

2.6 Neural Network Pruning

The cost of many neural networks in terms of computational power, memory or energy consumption can be huge and thus, unaffordable for most limited hardware. Yet, many fields can benefit from their use, so it is important to reduce their cost while maintaining their performance.

Hence, identifying techniques that permit model compression is important in these over-parametrized models in order to reduce the computational resources.

Neural network pruning [Riera et al., 2015] [Blalock et al., 2020] is a technique based on the idea of removing the least important parts of a network, achieving a reduction in cost of resources without a significant decrease in the performance of the model.

Pruning can be accomplished in several ways depending on its structure, criteria and the method that it follows. The different **pruning structures** differ in the parts of the model that are pruned. The **pruning criteria** refers to which way pruning should be applied to the model. The **pruning method** refers to the mode in which pruning is included in the process of obtaining a neural network model.

In the next subsections the different structures, criteria and methods will be explained. Eventually, an explanation of how pruning works on the PyTorch framework will be undertaken.

2.6.1 Pruning structures

Depending on the parts of the model that are pruned it is possible to differentiate between **structured** and **unstructured** pruning. Next, both structures are explained.

2.6.1.1 Structured pruning

Structured pruning [Anwar et al., 2015] focuses on large structures such as nodes in the case of fully connected layers, or filters in the case of convolutional layers.

One essential aspect that must be considered is that, when using this kind of pruning, as we are reducing the number of nodes or filters from the layers of a model, we are also removing the feature maps that are the outputs of such nodes and filters, and therefore, the inputs of the following layer.

This important factor can be understood with a simple example. Let us consider a convolutional layer with N_{in} input channels and N_{out} output channels. By definition, the convolutional layer has N_{out} filters with N_{in} kernels in each filter. Having this architecture, we can observe that when pruning a layer we are not only reducing the number of filters of that layer, but also reducing the corresponding kernels of the following layer.

This means that when using structured pruning, it must be expected a pruning of twice the percentage of parameters that would be thought in the first place. This is, if a 30% of a convolutional network is pruned, the first and the last convolutional layers will experience a reduction of 30% of the parameters, while the rest on the layers will have a reduction of 51% of the parameters. The first and the last layers will maintain the 70% of the parameters, whilst the hidden layers will keep the $(0.7 * 0.7) * 100 = 49\%$ of the parameters.

Therefore, when pruning filters or nodes it is necessary to consider calculating the number of actual pruned parameters.

2.6.1.2 Unstructured pruning

Unstructured pruning focuses on reducing the size of a model by removing parameters, such as weights or biases. Han et al. [Han et al., 2022] presented the basis of this kind of pruning, which has become one of the most widespread paradigms in the pruning literature.

Pruning connections has advantages such as its simplicity,as it is only needed to replace the value of a parameter with zero to prune a connection. Still, the most important advantage of the unstructured pruning is that it focuses on he most fundamental elements of a network, and thus, it permits removing a large amount of them without having a significant impact on the performance of the model.

However, this method has a major disadvantage. With most of the current frameworks, replacing the parameter tensors with zeros does not have an impact on the cost of the model, since they cannot accelerate sparse tensors' computation. To have a real impact, frameworks would need to modify the architecture of the network, but nowadays no framework can cope with this.

The Figure 2.17 [Tessier, 2021] represents the difference between unstructured and structured pruning in convolutional layers. While unstructured pruning is focused on pruning the parameters inside the kernel, structured pruning removes complete filters from the layer.

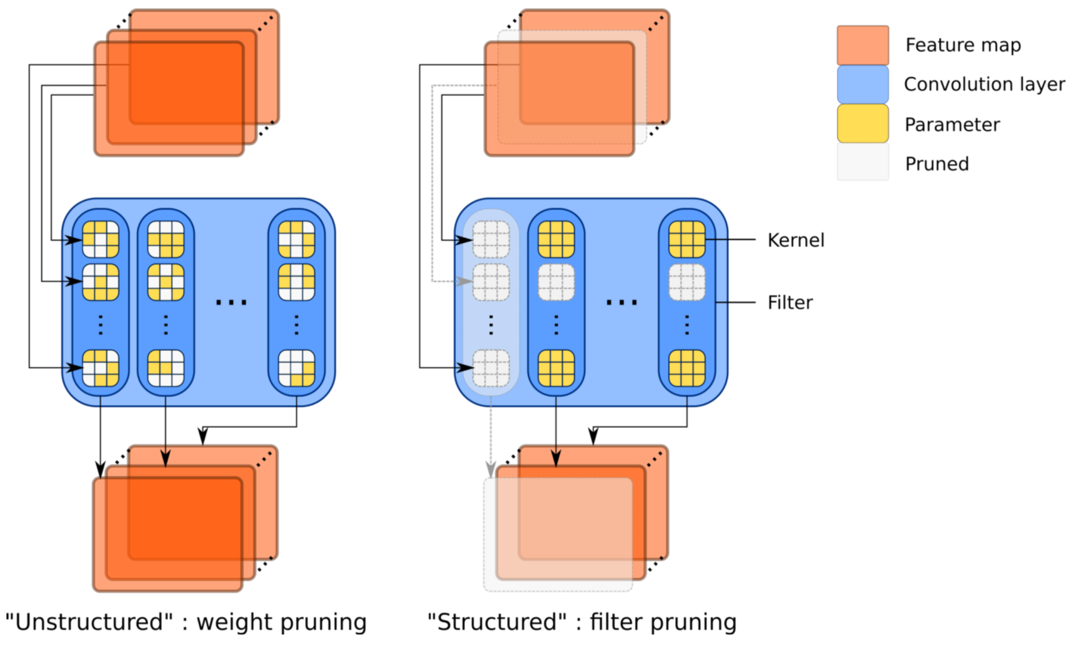

Feature map
Convolution layer
Parameter
Pruned
Kernel
Filter
"Unstructured" : weight pruning
"Structured" : filter pruning

Figure 2.17: Graphic representation of unstructured and structured pruning in a convolutional layer.

2.6.2 Pruning criteria

Once the pruning structures have been seen, it is important to understand that pruning can be applied to a model in two different ways: locally or globally.

Local pruning focuses the pruning process on each layer individually and independently for each other. On the other hand, global pruning is applied to all the parameters of the network at the same time.

Global structured pruning should not be used unless the norm is normalized by the size of the parameters, therefore global pruning must be limited to unstructured methods.

Figure 2.18 graphically represent local pruning on the left part and global pruning on the right. As it is shown, local pruning applies a pruning rate (in this case 50%) to each layer individually, while global pruning applies it on the whole network at once.

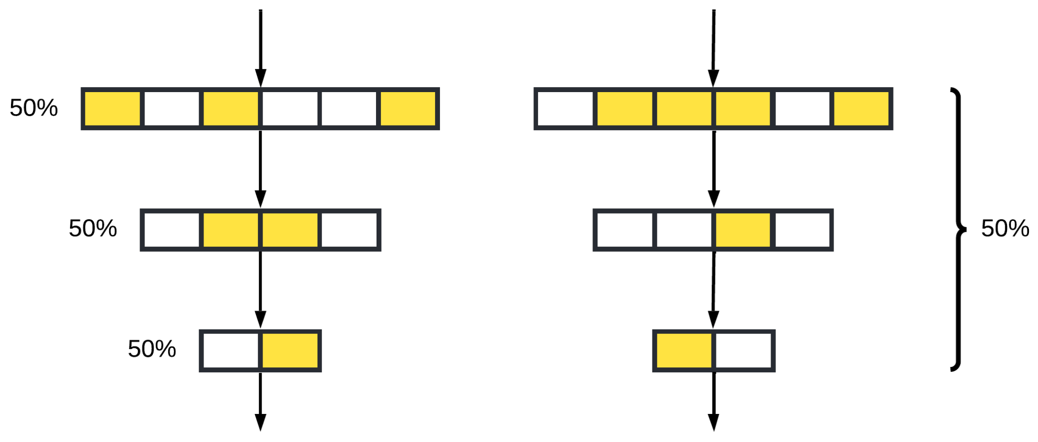


Figure 2.18: Graphic representation of local pruning (left) and global pruning (right).

2.6.3 Pruning methods

Once pruning structures and criteria have been seen, we are focusing on the pruning method. There are multiple pruning methods in the literature, since many papers bring their own. However, these methods tend to be customisations of other which are more general.

In this project we will work with two of the best known methods: the one-shot pruning and the iterative training-pruning. Next we are explaining them.

2.6.3.1 One-shot pruning

One-shot pruning [Chen et al., 2021] is a method based on the application of pruning on a DNN which is completely trained. The only objective of this method is to seek the reduction in size of the model, since it is already trained and therefore, there cannot be an improvement in training time.

With this method, the definition of pruning is directly applied on a DNN. In practice, one-shot pruning does no offer many benefits, but it can be an interesting method to figure out which layers of a model are the least important. Therefore, it is an interesting way to theoretically understand the performance of a DNN model layer by layer in order to take another actions afterwards.

As the one-shot pruning is done at the very end of the training process, there are no constrains in its use, and thus, it can be combined with the different structures and criteria.

2.6.3.2 Iterative training-pruning method

Iterative training-pruning [Paganini and Forde, 2020] is a method which aims to reduce the size of the model and its training time. Its performance is based on the fact that the model is pruned each iteration and hence, it becomes smaller and faster to train.

This method is specially adequate for those models whose training process must be optimized in time.

The Figure 2.19 represents the iterative process that training-pruning method follows for each epoch. It starts with the training of the model, continues with its pruning and finalizes with its validation.

This method is limited to structured local pruning techniques, since, as previously said, unstructured techniques can not improve the performance of pruned models in time.

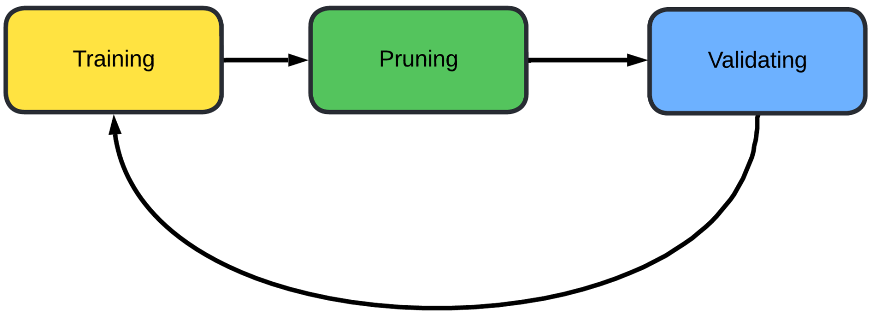


Figure 2.19: Representation of the iterative training-pruning process.

2.6.4 Pruning in PyTorch

The *torch* package of PyTorch includes a set of functions that allow the pruning of a model. In this section it is explained the functioning of these functions and their limitations.

The first task that must be accomplish when pruning in PyTorch is selecting the technique to prune our model. PyTorch offers different types of pruning in the *torch* package, for example, the `l1_unstructured`, `global_unstructured` or `l2_structured` methods, which perform a norm one unstructured pruning, a global unstructured pruning and a norm two structured pruning, respectively.

The pruning methods of PyTorch can only be applied on modules, this is, layers or filters. So the next task is to identify the modules of the model to prune and include them as the first argument in the pruning method.

Then, the next parameter which must be specified is the sort of parameter to prune within the module. This argument is normally completed using a string of the value *'weight'* or *'bias'*.

Finally, the last argument that must be filled is the percentage of parameters or modules that have to be pruned (if it is a number between 0 and 1) or the absolute number of parameters ro modules that should be pruned (if it is a positive integer higher than 1).

In PyTorch, it is possible to have access to the value of all the tensors of parameters that form a module and we can see if those tensors belong to a bias or a weight. When pruning a parameter or module, PyTorch fulfills two operations:

1. Firstly, it replaces the name of pruned elements with a parameter formed by adding *_orig* to the initial parameter name. This is, if *weight* or *bias* is the name of the parameter, the new parameter will be called *weight_orig* or *bias_orig*.

2. Secondly, it creates a buffer with the same size as the parameter in which the pruning masks are stored. This buffer will only contain zeros or ones depending on whether the corresponding parameter is pruned or not.

After a model is pruned, all its relevant tensors, including mask buffers and original parameters, are stored in the model's state dictionary and thus, can be saved easily.

However, if a permanent pruning is needed, it must be done by a re-parametrization. This can be performed by means of the `remove` functionality, which combines the mask buffers with the original parameters to create a permanent parameter in its pruned version.

The Figure 2.20 represents the pruning process performed by the pruning library of PyTorch. As it is seen, it is divided into two steps. Starting from an original weight parameter, a pruning function is used (in this case, unstructured with a scope of the 30%), obtaining the two attributes called *weight_orig* and *weight_mask*. Afterwards, the *remove* method is used to implement the permanent prune.

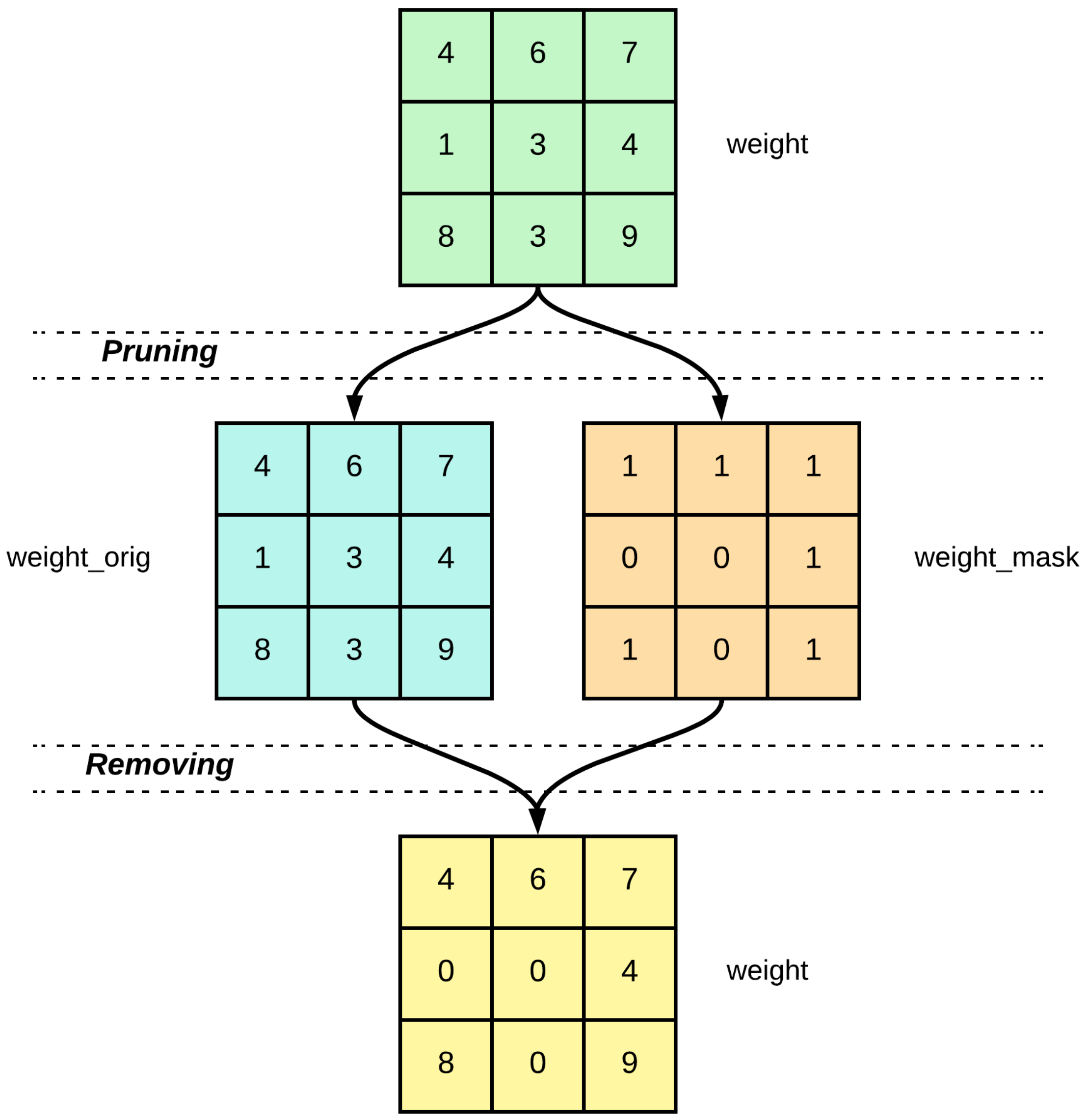


Figure 2.20: Representation of the pruning process in PyTorch.

However, there is a major drawback in the pruning library of PyTorch. When a module is totally pruned, it remains with the same size (same number of parameters) as at the beginning of the pruning process. The `remove` method does not literally remove the parameters that have been pruned from the model, it just makes them zero.

Currently, PyTorch does no have any way to remove those parameters from the model and solve this problem. Consequently, using PyTorch it is not possible to implement the definition of pruning in a model, since it will not experience any optimization.

2.6.5 The Simplify library

Simplify [Bragagnolo and Barbano, 2022] is a library created by Andrea Bargagnolo and Carlo Alberto Barbano whose purpose is to solve the main problem that pruning frameworks present.

As it has been said in the previous section, neural network pruning theoretically aims to reduce the models size and complexity. However, in practice, pruning offers few benefits since current frameworks are limited to converting the pruned weights into zero, and therefore not removing the pruned parameters.

Simplify allows the effective pruning of neural network model by removing its zeroed parameters, accomplishing the reduction of computing resources, such as memory footprint or inference time.

This library is only capable of pruning complete nodes or filters, so its scope is restricted to models in which structured pruning techniques have been performed.

Simplify provides a method which is very simple to use. This method, which called `simplify`, has two main arguments. The first one, is the model to simplify, which should have been previously trained and pruned. The second argument must be a tensor of zeros with the same size as the input of the model. Other arguments can be added, but it is optional.

Chapter 3

Compressive optimization of Deep Neural Networks

In this chapter, the work accomplished along this project is presented. The chapter includes two sections, which correspond to the two ways that we have studied to optimize Deep Neural Networks: knowledge distillation and neural network pruning.

All the computations have been run on a NVIDIA GeForce RTX 3060 Ti GPU with a 12th Gen Intel(R) Core(TM) i5-12600 CPU.

In the 3.1 section we started working with two different teacher models, the Densenet-121 and the VGG16. However, due to several problems with the Simplify library we discarded the Densenet-121 model, so this work only focuses on the VGG16 DNN. However, also because of the Simplify library, we needed to tune the VGG16 model to make it work properly.

In our experience, the `simplify` method does not accept models in which any of their layers have bias. Although Densenet-121 does not have bias in any of its layers, it includes some transition layers that, by default, create a bias and therefore, it is impossible to make it work. Furthermore, Densenet-121 is a very complex DNN, so it is difficult to tune its structure for the purposes of this project. On the other hand, VGG16 is a bigger but simpler DNN. Although its pre-trained model has biases in all its layers, it is possible to remove them and adjust the model to make it work with the `simplify` method. Hence, we have chosen VGG16 as teacher model for this work.

For the fulfilment of this project, we have used the data set CIFAR10, which includes

60000 32x32 coloured images distributed in ten classes. To use train the models with CIFAR10, firstly, the data need to process. To do it, the image data is transformed into tensors, normalized with a mean of $[0.485, 0.456, 0.406]$ and a standard deviation of $[0.229, 0.224, 0.225]$, resized to 256x256 and finally, cropped with a size of 224x224. With this pre-process, the data set is perfectly ready to be used with most of the pre-trained models available in PyTorch, and specifically, with VGG16.

Next, we explain all the stages that have been followed to perform a software compressive optimization of a VGG16 model.

3.1 Knowledge distillation

Knowledge distillation, as it has been explained in the section 2.5, is a technique that seeks to transfer the knowledge of a large DNN model to a smaller one. In this work, we have performed an off-line response-based knowledge distillation, since the training of the student is based on the outputs of a previously trained teacher model.

For the teacher, we have chosen a tuned VGG16 model and for the student, we have tried different customized models and chosen the one that performs better. Next, we are explaining all the processes that we have followed to accomplish the knowledge distillation from the tuned VGG16 model to the student.

3.1.1 The Teacher model

As previously said, the chosen Teacher model is the VGG16. This DNN is comprised of thirteen convolutional layers with a kernel size of 3x3. These convolutional layers are clustered in five groups, which are separated by max pooling layers with a kernel size of 2x2. The first group is formed by two layers of 64 filters, the second has two layers of 128 filters, the third one is comprised of three layers of 256 filters each and the four and fifth groups are compounded by three layers of 512 filters each.

Finally, VGG16 has three fully connected layers with 4096, 4096 and 1000 nodes. As we can see, VGG16 has 1000 outputs, since it is pre-trained with an image data set called ImageNet, which has 1000 different classes. The activation function used in all its convolutional and dense layers is the ReLu function. The Figure 3.1 shows the structure of the VGG16 DNN with all its convolutional and dense (fully connected) layers.

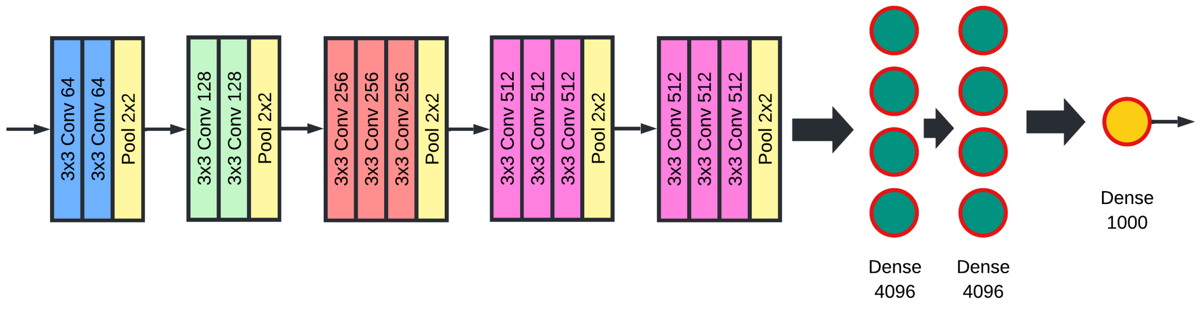


Figure 3.1: Graphic representation of the VGG16 DNN structure.

In our work, in order to train the VGG16 DNN as a teacher model it is necessary to tune it. To do so, we have fulfilled the next steps:

1. Firstly, we load the pre-trained VGG16 model with biases from the PyTorch set of models.
2. We define a customized VGG16 model with the same number of layers, filters and nodes, but without biases.
3. We change the number of nodes of the pre-trained model output layer to ten, so that it matches with the number of classes of the CIFAR10 data set.
4. We copy the state dictionary (parameters) of the pre-trained model into a variable.
5. We copy all the items of the state dictionary that are not biases into a new variable.
6. We change the names of the layers of the copied non-bias state dictionary in order to make them match with the names of the customized VGG16 model.
7. We assign the items of the non-bias state dictionary to the customized VGG16 model.
8. Finally, we delete the pre-trained model and the variables that store the state dictionaries in order to save memory space.

Once the model is tuned, it is prepared to be trained. However, as we have introduced in section 2.4, we do not need to train the whole model, since it is already pre-trained. Therefore, we must select the number of layers that must be trained, considering the final performance of the model. After some tests, the chosen layers to be trained are the three fully connected layers and the last five convolutional layers.

Besides, the Stochastic Gradient Descent is used as optimization function, with a learning rate of 0.001, a momentum of 0.9 and a weight decay of 0.0005. We also set the Cross

Entropy Loss as error function and finally, the number of epochs to 30. Once all the above steps are done, we are prepared to train the model.

The Figure 3.2 shows the complexity graph of the Teacher model. To train it and avoid overfitting, selected a 50% of dropout rate is selected and the technique of early stopping is used.

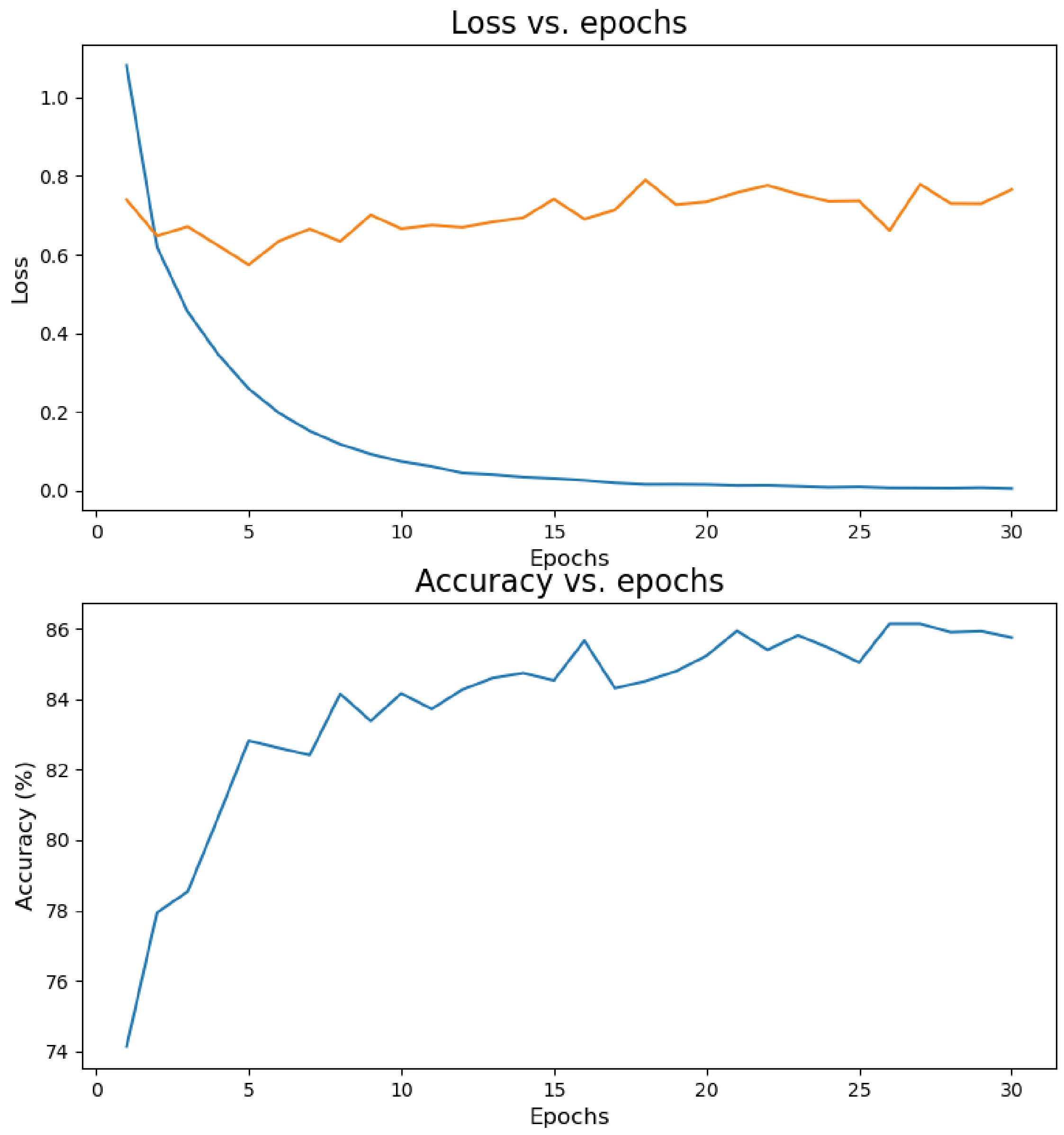


Figure 3.2: Complexity graph of the VGG16 teacher model.

During the training process, the time that it takes for the model to be trained each epoch is around 222 seconds.

The final teacher model is reached in the fifth epoch of the training process, archiving a training loss of 0.26, a validation loss of 0.57 and a validation accuracy of the 83%. The VGG16 Teacher model has a size of 524572 KB.

3.1.2 The student model

Once the teacher model is trained and selected, the next step is to define the student model based on the teacher results. To do it, we have created three different neural network structures that have been trained with result-based knowledge distillation in order to compare their performance.

As said before, the student is trained with respect to the results of the teacher. However, in order to optimize the training process we need to avoid the feedforward process of the teacher in each epoch. Thus, the inference of the teacher is done out of the training loop of the student and its results are saved in a variable. To do this, we must be sure that the data that the teacher and student models receive is not shuffled.

To train the student models we need to create a new error function that compares the output of the student with the output of the teacher, which is the target. This error function passes both outputs through a softmax function and then calculates the minimum square error between them. However, we must be aware that if we pass the outputs themselves, when the backpropagation is performed it is done for both models. Therefore, the inputs of the error function must be the output of the student model and the value of the output of the teacher model. In this way we achieve that the backpropagation is fulfilled only in the Student model. To have access to the value of the teacher output it is enough to add *.data* to the input variable. Another alternative to avoid the backpropagation is to set the Teacher model in evaluation mode.

In this case, we obviously need to train all the layers of the models. To do it, we select the Adam algorithm optimizer with a learning rate of 0.00005.

Model 1
The first model is the smallest of the three and its structure is formed by three convolutional layers and two fully connected layers. All the convolutional layers are separated by a max pooling layer and have a kernel size of 3x3. The first layer, which receives the input data, has 8 filters, the second is comprised of 64 filters and the last one is formed by 128 filters. The first and second fully connected layers have 100 nodes and 10 nodes, respectively. The Figure 3.3 shows a graphic representation of this structure. Although it

does not appear in the figure, the activation function used in all the layers of this DNN is the ReLu function.

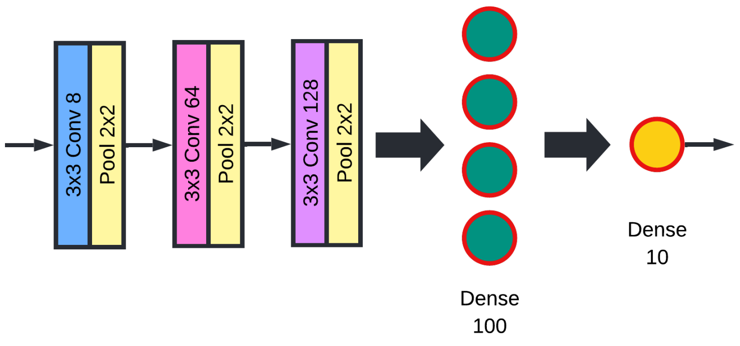


Figure 3.3: Representation of the structure of the student neural network 1.

This DNN has been trained for 200 epochs to obtain reliable results, using a dropout of the 55% and the technique of early stopping. Eventually, the final model is reached in the epoch 177, achieving a training loss of 0.0051, a validation loss of 0.0091 and a validation accuracy of the 61.5%. The Figure 3.4 represents the complexity graph of the Student model 1. It is interesting to notice that the training and validation loss are computed with respect to the outputs of the teacher model, but the validation accuracy is obtained with regard to the final targets. The training process of the Student model 1 has taken 58 seconds for each epoch and has achieve a model with a size of 2763 KB.

Model 2

The second neural network that is proposed as student is larger than the first one. In this case, the DNN is formed by five convolutional layers separated by max pooling layers and four dense layers. As it is shown in the Figure 3.5, the first convolutional layer has 8 filters, the second 16 filters, the third has 32 and the fourth and fifth, 64 and 128, respectively. On the other hand, the DNN has two fully connected layers of 100 nodes, one of 50 nodes and finally, the output layer with 10 nodes. In this case, all the activation functions of the different layers are also ReLu functions.

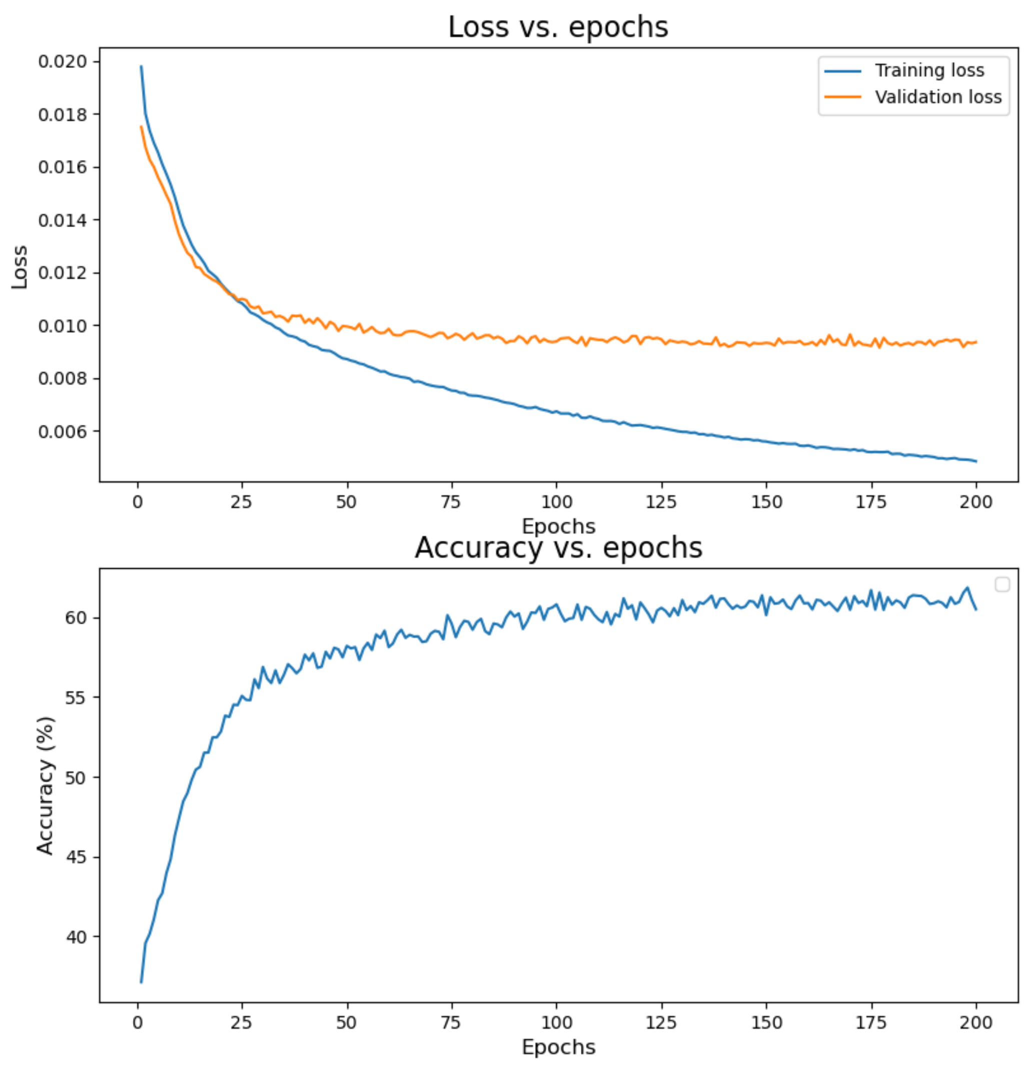


Figure 3.4: Complexity graph of the Student 1 model.

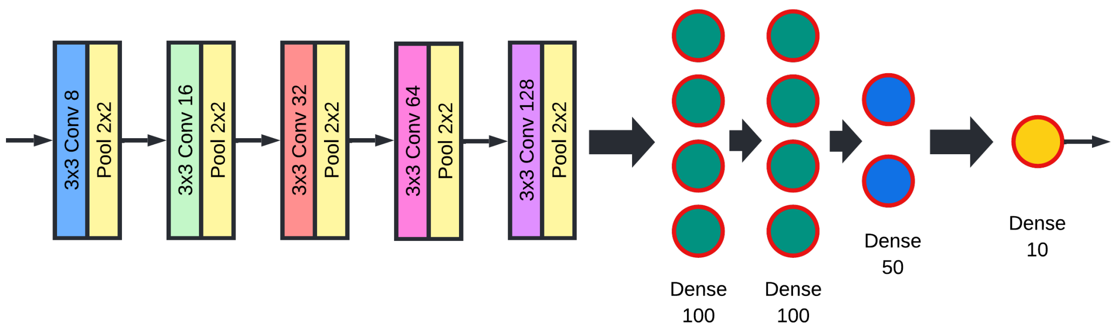


Figure 3.5: Representation of the structure of the student neural network 2.

This DNN has been trained for 100 epochs with a dropout rate of 55% and using the technique of the early stop. As it is shown in the Figure 3.6, the complexity graph indicates that the final Student model 2 is reached in the epoch 91, obtaining a training loss of 0.0064, a validation loss of 0.00911 and a validation accuracy of the 62%. Just like the Student model 1, the training and validation losses shown in the Figure 3.6 are computed with respect to the teacher model outputs, and the accuracy, with respect to the final targets. Each epoch of the training process of the model 2 has taken 44 seconds; and the model obtained has a size of 2897 KB.

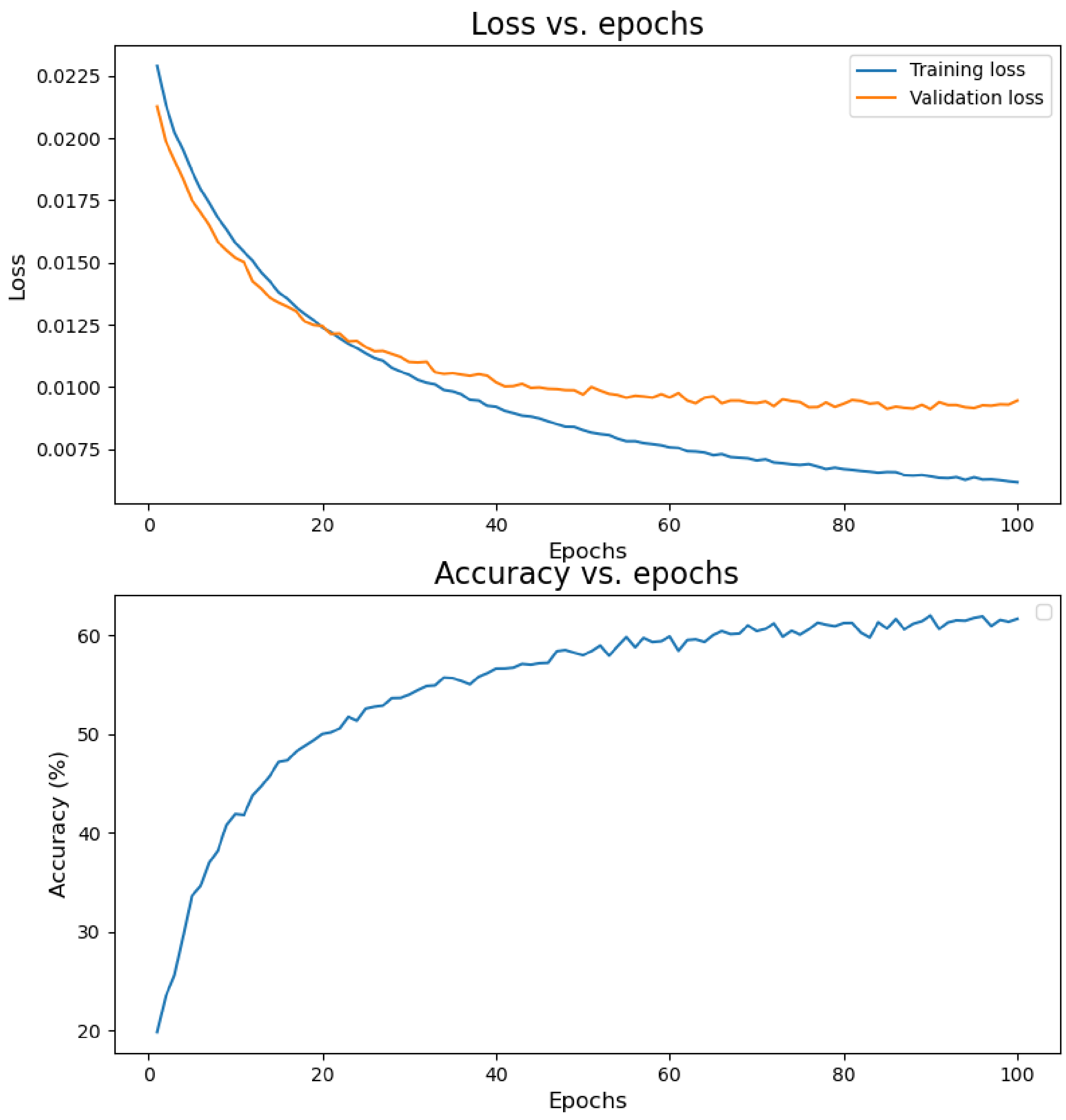


Figure 3.6: Complexity graph of the Student 2 model.

Model 3
The DNN of the Model 3 is the largest of the three proposed models. As it can be seen in the Figure 3.7 this DNN is comprised of seven convolutional layers and five dense layers. The convolutional layers are divided in five groups which are separated by max pooling layers. The first group is formed by a layer of 8 filters, the second by a layer of 16 filters, the third by a layer of 32 filters, the fourth group is comprised of two layers of 64 filters and finally, the last group is formed by two layers of 128 filters. After the convolutional layers, the DNN has a sequence of three fully connected layers of 100 nodes, followed by a dense layer of 50 nodes and finally, the output layer with 10 nodes. Just like the previous Models, the activation function used by all the layers is the ReLu function.

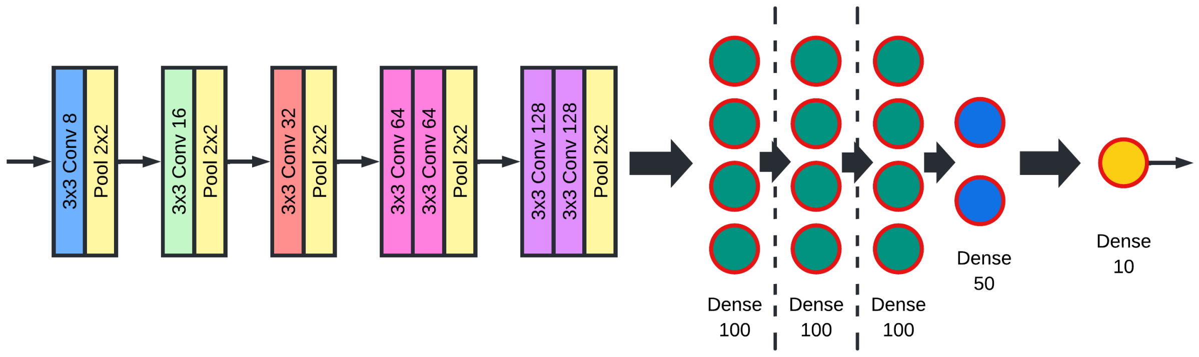


Figure 3.7: Representation of the structure of the student neural network 3.

This DNN has been trained for 300 epochs with a dropout rate of 55% and using the technique of early stopping. In the Figure 3.8 it is shown the complexity graph of the Student model 3, in which it can be appreciated that the Student model 3 is reached in the epoch 269, having a training loss of 0.0047, a validation loss of 0.0074 and a validation accuracy of 66.9%. In the Figure 3.8, around the epoch 200, we can observe a little step in the training loss that is due to the need to perform the training process of the model in two batches. Computing the Student model 3 takes a training time of 48 seconds for each epoch, obtaining a model with a size of 3657 KB.

Having the data of the three models we need to choose one of them to continue with the software optimization. Depending on our final aim there are different criteria that can be followed in order to select the appropriate model, such as the accuracy, the size or the training time. The main results of the student models are the following:

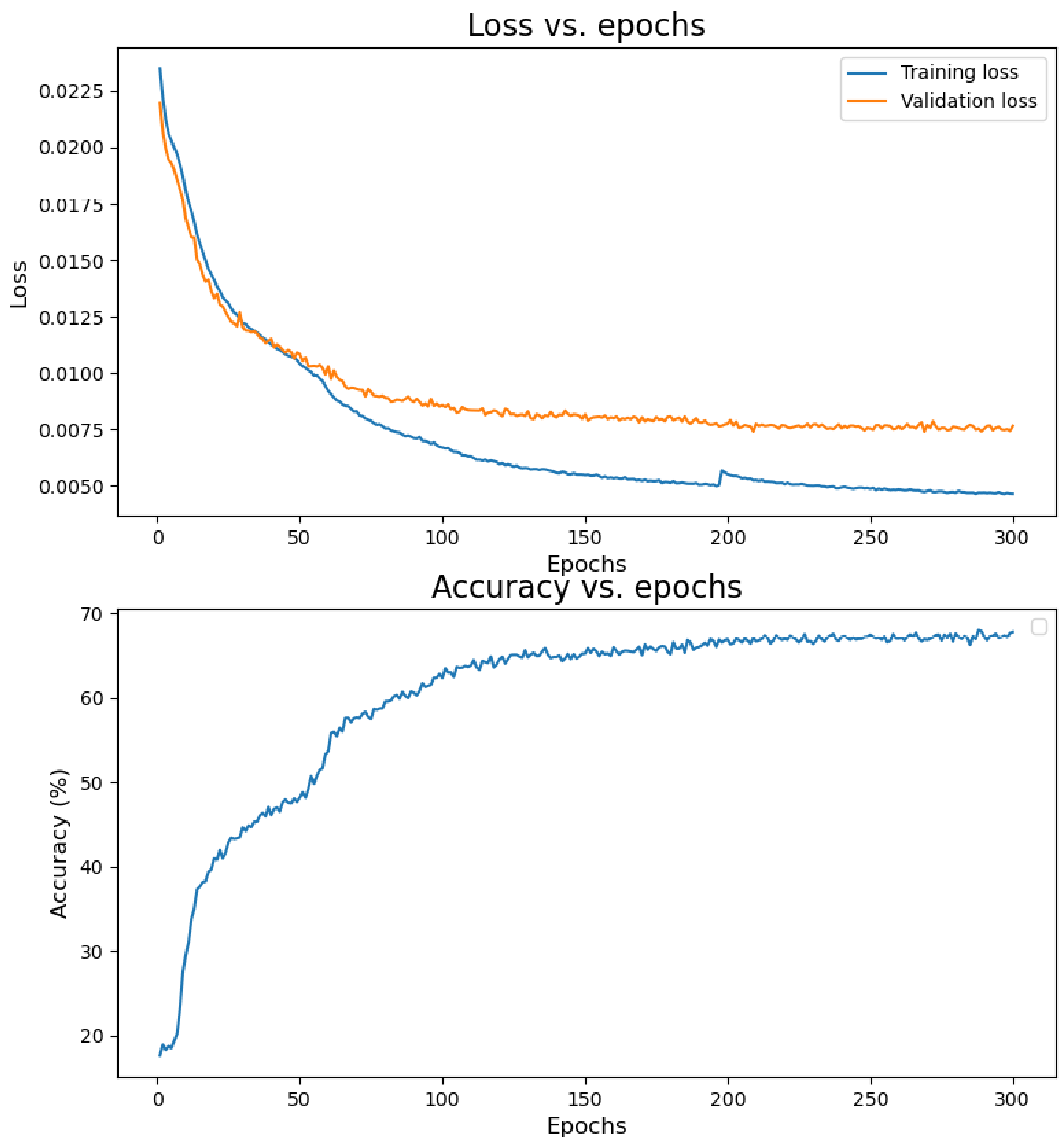


Figure 3.8: Complexity graph of the Student 3 model.

- Student model 1 reaches an accuracy of the 61,5% with a size of 2763 KB and a training time of 58 seconds per epoch.
- Student model 2 reaches an accuracy of 62%, having a size of 2897 KB and taking a training time of 44 seconds per epoch.
- Student model 3 reaches an accuracy of 66.9%, with a size of 3657 KB and a training time of 48 seconds.

In terms of size, the three models involve a huge reduction with respect to the size of the Teacher model, which is 524572 KB. The Student 1 means a reduction of the 99.47%; the Student 2, a reduction of the 99.45%; and the Student 3, a reduction of the 99.3%. As we can see, despite the Student 1 is the one that reduces the size the highest quantity, all of them are good enough. In practice, unless there is an explicit constrain in the size of the model, all the students would be adequate.

The training time can be important in some applications such as communications or distributed learning. Again, the training times are very similar, with 58, 48 and 44 seconds for the Student 1, 2, and 3, respectively. So this feature does not make a big difference.

Accuracy might be the most important attribute of a model, since it is directly related to its performance. Student models 1 and 2 have very similar accuracy, with a 61.5% and a 62%. However, the Student model 3 has almost a 5% more of accuracy than the others, achieving the 66.9%.

Therefore, taking into consideration all the results, the selected model is the number 3, since it has a significantly better accuracy than the rest, its training time is the second lowest and still, conserves a very small size with respect to the teacher.

3.2 Neural Network Pruning

Currently, neural network pruning has become one of the most popular techniques of software optimization. This popularity has lead to extensive research in the scientific community. However, despite this effort, nowadays it is still far from being fully developed since there are great constrains in its deployment.

As already seen in section 2.6, in practice, pruning scope is very limited. Unstructured methods do not have an actual influence in neural network performance due to the existing difficulties in removing the the zeroed parameters and main frameworks do not offer real alternatives for structured pruning either. In the case of PyTorch, pruning scope is also limited to turning the required parameters into zero, but not removing them form the neural network. Due to this major drawback, we needed to look for alternatives to accomplish the removal of zeroed parameters. After an deep search in the field, we found a library capable of what we were pursuing.

Simplify, introduced in section 2.6.5, is a library able to remove the zeroed parameters from a pruned neural network. In this context, we will talk of pruned neural network when its least important parameters or nodes (or filters) are turned into zero; and we will talk of a simplified neural network when the parameters that have been made zero during the pruning process are removed from the neural network.

However, this library also presents some inconvenience. Although it is supposed to differentiate which layers must be simplified and which are not (as for example, the output layer), in our experience, it does not do it. It does not accept deep neural networks whose nodes or filters have bias. We have experience several problems related to this fact that have forced us to use a modified VGG16 DNN as a teacher and a student without bias.

Initially, this work was thought around the idea of using a Densenet-121 DNN as a teacher, since it is a light and powerful neural network, but the impossibility of using it with the Simplify library made us take the decission of using the VGG16 DNN.

Next, we explain the work performed in this project related to the pruning techniques. This work is divided in three subsections which are based on the different stages of the project:

1. Firstly we perform an unstructured one-shot global pruning and explain its implications.
2. Then we perform a structured local training-pruning.
3. Finally, we perform a structured local training-pruning taking into account the results obtained from the first unstructured one-shot global pruning.

3.2.1 One-shot unstructured global pruning

Unstructured pruning is a technique based on the removal of the least important parameters of a model individually. As previously mentioned, in practice it is not possible to remove those parameters, hence, what frameworks actually do is just zeroing them.

In our case, we are performing a one-shot unstructured global pruning, which means that we are zeroing the least important parameters of a complete deep neural network at the same time, at the end of the training process. Since we cannot optimize a model doing this technique, our main goal is to identify where the least important parameters of a model are located.

It is important to mention that we have experience a major drawback in the application of this technique on the teacher model. VGG16 Teacher model has a huge size, more than 500 MB, which means a massive number of parameters. For this reason, when applying global pruning to the VGG16 Teacher model and the pruning method of the PyTorch pruning package takes places, the GPU runs out of memory, reserving more that 5.5 GB for PyTorch.

To solve this problem, we have tried to accomplish a local unstructured pruning. However, the proportion of parameters throughout the VGG16 model is not distributed homogeneously. In fact, up to a 90% of VGG16 parameters are located in its classification part, namely, the fully connected layers. Therefore, even performing the unstructured pruning locally, when the process reaches the point in which the first fully connected layer must be pruned, the GPU crashes.

Since it is not possible to accomplish the unstructured global pruning on the teacher model, we have only performed it on the student. To do so, PyTorch pruning package includes an specific method called `global_unstructured`. This method has three arguments, which are the parameters to prune, the pruning method used and the amount of pruned parameters.

Having this information, we can follow the next procedure to fulfil the one-shot unstructured global pruning:

1. Firstly, we load the trained model.
2. We set the percentage of the parameters of the model that must be pruned.
3. Then, we stablish the type of modules that must be pruned. In PyTorch, these modules usually are `nn.Conv2d` or `nn.Linear`, which make reference to the convolutional and dense layers.
4. Having the modules, we identify the parameters that must be pruned.
5. We set the pruning method and perform the pruning process using the function `global_unstructured`.
6. We use the `remove` method on the pruned modules.
7. Finally, we accomplish an inference of the pruned model to test its performance.

In our case, firstly we load the state dictionary of the already trained Student model 3. Our objective is to prune it for different percentage of pruned parameters, so we set those percentages to run a loop. Afterwards, we identify all the parameters of all the layers as parameters to prune and set the `L1Unstructured` method, which compares the values of those parameters using the norm 1. Finally, we run the `global_unstructured` method to perform the pruning process.

In Figures 3.9, 3.10, 3.11, 3.12 and 3.13, it is shown the sparsity (percentage of pruned parameters) of the layers of the Student model 3 for a total amount of pruned parameters of the 20%, 30%, 40%, 50% and 60%, respectively.

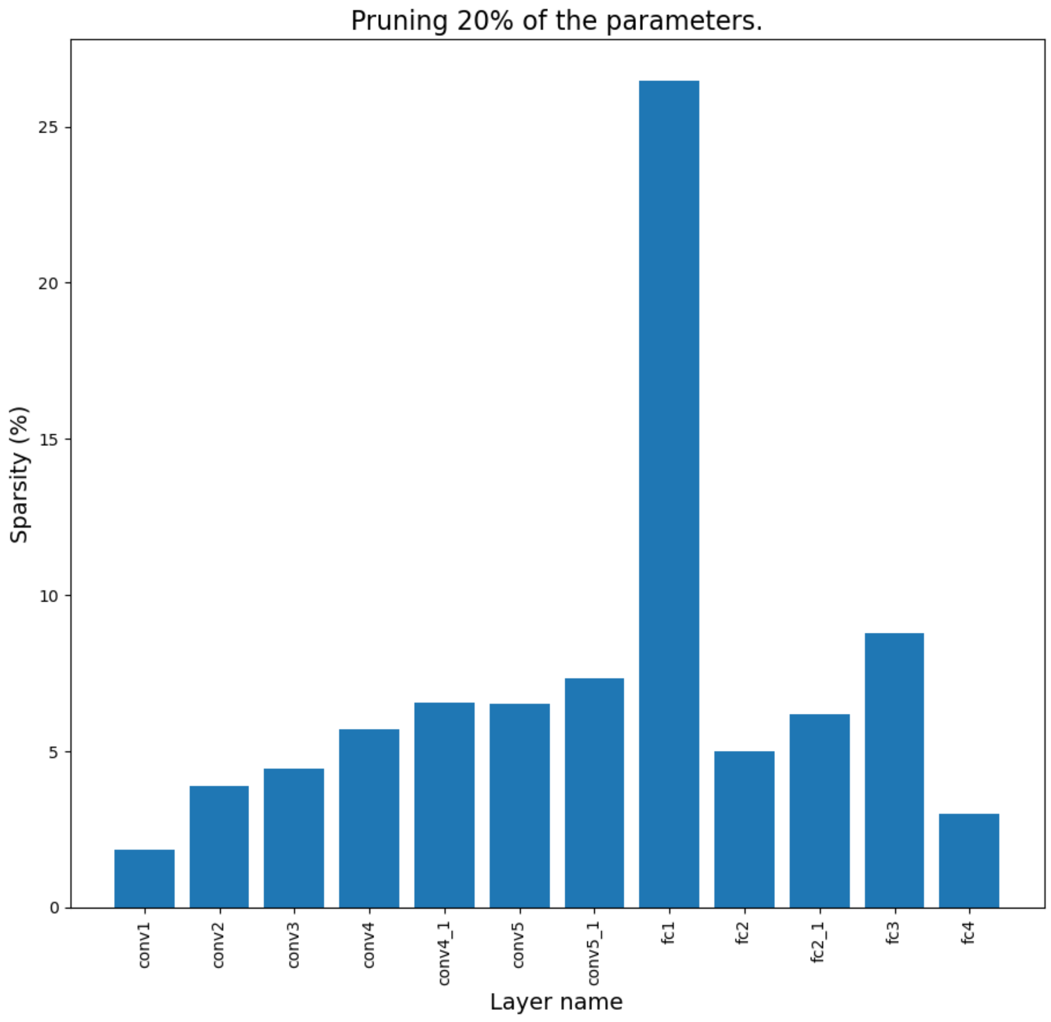


Figure 3.9: Sparsity of each layer of the Student model after applying a one-shot unstructured global pruning to the 20% of parameters.

In the graphic represented in Figure 3.9 we can observe that the first classification layer, whose name is *fc1*, has a much higher sparsity than the rest of the of the layers.

It is important to mention that convolutional neural networks which have a feature-extracting part, formed by convolutional layers, and a classification part, formed by fully connected layers, the first fully connected layer of the classification part tends to have a great percentage of the parameters of the whole model. In the case of our Student model, this layer is the *fc1* and has the 67.07% of the parameters of the entire model.

This explains why the sparsity of the rest of the layers is so low.

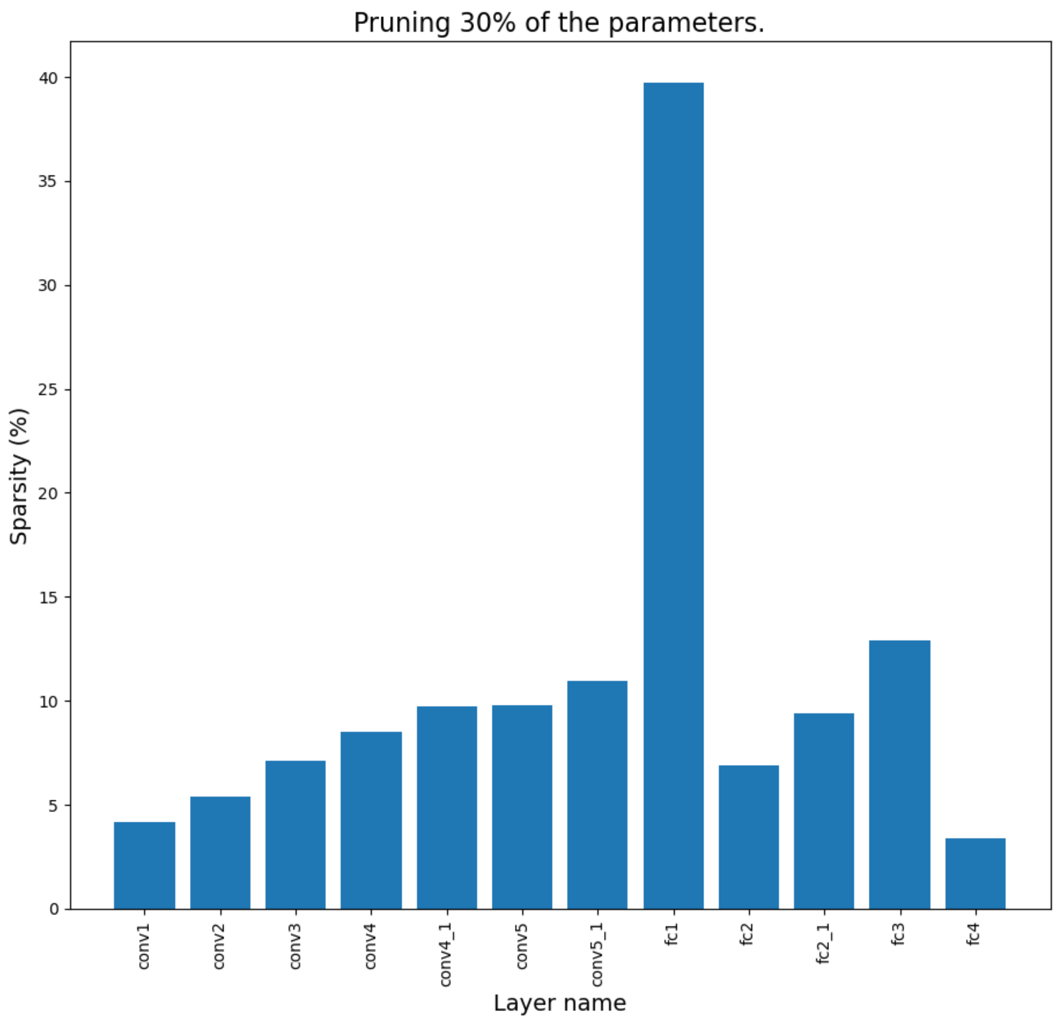


Figure 3.10: Sparsity of each layer of the Student model after applying a one-shot unstructured global pruning to the 30% of parameters.

Analysing the Figure 3.10 we can confirm the tendency of the layer *fc1*. In the feature-extracting part, we can also observe that in the first layers the sparsity is lower than in the last ones. This can be explained according to the sort of features that each layer extract: while the first layers manage more general features which are usually important, the lest layers focus on specific features that can be irrelevant.

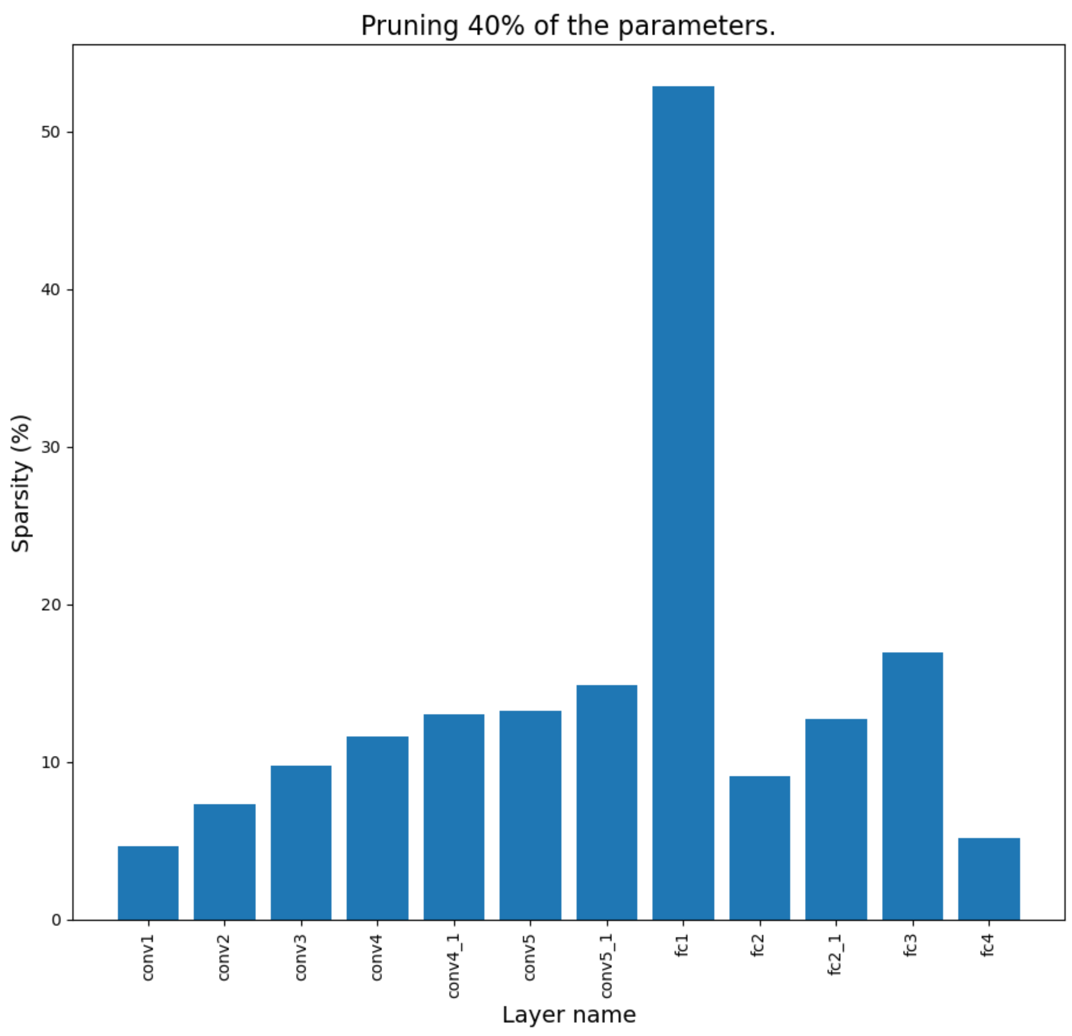


Figure 3.11: Sparsity of each layer of the Student model after applying a one-shot unstructured global pruning to the 40% of parameters.

In the Figure 3.11 we can observe that the tendency of pruned parameters in the different layers is confirmed. However, we can also observe that *fc2* and *fc4* have a lower sparsity than the rest of the fully connected layers. The main reason for this behaviour is that *fc2* receives a huge amount of information from the previous layer (*fc1*) and therefore, its classification must be important. On the other hand, *fc4* is the output layer and thus, its parameters take an important part in the performance of the model.

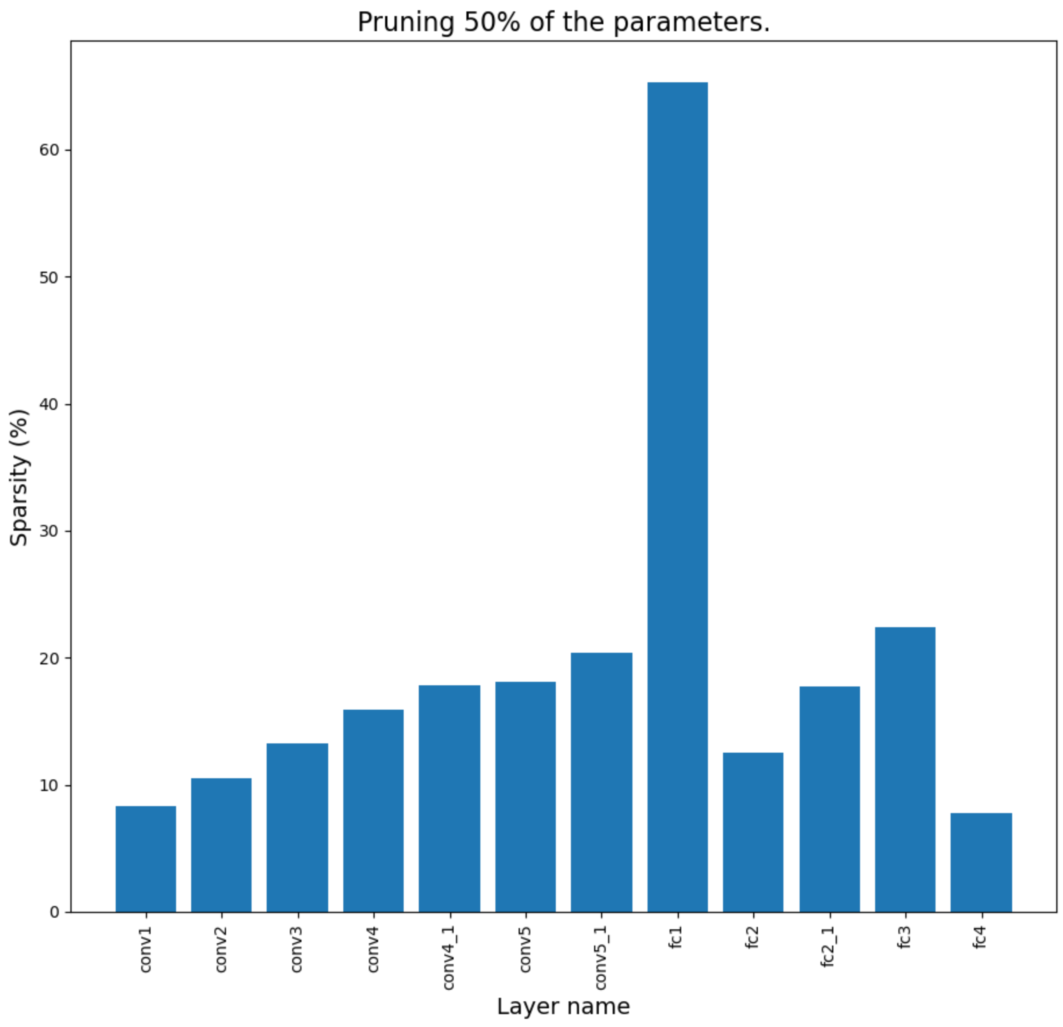


Figure 3.12: Sparsity of each layer of the Student model after applying a one-shot unstructured global pruning to the 50% of parameters.

In Figures 3.12 and 3.13 it can be observed that the trends commented before continue. Particularly, in Figure 3.13, in spite of these trends, the sparsity of all the layers increases significantly as a consequence of the higher percentage of global parameters.

Once we have seen the impact that pruning has in the different layers of the Student model, we must analyse its impact on the final accuracy of the model. To do this, we have compared the amount of pruned parameters of the whole model with its accuracy.

The result is the graphic shown in the Figure 3.14. Analysing this plot, it can be seen that, theoretically, for a wide range of pruned parameters the accuracy of the model does not decrease.

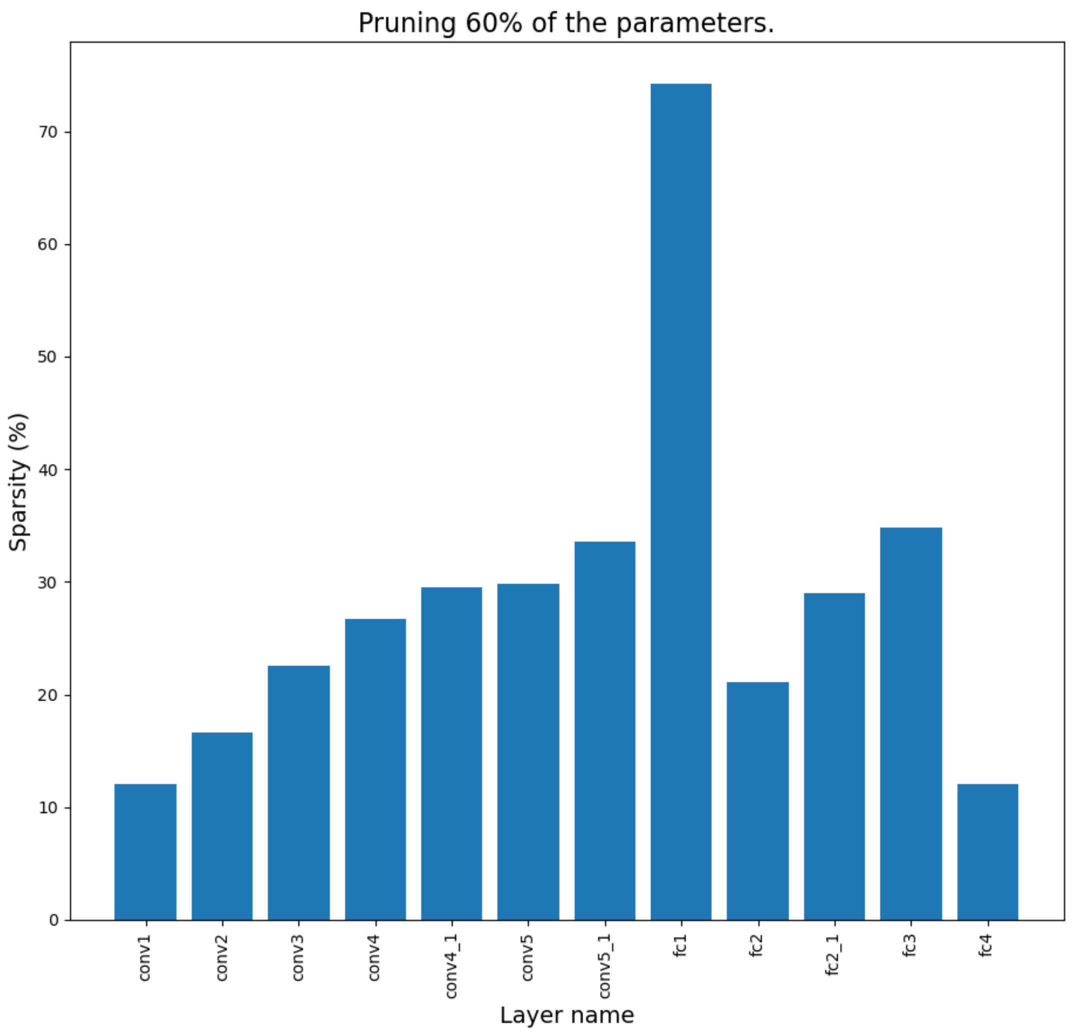


Figure 3.13: Sparsity of each layer of the Student model after applying a one-shot unstructured global pruning to the 60% of parameters.

It is important to remember that global unstructured pruning affects directly the least important parameters of the model and therefore, it is very powerful. As we can see in Figure 3.14, using this method we can prune the 60% of the parameters without any loss in accuracy. This result is astonishing, since it reveals that the pruned model can perform exactly the same as the original model, but having the 40% of its parameters. However, as we have seen in previous sections, unstructured pruning is theoretical and can not be put into practice. This means that, even thought a 60% of the parameters of the model can be pruned maintaining its accuracy, there will not be an actual improvement in the structure of the model, this is, its size will remain the same (3657 KB) and its training time as well.

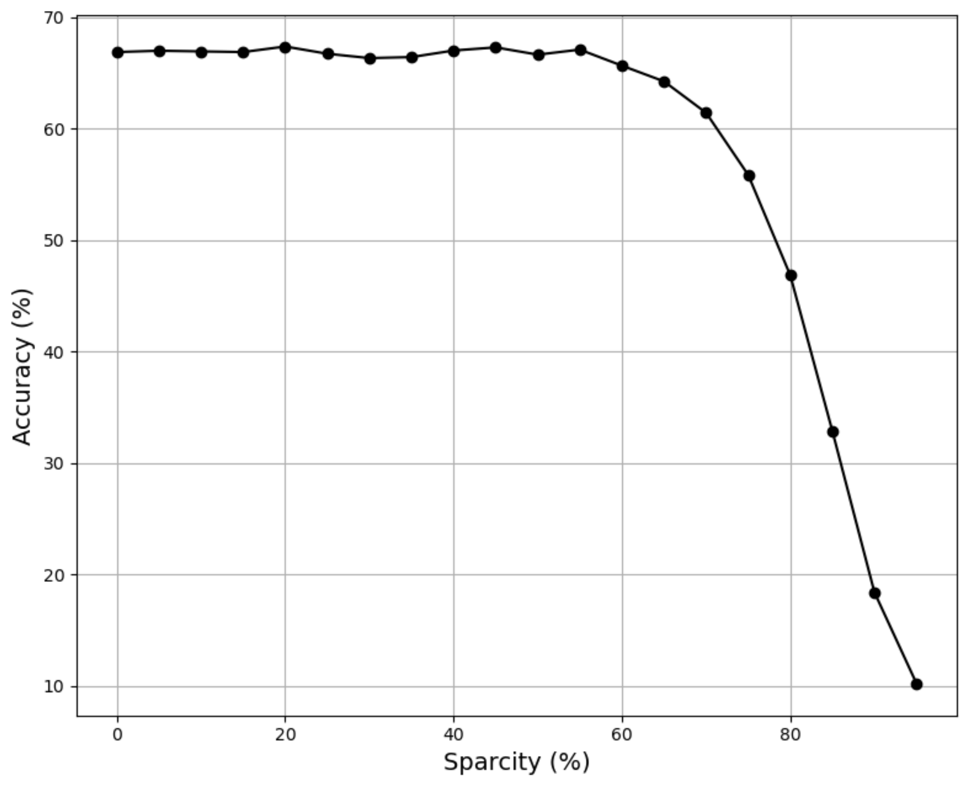


Figure 3.14: Representation of the accuracy achieved by the Student model when it is pruned for different percentages of parameters.

3.2.2 Structured local training pruning

Once the unstructured global pruning is done, the next step in the proposed experiments is to perform a structured local pruning. As said in section 2.6, using this method, nodes and filters will be pruned instead of parameters, so if we want to accomplish a pruning of a certain percentage of parameters, we need to relate the percentage of pruned nodes with the percentage of pruned parameters. This relation has also been introduced in section 2.6 and links the percentage of pruned input and output channels with the percentage of pruned parameters.

Unlike the one-shot pruning, with training pruning we must prune the model at the same time as it is trained. This implies that we must retrain the teacher and the student models in order to obtain reduced models with adjusted parameters.

For each epoch, the training pruning process follows the next steps:

1. Firstly, the model is trained as it is done normally.
2. Then, we perform the pruning structured local training of the model. Unlike with unstructured global pruning, in this case we need to directly identify and prune the desired layers. To do it, we run a loop throughout the whole model checking if each module must be pruned. If so, the structured local pruning is performed using the method `ln_structured` from the PyTorch pruning package. This method receives four inputs as arguments, which are the module that contains the tensor to prune, the name of the parameter on which pruning will act (it can be 'weight' or 'bias'), the percentage of nodes or filters to prune, the norm used to compare and choose the modules to prune and the dimension along the channels to prune are defined. In our case, we will only prune weights, since they represent the majority of the model and the two final arguments will always have a value of 1 and 0, respectively. Therefore, the pruning function will look like this: `prune.ln_structured(module, name="weight", amount, n=1, dim=0)`. It is important to mention that we never prune the output layer, since it creates errors when using the `simplify` method.
3. Afterwards, we simplify the model using the `simplify` method. To do so, we only need to introduce the model and a tensor of zeros as arguments.
4. Then, we need to accomplish the fine-tuning process. Since the structure of the model changes during the pruning and simplify processes, we need to re-select which parameters must be trained in the next training iteration.
5. Finally, we validate the pruned model.

As previously said, since we are now pruning nodes and filters, it is possible to use the Simplify library. Doing this, we will no longer do a theoretical pruning, but will start performing a real compression of the models.

Another aspect to keep in mind is that, since the training pruning procedure removes a little amount of nodes or filters each epoch for several epochs, in every iteration the pruned model is smaller than in the previous one. This involves that the amount of nodes or filters that are pruned each epoch is not constant for the same percentage, as while the percentage of pruned modules is constant, the size of the model is constantly reduced. However, it is possible to compute the percentage of pruned modules in each epoch according to the final percentage of pruned modules and the number of epochs in which pruning is performed. The equation 3.1 shows this relation, where y represents the final percentage of remaining nodes or filters of a layer, x represents the constant percentage of pruned modules in each epoch and n represents the number of epochs in which the local pruning of the layer is accomplished.

$$y = (1 - x)^n \tag{3.1}$$

Therefore, knowing the final percentage of remaining modules of a layer and establishing the constant percentage of pruned modules in each epoch, it is possible to compute the needed number of epochs to fulfil this process. Taking logarithms, the equation 3.2 is achieved.

$$n = \frac{\log(1 - x)}{\log(y)} \tag{3.2}$$

In the case of structured pruning, it has been possible to perform the experiments for both the teacher and the student models, since the GPU has been able to withstand the structured local pruning process. These experiments have the objective of testing the accuracy and the computing time of both models while training.

The structured local training-pruning experiments are divided into two groups.

1. In the first group of experiments, all the layers have the same percentage of pruned modules. These experiments will be called static structured training pruning.
2. The second group of experiments is based on the results obtained in the unstructured global pruning experiment. In this experiment, the final percentage of pruned modules of each layer is selected according to the percentage of pruned parameters of each layer achieved in the unstructured pruning experiment. These experiments will be called dynamic structured training pruning.

Next, we explain the steps followed to accomplish the static and dynamic structured pruning experiments for the Teacher and the Student models.

3.2.2.1 Static structured training pruning

These experiments are characterised by the fact that all the layers of the pruned model have the same percentage of pruned modules. Below, we explain the static experiments applied on the Teacher and the Student models.

Teacher model

The aim of this training-pruning process of the Teacher model is to get a final percentage of pruned parameters of the 45%. To perform it, we firstly train the model, without pruning, for 10 epochs and afterwards, we prune a 3 percent of the nodes or filters for the following ten epochs.

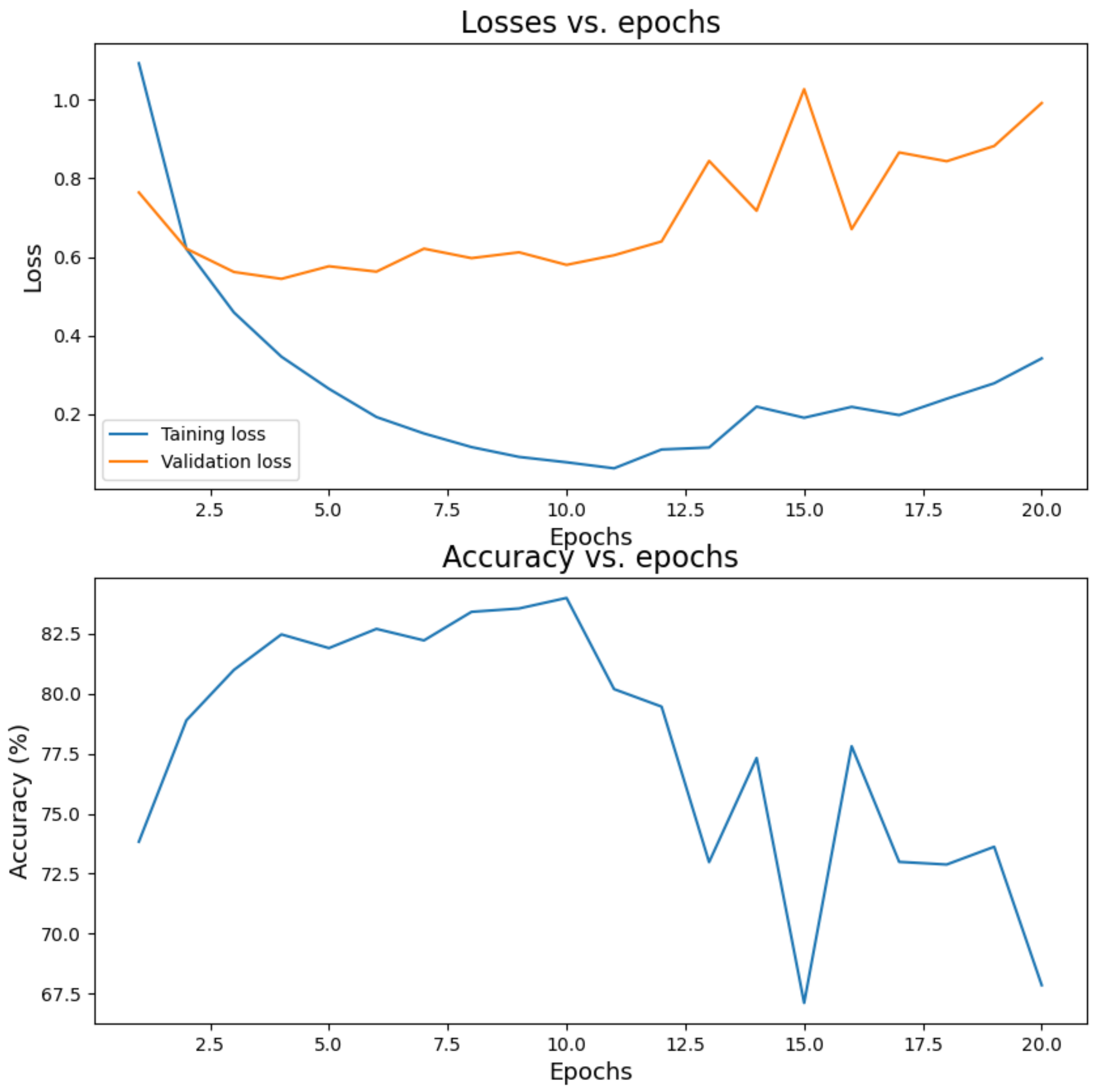


Figure 3.15: Complexity graph of the structured trained-pruned Teacher model.

Doing this and using the equation 3.1, at the end of the training process a 26.25% of the nodes or filters of each layer will be pruned, which means a reduction of the 45.6% of the parameters.Pruning a 26.25% of the total nodes or filters of each layer means that the 73.75% are still available. As explained in section 2.6, to compute the percentage of remaining parameters it is only necessary to multiply the percentage of input and output channels of each layer, which leads to the equation 3.3.

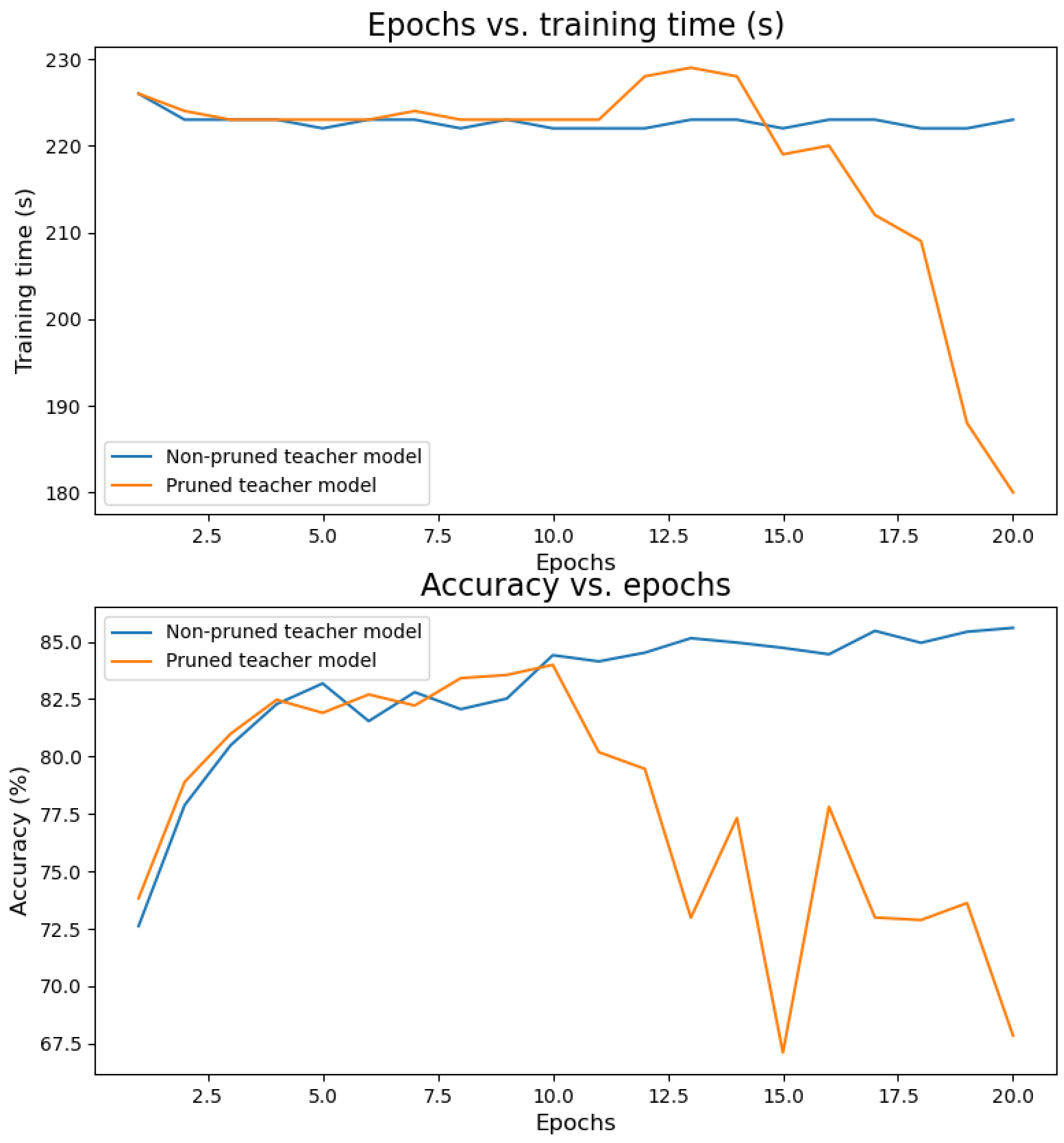


Figure 3.16: Comparison of the pruned/non-pruned Teacher models.

$$0.7375 * 0.7375 = 0.55 \tag{3.3}$$

Therefore, having a reduction of the 26.25% of the nodes will lead to a reduction of the 45% of the parameters in each layer.

However, the input and output layers are an exception to this rule, since they can not be modified and therefore, can not be pruned.

The Figure 3.15 shows the complexity graph of the structured local training-pruning Teacher model. When the model starts being pruned at epoch 10, the training and validation loss increase and the validation accuracy starts decreasing.

Moreover, if we use the early stopping technique, in this context, the pruning process is useless, since the lowest validation loss remains in the non-pruned region, this is, the ten first epochs.

However, in the Figure 3.16 it can be seen that the pruning process of the model involves a great reduction in the training time, above all in the last epochs, when the amount of pruned parameters is higher.

Therefore, depending on the application of the Teacher model there must be a balance between the accuracy that the model can achieve and the time that it needs to be trained.

Student model

The objective in the training-pruning process of the Student model is to achieve a 36% of pruned parameters at the end of the training. To do it, we need to prune a 20% of the nodes of each layer, allowing keeping a 80% of the nodes or filter. As we have previously seen, this can be explained by the equation 3.4 which states that:

$$parameters(\%) = input\_channels(\%) * output\_channels(\%) \tag{3.4}$$

And thus, it can be proved that if we keep the 80% of the nodes or filters and using the equation 3.4, the final percentage of parameters is $(0.8 * 0.8 = 0.64)$ the 64%.

To deploy this process, we prune the Student model a 2% for every epoch that is higher than 200 and lower than 280 and also is a multiple of 7. Doing this, all the modules of the model will be pruned a 2% eleven times and thus, according to the equation 3.1 we will achieve the desired pruning percentage of 20%.

It must be noticed that using this procedure, the first two convolutional layers will never be pruned, since they only have 8 and 16 filters, respectively, and thus, when a pruning of 2% is applied to them, the result is always under 0.5, which makes these layers impossible to prune.

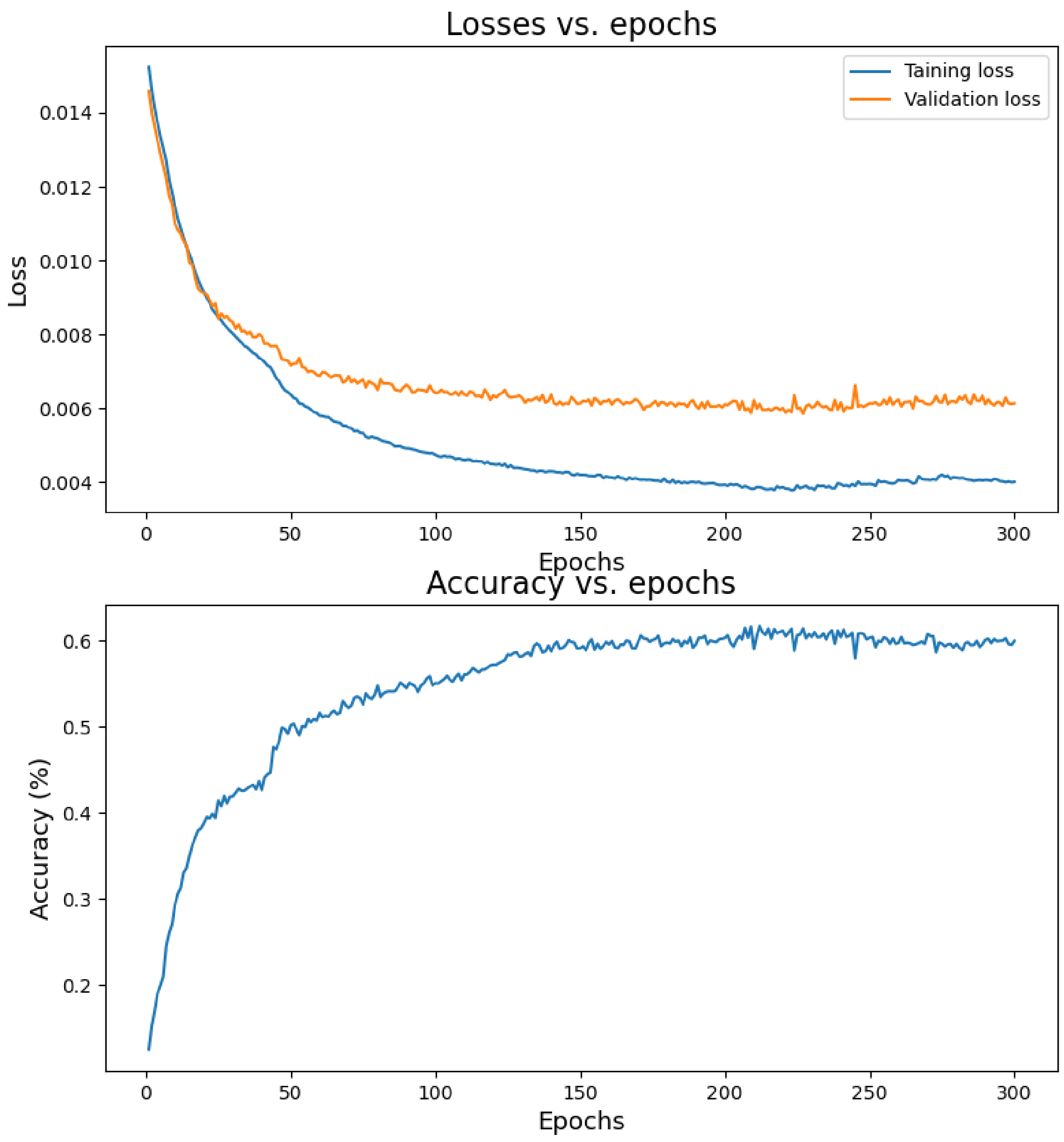


Figure 3.17: Complexity graph of the structured trained-pruned Student model.

It is necessary to be aware that for every epoch that we prune the model, we also need to simplify it, and therefore, we also need to used the `simplify` method every epoch that is higher than 200 and lower than 280 and also is a multiple of 7.

In the Figure 3.17 it is shown the complex graph of the pruned Student model.

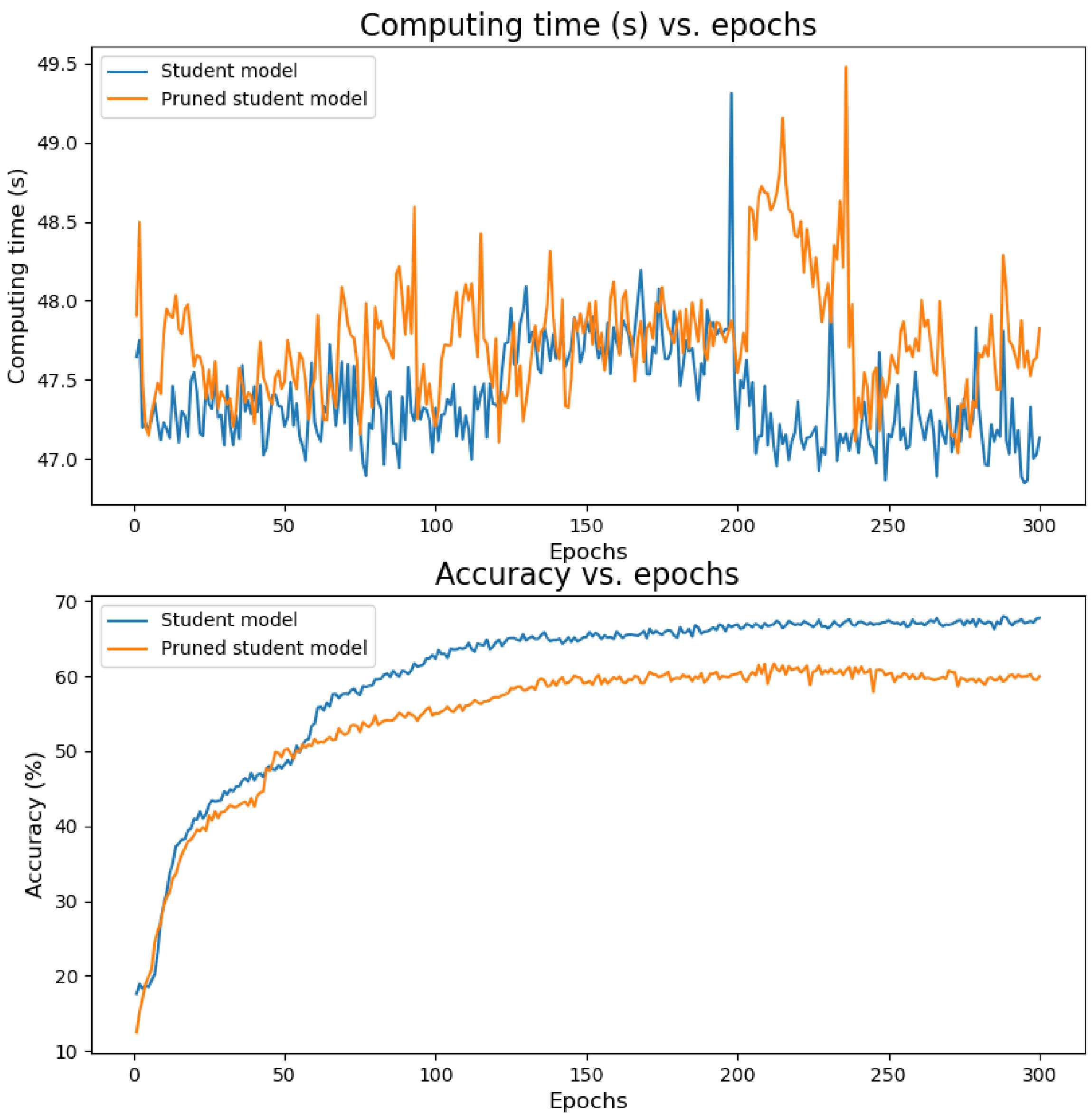


Figure 3.18: Comparison of the pruned/non-pruned Student models.

During this training pruning process we have only pruned the model for the last 100 epochs. In order to train the model it has been used a 50% of dropout rate and the early stopping technique, and therefore, the final model does not have a 36% of pruned parameters. In fact, the achieved model has a size of 3142 KB, which means a 15% of size reduction.

Comparing the complexity graph of the Figure3.17 with the one of the Figure 3.4, it is possible to observe that the pruned model slightly increases its validation losses with respect to the non-pruned model. However, with the Figure 3.18 we can also notice that the validation accuracy of the pruned model at the end of the training process is reduced approximately a 7%, which means an undesired effect of pruning. Regarding the computing time, in this case we can not conclude that pruning affects it directly.

3.2.2.2 Dynamic structured training pruning

In this subsection we are testing the performance of structured local training pruning using the data obtained from the unstructured global pruning. Namely, we will locally prune the different layers of the deep neural networks with the percentage of pruned parameters obtained in the unstructured global pruning experiment.

Since the objective is to prune a specific percentage of parameters for each layer while we prune the different nodes or filters of each layer, we need to relate the desired percentage of pruned parameters with the percentage of modules that we aim to prune. This process has been done using the equation 3.4 and taking into account that every layer is connected to the following one.

The result of these computations is the Table 3.1 for the teacher and the Table 3.2 for the student. Since it has not been able to perform the unstructured global pruning of the Teacher model due to the computing requirements of the process, we will use, as a reference, the results obtained from the unstructured global pruning of the Student model, as they share a similar structure.

Teacher model

Our objective in this experiment is to perform a training-pruning process which prunes approximately the 40% of the parameters at its end. To do it, we will customize the amount of pruned parameters of each layer by an estimation, as it is shown in the Table 3.1.

This experiment follows a training-pruning process of 20 epochs in which the model is trained without pruning for the first ten epochs, and for the second ten it is trained and pruned iteratively.

As we can see, this table is created around the idea that the first classification layer of the model is the one that has the largest number of superfluous parameters and therefore, we need to prune it for the highest percentage of parameters. However, doing this we are also forced to prune a high percentage of the parameters of its closest layers. The layers *features28* and *classifier3* are the most affected by this fact.

Table 3.1: Relations among the pruned nodes, pruned parameters and final parameters of the Teacher model.

Name	**Pruned nodes (%)**	**Final nodes(%)**	**Pruned parameters (%)**	**Final parameters (%)**
features0	4	96	4	96
features2	4	96	8	92
features5	4	96	8	92
features7	4	96	8	92
features10	4	96	8	92
features12	4	96	8	92
features14	4	96	8	92
features17	4	96	8	92
features19	4	96	8	92
features21	6	94	10	90
features24	8	92	14	86
features26	10	92	17	83
features28	26	74	35	65
classifier0	40	60	57	43
classifier3	10	90	46	54
classifier6	0	100	10	90

Since the Table 3.1 represents the final percentage of pruned nodes or filters that the model must have, it is necessary to follow an strategy to perform the training-pruning process and thus, remove the redundant nodes gradually. To do it, we use the equation 3.2, as we already know the percentage of final nodes (second column of the Table 3.1).

The pruning process starts at the epoch 11. All the layers that have a final pruning percentage of the 4% (see Table 3.1) of their filters are pruned twice, in the epochs 12 and 14, removing the 2% of the filters each time. The convolutional layers called *features21* and *features24* have a 2% of their parameters pruned along three and four epochs, respectively.

The rest of the layers are pruned every single iteration from the epoch 11 until the end of the training process. Specifically, for each epoch, the layer *features26* is pruned a 1%, the layer *features28* is pruned a 3%, the layer *classifier0* is pruned a 5% and the layer *classifier3* is pruned a 1% of its nodes. Finally, the layer *classifier6*, since it is the output layer and therefore, must be unmodified.

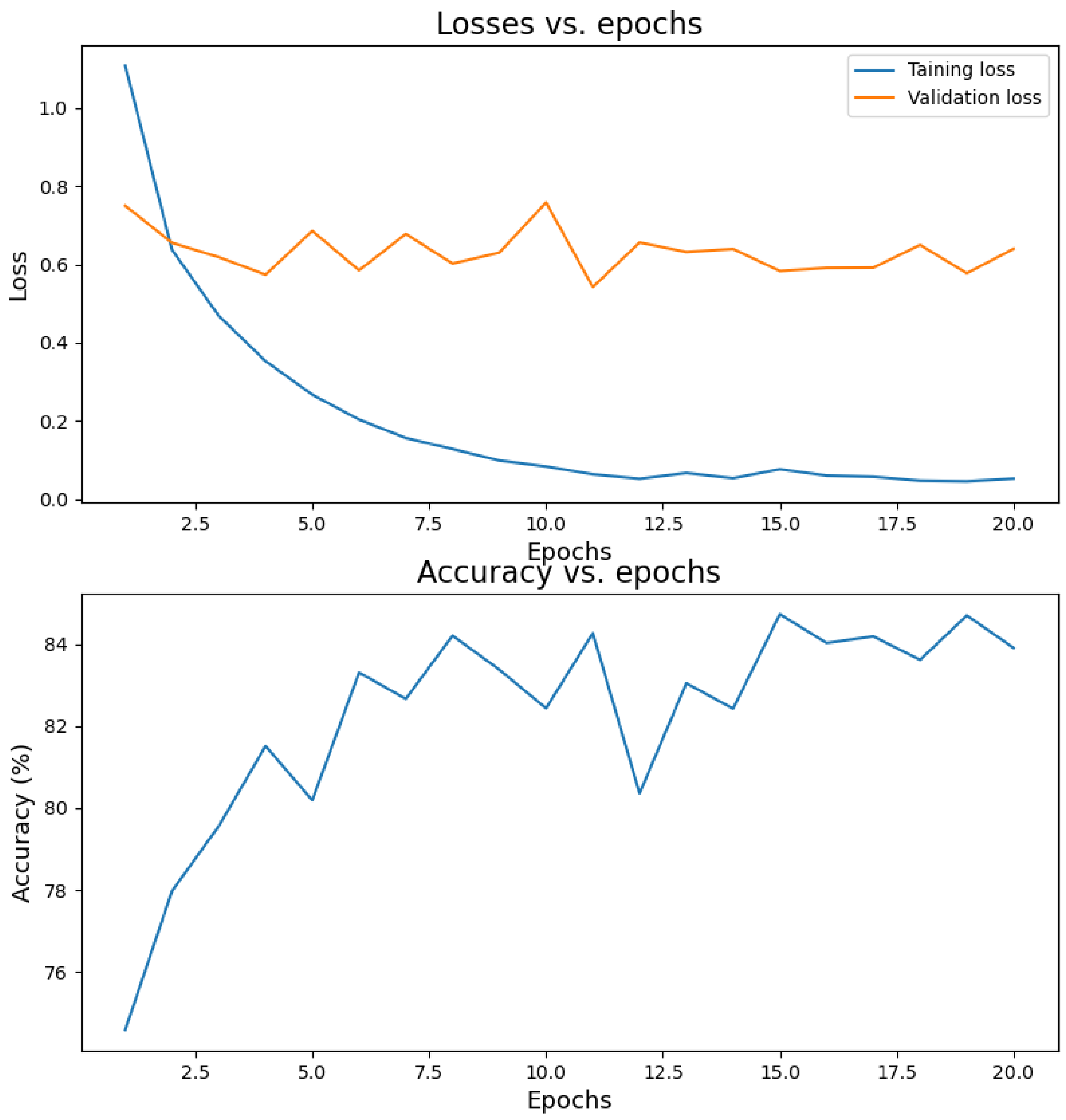


Figure 3.19: Complexity graph of the structured trained-pruned Teacher model with customized amount of pruned parameters.

With this pruning strategy, the Teacher model has been trained-pruned, obtaining the complexity graph showed in the Figure 3.19.

This complexity graph has a major difference with respect to the one seen in the section 3.2.2.

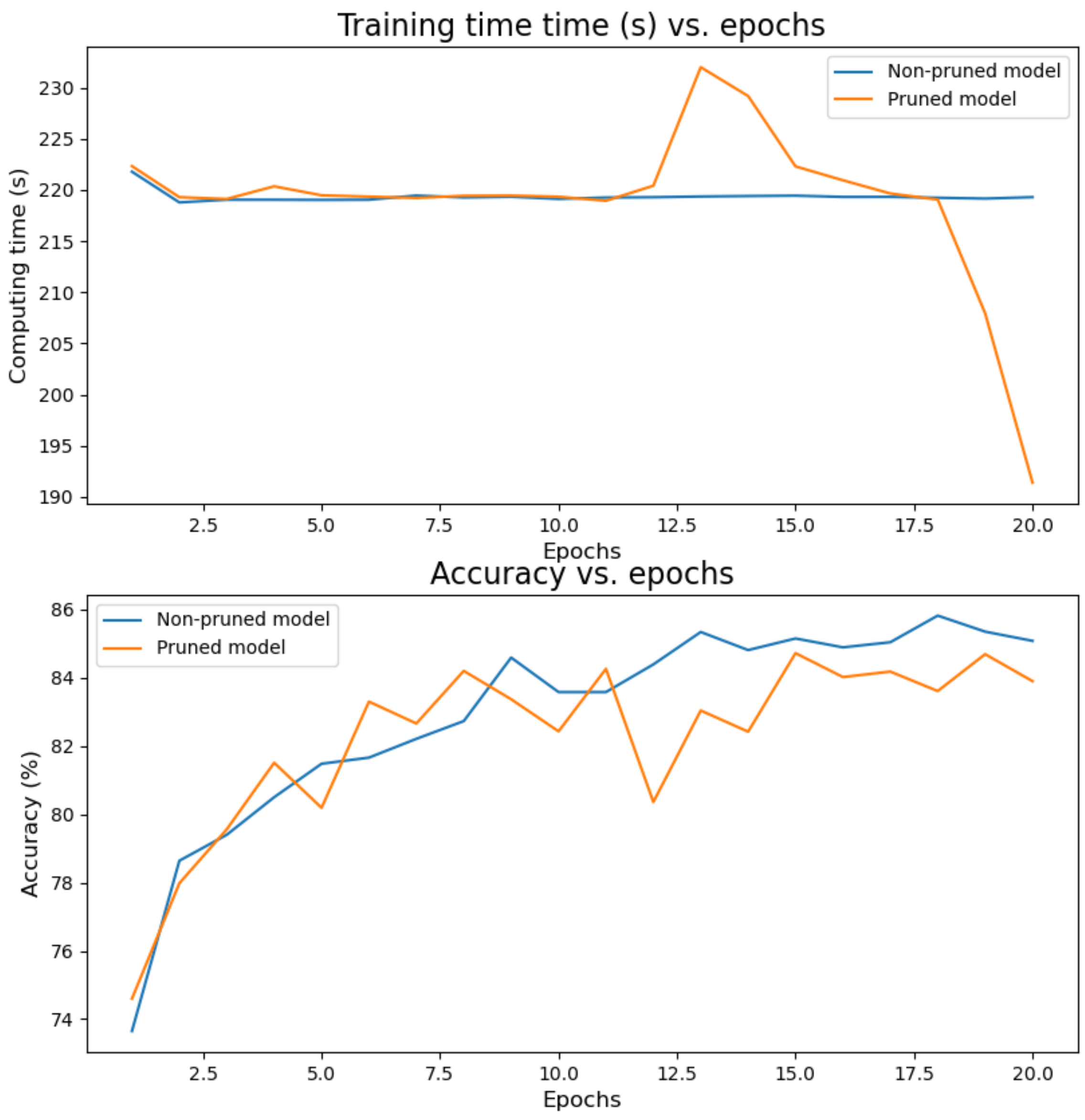


Figure 3.20: Comparison of the pruned/non-pruned Teacher models.

On the one hand, the training loss behaves in a very good way, descending almost all the time, and the validation loss remains quite invariant, achieving its minimum at the epoch

11. This means that in this case, if we use the early stopping technique, the resulting model will be pruned a 10% out of the expected pruning aim, this is, approximately a 5% of the whole model. Hence, the model passes from a size of 524572 KB to a size of 495081 KB.

On the other hand, the validation accuracy remains quite stable, not descending severely. In the Figure 3.20 it can be seen that this reduction in the accuracy is very slight for most of the epochs. Even, in epoch 11 (when the model starts to be pruned) the pruned Teacher model presents a better accuracy than the original teacher.

The training time graphic shown in the Figure 3.20 also presents an interesting behaviour. When the model is pruned for the first epochs, the training time increases. However, despite that increment of the training time, when the model is pruned for a larger amount of parameters, the training time performs as expected and decreases continuously.

Student model

Our objective with the Student model is also to prune a 40% of its parameters at the end of the training process. We will use the results obtained from its unstructured global pruning experiment, which are shown in the Figure 3.11. Using these results, we have created the Table 3.2.

Table 3.2: Relations among the pruned nodes, pruned parameters and final parameters of the Student model.

Name	Pruned nodes (%)	Final nodes (%)	Pruned parameters (%)	Final parameters (%)
conv1	6	94	6	94
conv2	2	98	8	92
conv3	8	92	10	90
conv4	4	96	12	88
conv4_1	10	90	14	86
conv5	4	96	14	86
conv5_1	32	68	35	65
fc1	32	68	54	46
fc2	4	94	35	65
fc2_1	8	92	12	88
fc3	8	92	16	84
fc4	0	100	8	92

However, although all the layers present a similar percentage of pruned parameters as in the previous unstructured global pruning experiment, there is a slight difference in the layers *conv5_1* and *and fc2*, since the constrains regarding pruning nodes instead of the pruned parameters must be taken into account.

As it can be seen in Table 3.2, each layer is supposed to have a certain percentage of pruned nodes or filters at the end of the training-pruning process. This percentage has been calculated using the results obtained from the unstructured global pruning experiment.

Besides, three experiments will be performed using the dynamic structured global pruning in order to test the effect of starting pruning in a early, middle or late stage of the training pruning process.

1. In the first strategy, called early pruning, the model is pruned along the entire training process.
2. In the second strategy, called middle pruning, the model starts being pruned at epoch 100 and lasts until the end of the training process.
3. In the third strategy, called late pruning, the model is pruned from epoch 200 until the end of the training process.

In all the strategies the same percentage of nodes and filters is pruned in order fulfil them using the same number of pruning epochs. Using the equation 3.1 it is possible to establish a relation among the expected percentage of pruned modules, the real percentage of pruned nodes and filters, the percentage of pruned modules in each iteration and the number of epoch that the layers must be pruned to achieve the objective. The Table 3.3 shoes this relation.

Table 3.3: Relations among the percentage of pruned modules in each epoch, the number of pruning epochs, and the expected and real percentage of pruned modules of each layer of the Student model.

Pruning step (%)	**Number of steps**	**Real pruned nodes (%)**	**Expected pruned nodes (%)**
2	1	98	98
2	2	96	96
2	3	94.1	94
2	4	92.2	92
2	5	90.4	90
2	20	66.7	68

Therefore, having the Tables 3.2 and 3.3, it is possible to relate each layer of the model with its expected final percentage of pruned modules and thus, the number of epochs that their nodes or filters must be pruned a 2%.

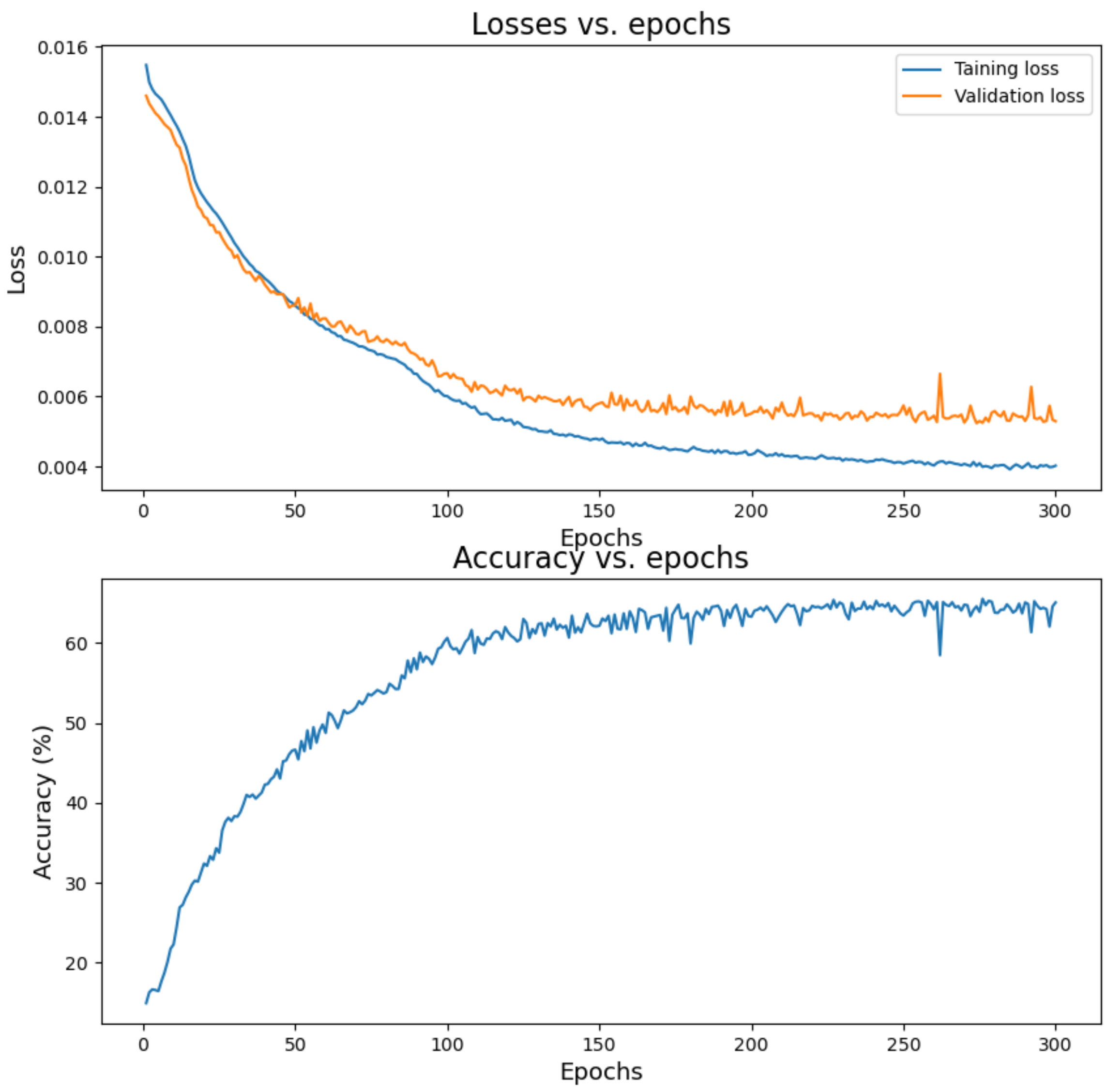


Figure 3.21: Complexity graph of the structured trained-pruned Student model with customized amount of pruned parameters following an early pruning strategy.

Since the layers *conv1* and *conv2* have few output filters (8 and 16, respectively), it is worthless to prune a 6% or 2% of their filters, as it means that we would remove 0.48 and 0.32 parameters, what is impossible. Therefore, these layers are not included in the pruning process. This fact, does no affect the quality of the pruning process, since they only represent the 1.5% of the parameters of the model.

Knowing this, the three strategies are presented.

Early pruning
In the early pruning, we follow the next strategy, which can also be seen in the Appendix E:

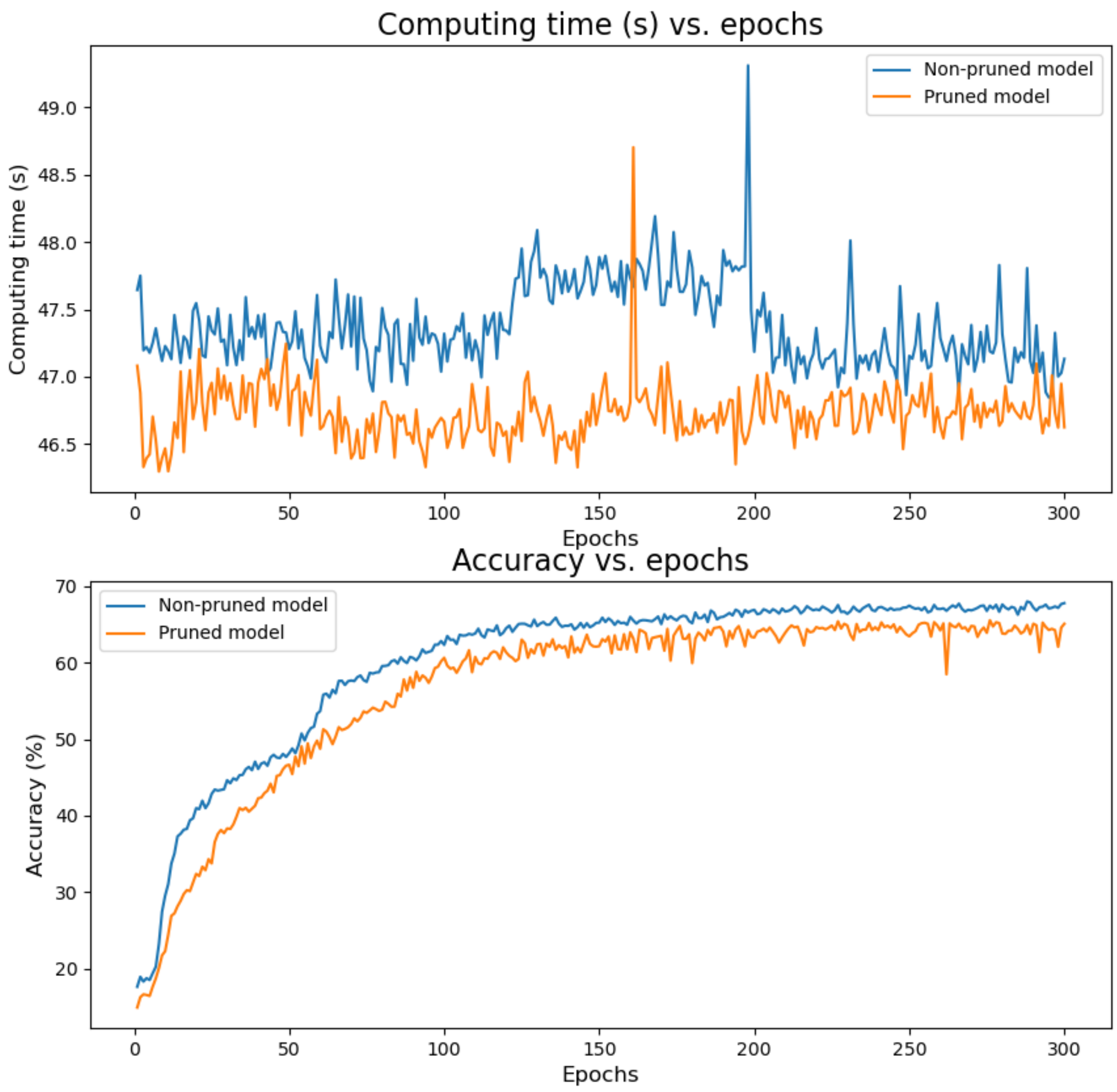


Figure 3.22: Comparison of the pruned/non-pruned Student models following an early pruning strategy.

- For layers *conv4*, *conv5* and *fc2* we prune a 2% of their modules on epochs 50 and 150.
- For layers *conv3*, *fc2_1* and *fc3* we prune a 2% of their modules on epochs 60, 120, 180 and 240.
- For the layer *conv4_1* we prune a 2% of its modules on epochs 50, 100, 150, 200 and 250.
- For layers *conv5_1* and *fc1* we prune a 2% of their modules every epoch that are multiple of 10, higher than 10 and lower than 220.

Performing the training-pruning process with the pruning strategy explained before, leads to the complexity graph presented in the Figure 3.21. As it can be seen, at the end of the training process, the training loss and the validation loss achieve values of 0.004 and 0.0053, respectively, outperforming the non-pruned model.

On the other hand, as it can be observed in Figure 3.22, the validation accuracy is slightly reduced with respect to the non-pruned model. However, this reduction means a 1% of accuracy, which is almost negligible. Using the technique of early stopping, the obtained model has a size of 2460 KB, what means that the dynamic structured training-pruning along with an early pruning strategy is capable of reducing approximately a 33% the size of the original model maintaining its full performance.

Middle pruning
In the middle pruning strategy, we follow the next scheme that can also be found in the Appendix E:

- For layers *conv4*, *conv5* and *fc2* we prune a 2% of their modules on epochs 101 and 201.
- For layers *conv3*, *fc2_1* and *fc3* we prune a 2% of their modules on epochs 120, 160, 200 and 240.
- For the layer *conv4_1* we prune a 2% of its modules on epochs 110, 140, 170, 200 and 230.
- For layers *conv5_1* and *fc1* we prune a 2% of their modules every epoch that are multiple of 10 and higher than 100.

Applying the middle pruning strategy to the Student model leads to the complexity graph shown in the Figure 3.23. In this figure it can be seen that the model experience a strange behaviour when it starts being pruned. The accuracy, which is almost stabilized before epoch 100, rises sharply afterwards. The training and validation loss also have a marked decrease around epoch 100. This could be caused because, before pruning the error function is stabilised around a local minimum and when it starts being pruned, the error function suffers an alteration, reaching a lower minimum.

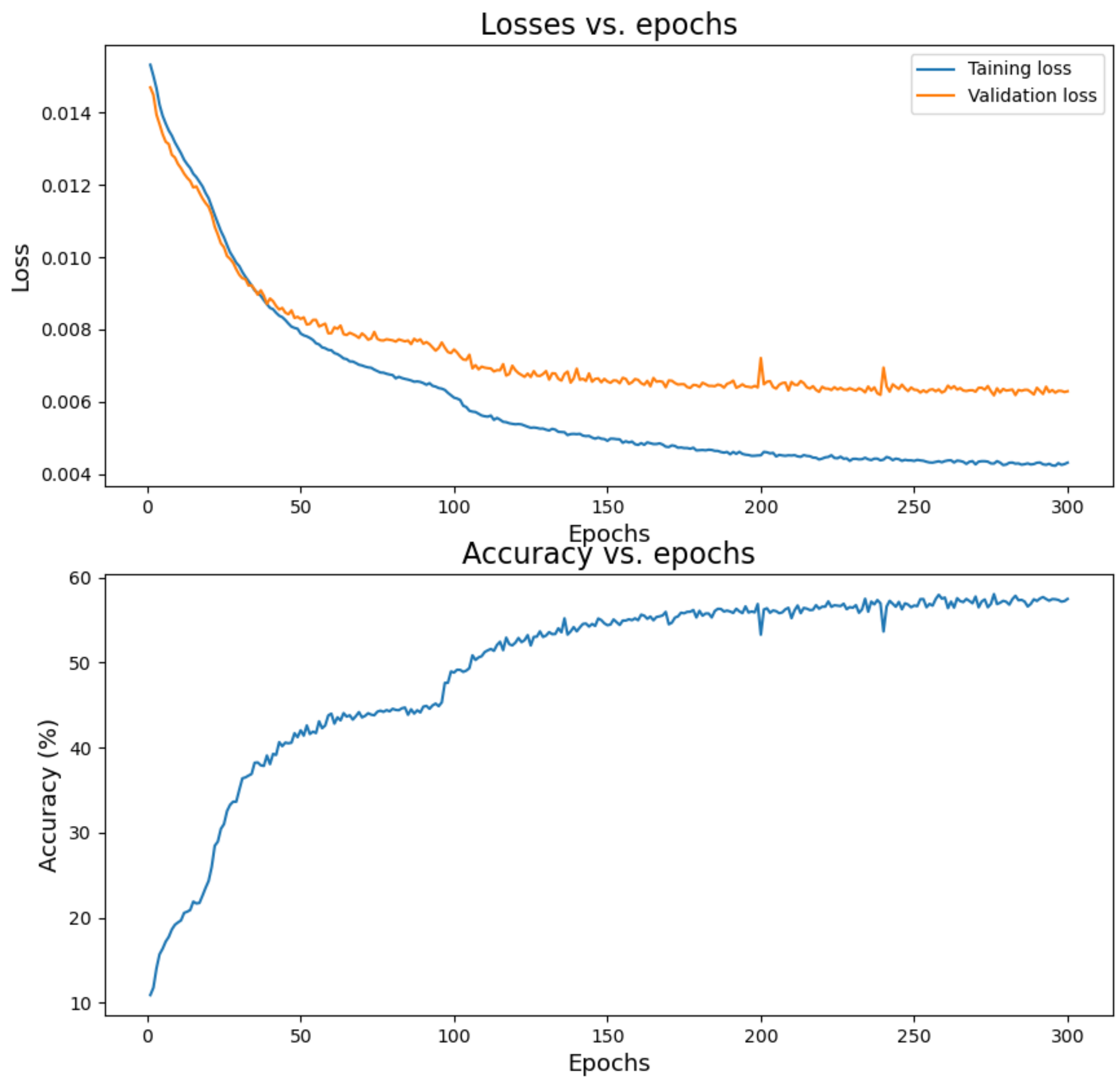


Figure 3.23: Complexity graph of the structured trained-pruned Student model with customized amount of pruned parameters following a middle pruning strategy.

However, in Figure 3.24 it is shown that the final performance of the middle strategy is not as good a the early strategy. The final validation accuracy of the middle pruning only achieves a maximum value of the 57.5%, what means a reduction of almost the 10% with respect to the non-pruned model. In Figure 3.24 it can also be seen that in this case the computing times of the pruned model is slightly lower than the times of the non-pruned model. However, this decrease is not very important, since it could simply be caused by a release of resources of the GPU.

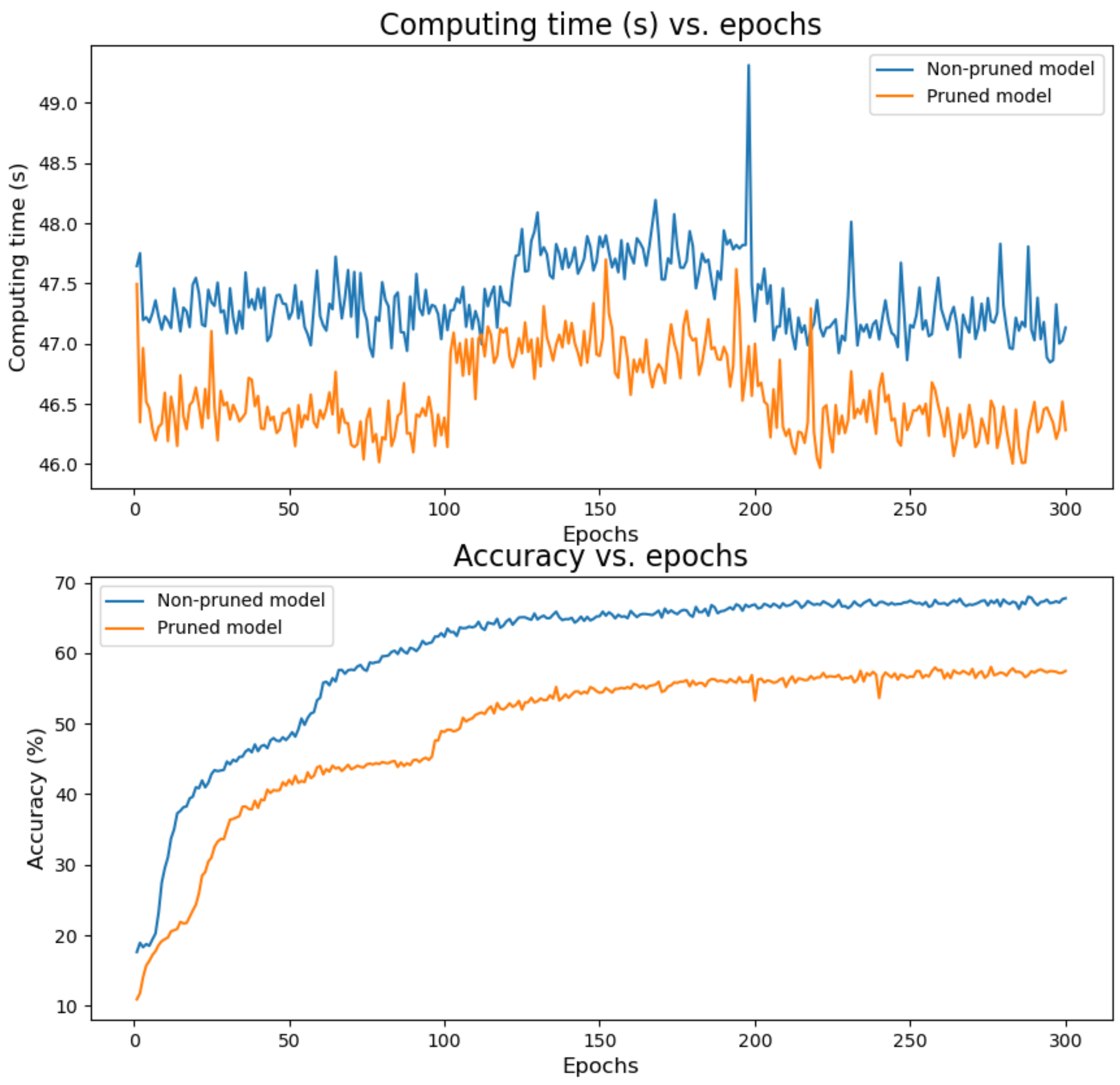


Figure 3.24: Comparison of the pruned/non-pruned Student models following a middle pruning strategy.

Late pruning

In the late pruning strategy, that can also be found in the Appendix E, we follow the next strategy:

- For layers *conv4*, *conv5* and *fc2* we prune a 2% of their modules on epochs 210 and 220.
- For layers *conv3*, *fc2_1* and *fc3* we prune a 2% of their modules on epochs 210, 230, 250 and 270.

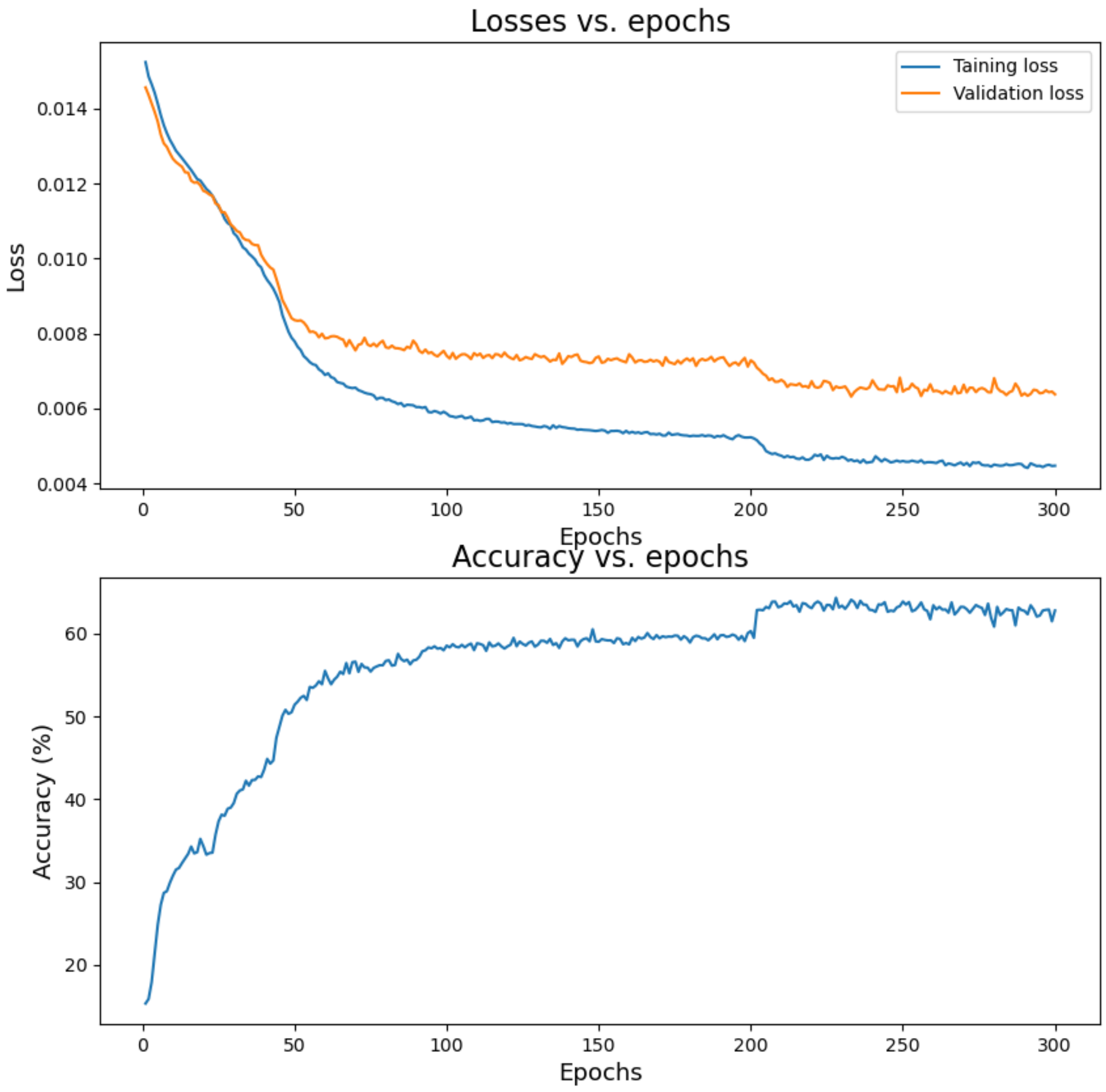


Figure 3.25: Complexity graph of the structured trained-pruned Student model with customized amount of pruned parameters following a late pruning strategy.

- For the layer *conv4_1* we prune a 2% of its modules on epochs 220, 230, 240, 250 and 260.
- For layers *conv5_1* and *fc1* we prune a 2% of their modules every epoch that are multiple of 5 and higher than 200.

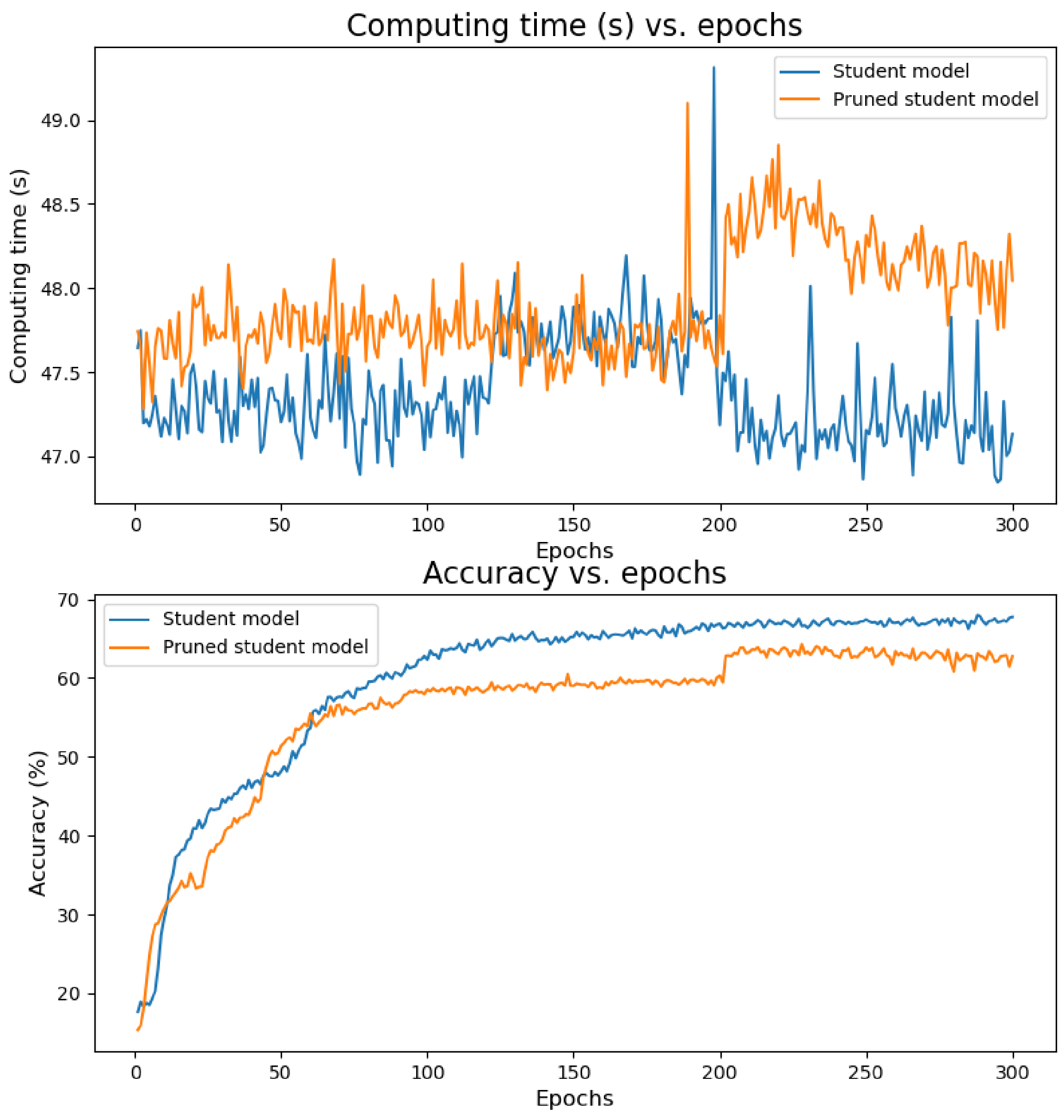


Figure 3.26: Comparison of the pruned/non-pruned Student models following a late pruning strategy.

The complexity graph of this experiment is presented in the Figure 3.25 and, as in the complexity graph of the middle pruning strategy, shows a very interesting behaviour. When the model starts getting pruned, the loss and validation losses are slightly reduced while the validation accuracy experiences a sharp increase, achieving a value of approximately the 63%.

Nevertheless, we can observe in Figure 3.26 that the computing time of the pruned model compared to the non-pruned model does not present any kind of reduction.

These results confirm the good performance of the training-pruning process when the amount of pruned nodes or filters are selected according to the pruned parameters of an unstructured global pruning experiment.

Chapter 4

Results

In this chapter the results of the different experiments performed along this project are presented and discussed. We have used and combined three techniques whose objective is the compressive optimization of a deep neural network. These techniques are the knowledge distillation, the one-shot unstructured global pruning and the structured local training-pruning. Next, the settings and results for these experiments are explained.

4.1 Knowledge distillation

In the knowledge distillation experiment we selected a pre-trained VGG16 model without biases as teacher and then, tested the performance of three different deep neural networks as student. The established settings and the obtained results of this experiment are explained in the next subsections.

4.1.1 Settings

Firstly, the pre-trained VGG16 DNN without biases was selected as the Teacher model. All the fully connected layers of this model and the last 5 convolutional layers were trained for 20 epochs, keeping the gradients of the rest layers frozen. In order to train the model, the stochastic gradient descent optimizer was chosen along with a cross entropy loss function. The inputs of the optimization function were a learning rate equal to 0.001, a momentum of 0.9 and a weight decay of $5 * 10^{-4}$.

The Student model 1 was the smallest among the three proposed student neural networks, having three convolutional layers and two dense layers, which means a size of 2763 KB. The Student model 2 had five convolutional layers and four dense layers, resulting a size of 2897 KB. Finally, the Student model 3 had seven convolutional layers and five dense layers, involving a size of 3657 KB.

All the proposed student neural networks were fully trained using an offline knowledge distillation and a response-based knowledge approaches. The loss function used to train this model was the minimum square error, which compared the softmax outputs of the teacher and the students. The optimizer used was the Adam function, with a learning rate of 0.00005. In this case, the Student model 1 was trained for 200 epochs, the Student model 2 for 100 epochs and the Student model 3 for 300 epochs. All the models were trained using a dropout rate of the 50% in their classification layers.

4.1.2 Results

The results of this experiment vary depending on the complexity of the model. The Student model 1, which is the simplest, shows an accuracy of the 61.5%, a training loss of 0.0051 and a validation loss of 0.0091; the Student model 2 shows an accuracy of the 62%, a training loss of 0.0064 and a validation loss of 0.0091; and the Student model 3, which is the largest, shows a 66,9% of validation accuracy, a training loss of 0.0047 and a validation loss of 0.0074.

Amongst the three models we have finally selected the Student model 3 in order to fulfil the following experiments. Even though this model is larger than the others, it outperforms them in terms of validation accuracy by almost a 6%. This means a better performance in the classification problem.

With the results obtained in the knowledge distillation experiment we can conclude that this technique is highly efficient in terms of compression. In our case, we have passed from a Teacher model with a size of 524572 KB to a Student model of 3657 KB, what means a size reduction of the 99,3%. However, this high decrease involves a loss in the accuracy. Specifically, we pass from a 83% of accuracy of the teacher to a 66.9% of the student, what means a reduction of a 16.1% of the accuracy.

Depending on the needed application, it is necessary to balance between the size and the accuracy of the Student model. Results show that the larger the model is, the better accuracy it will have. However, it is also important to know how to enlarge the Student model. We must be aware that, usually, the first classification layer of the model, which receives the data from the last convolutional layer, means more than the 50% of the size of the whole model. Hence, in order to increase the accuracy of a Student model the key is to add new convolutional and dense layers that do not involve an increase in the number of parameters of the first classification layer.

4.2 One-shot unstructured global pruning experiment

In the one-shot unstructured global pruning experiment this technique was applied to the VGG16 Teacher model and the Student model selected in the Knowledge distillation experiment. Next, we explain the settings of this experiment and its results.

4.2.1 Settings

To perform this experiment the same steps were followed for both the Teacher and the Student models. Firstly, the state dictionary saved from the training process of the Knowledge distillation procedure was loaded into a variable. Then, all the parameters of the models were identified and assigned to a variable to prune them. Afterwards, the model was globally pruned and the inference process was performed in order to calculate the accuracy of the pruned model. This procedure was repeated 20 times, beginning with a 0% of pruned parameters and rising this percentage a 5% until achieving a 95%.

4.2.2 Results

On the one hand, accomplishing the experiment on the Teacher model was impossible since its huge size made the GPU run out of memory. Since the global pruning method from the PyTorch pruning package (`global_unstructured(parameters_to_prune, pruning_method, amount)`) takes as first argument the total number of parameters to prune in the model, it is understandable that the GPU crashed, since it performed operations on more than 500 MB of variables at the same time. Knowing that the first classification layer of the Teacher model represents the 69% of its parameters, we also tried to apply the global pruning only to this layer in order to see if it was possible to perform the global pruning to the model partially. Trying this, the same problem occurred and the GPU still run out of memory.

On the other hand, it was possible to perform this experiment on the Student model, since, as previously commented, it is much smaller than the teacher. Although global pruning can be applied to the Student model, it is not possible to use it to compress and optimize the model, since the methods of the PyTorch pruning packages do not remove the pruned parameters from the model. Despite this fact, the results obtained shows that the majority of the most superfluous parameters of the model are located in its first classification layer. We can also conclude that the most important parameters are situated in the first convolutional layer, which is the input layer of the model, and the last fully connected layer, which is the output layer. This makes sense, as the first convolutional layer is used to extract the most general features from the data and the last fully connected layer is the main classifier. The results of this experiment also show the power of the unstructured global pruning, since, as it can be seen in the Figure 3.14, this type of pruning can support up to a 60% of pruned parameters without a significant loss of accuracy, what (if it were possible to put this technique into practice) would mean a reduction of more than half of the model while maintaining its full performance.

4.3 Structured local training-pruning

Finally, some experiments are fulfilled using a structured local training-pruning procedure.

1. The first experiments, called static pruning, were accomplished by pruning the same percentage of nodes and filters of each layer of the model locally.
2. The second experiments, called dynamic pruning, were accomplished by choosing the corresponding percentage of pruned nodes or filters of each layer according to the percentages of pruned parameters of each layer obtained from the one-shot unstructured global pruning experiment.

These experiments were fulfilled in both the Teacher and the Student models being aware that, unlike with the unstructured global pruning, now we are not pruning parameters but nodes and filters, and therefore if we aimed to perform the pruning procedure according to the percentage of pruned parameters, we needed to convert the percentage of pruned parameters of each layer into the percentage of pruned nodes or filters.

On the other hand, aiming to prune a final percentage of modules in different steps involves the use of the equation 3.1, in which y is the final percentage of pruned modules, x is the percentage of pruned modules in each epoch and n is the number of epochs in which pruning is accomplished.

Unlike the previous subsections, in the followings, the performance of both the Teacher and the Student models were tested. Therefore, the following subsections will be divided into the experiments accomplished on the teacher and the experiments of the student.

4.3.1 Static pruning

As previously mentioned, the static pruning experiments follow an approach in which the same percentage of modules of all the layers of a model are pruned gradually along the epochs. Next, we explain the settings and results obtained from the static pruning experiments performed on both the Teacher and Student models.

4.3.1.1 Teacher settings

In the static experiment of the Teacher model, our final aim was to prune a 45% of the parameters at the end of the training-pruning process. To do so, we trained the model for 10 epochs without pruning and then, we trained it while pruning a 3% of the modules of each layer every epoch for another 10 epochs. Thank to the equation 3.1, it can be checked that the final percentage of pruned modules of each layer is the 26.25%, and thus, the final percentage of pruned parameters is the 45.6%.

4.3.1.2 Teacher results

In the complexity graph of the Figure 3.15 it can be observed that, when the pruning is accomplished, the model starts to behave badly, increasing the training and validation losses and decreasing the validation accuracy. On the other hand, in the Figure 3.16 is shown that the training time decreases when the model is pruned, as it is expected. However, an anomaly occurs at epochs 11, 12, 13 and 14, when the training time increases despite the fact that the model is getting pruned. This can happen because the VGG16 structure is originally optimized to be processed in the most effective way. The number of nodes and filters of all the VGG16 layers is always a power of two, what optimizes the binary computations in hardware. If we remove a certain number of modules from these layers, the computations performed tend to be more complex and therefore, take a longer time. However, if we remove a sufficient number of modules from the layers, the computation time in the end is reduced. This explains the graphic shown in the Figure 3.16, in which the training time increases at the first pruning epochs and then, decreases.

4.3.1.3 Student settings

In the static experiment of the Student model, firstly, we trained the model for 200 epochs without pruning and afterwards, we pruned a 20% of the modules of each layer along the 100 last epochs, what means a final pruning of the 36% of the parameters. To do this, we pruned a 2% of the modules of each layer every epoch that is higher than 200, multiple of 7 and lower than 280, what involves that we pruned a 2% of the modules of each layer in 11 different epochs. Using the equation 3.1 and having the n (number of pruning epochs) and x (percentage of pruned modules in each epoch per layer) it is proved that we achieve a final pruning percentage of 20%.

4.3.1.4 Student results

The complexity graph of this experiment, which is shown in the Figure 3.17, exposes a good response to the training-pruning process. The training and validation losses have a tendency to descend and the validation accuracy grows until converge on a maximum, which is approximately a 60%. However, compared to the non-pruned student model, the validation accuracy is reduced by a 7%, which involves an important decrease. Regarding the computing time along the training-pruning process, there is no evidence that the pruned Student model is trained faster when its size is reduced. This result can be explained by the fact that the Teacher model needs to be loaded into the GPU memory in order to train the Student model. Since the Teacher model has a huge size compared to the student, this can slow down the whole training process.

4.3.2 Dynamic pruning

In this section it is reviewed the dynamic pruning experiment of the Teacher model and the three experiments fulfilled for the Student model, which test the performance of dynamic pruning, starting in an early, middle or late stage of the training-pruning process.

4.3.2.1 Teacher settings

In the dynamic experiment of the Teacher model, we estimated the final percentage of pruned nodes or filters in each layer using as a reference the results obtained by the one-shot unstructured global pruning experiment accomplished on the Student model. We did this because, in our case, the Teacher and Student models share a similar structure and therefore, it is acceptable to think that they also share a similar distribution of redundant parameters. The estimated final percentages of pruned nodes and filters of each layer are shown in the Table 3.1. To apply this pruning percentages we followed a strategy in which firstly, we trained the model for 10 epochs without pruning it and for the last 10 epochs we performed a training-pruning process.

Throughout this training-pruning process, all the layers that had a final pruning percentage of the 4% of their filters were pruned twice on epochs 12 and 14 a 2% of their modules. The layer *feature21*, which had a final pruning aim of the 6% of its filters was pruned a 2% on epochs 12, 14 and 16. The layer *feature24* aimed a 8% of pruned nodes at the end of the process and thus, a 2% of its filters were pruned on epochs 12, 14, 16 and 18. Layers *features26* and *classifier3* aimed a 10% of pruned nodes at the end of the process, so a 1% of its modules were pruned along the ten last epochs of the training process. The layer *feature28*, which aimed a 26% of pruned nodes at the end of the training process, was pruned a 3% of its filters for the last 10 epochs. Finally, the layer *classifier0*, which aimed a 40% of pruned nodes at the end of the training process, was pruned a 5% every epoch for the last 10 epochs.

4.3.2.2 Teacher results

The complexity graph of this experiment, which is presented in the Figure 3.19, shows that the training and validation losses do not increase with the training-pruning process, and the validation accuracy is maintained approximately around the 84%. Therefore, we can conclude that the performance of this experiment fairly improves the results of the model with respect to the results obtained with the static pruning.

4.3.2.3 Student settings

As explained previously, the dynamic pruning of the Student model was tested in three different experiments, which correspond to an early pruning experiment, in which the model is pruned from the beginning of the training process, a middle pruning experiment, in which pruning is performed from epoch 100 and a late pruning experiment, in which the model is pruned from epoch 200.

All these three strategies share the same percentage of pruning modules in each epoch, but it is done in different epochs.

Layers *conv4*, *conv5* and *fc2* were pruned a 2% of their modules on two epochs. In the early strategy these epochs were the 50 and 150, in the middle strategy these epochs were the 101 and the 201, and in the late strategy the epochs were the 210 and 220. Layers *conv3*, *fc2_1* and *fc3* were pruned a 2% on four different epochs. In the early strategy these epochs were the 60, 120, 180 and 240, in the middle strategy these epochs were the 120, 160, 200 and 240, and in the late strategy the epochs were the 210, 230, 250 and 270. The layer *conv4_1* was pruned a 2% on five different epochs; in the case of the early strategy, these epochs were the 50, 100, 150, 200 and 250; in the case of the middle pruning strategy the epochs were the 110, 140, 170, 200 and 230; and in the case of the late strategy, the epochs were the 220, 230, 240, 250 and 260. For layers *conv5_1* and *fc1* the objective was to prune a 2% of the modules 20 times. To do so, in the early strategy they were pruned every epoch that is multiple of 10, higher than 10 and lower than 220; in the middle strategy they were pruned every epoch that is multiple of 10 and higher than 100; and in the late strategy they were pruned every epoch that is multiple of 5 and higher than 200.

4.3.2.4 Student results

In this case, the results of the three experiments will be divided into three different paragraphs.

Early pruning strategy
The dynamic structured training-pruning along with an early pruning strategy shows a very effective performance. Since the model gets pruned from early epochs, using the early stopping technique it is possible to achieve a significant reduction in the size of the model. Besides, this experiment shows that the accuracy of the model is very slightly affected, keeping a percentage of the 65%. The training and validation losses, on the other hand, are not affected by the pruning, keeping similar values as the non-pruned training.

Middle pruning strategy
Compared to the early strategy, the middle pruning strategy performs worse. At the end of the training process, it only achieves a maximum of the 57.5% of validation accuracy, what means a great reduction in accuracy with respect to the non-pruned model.

Furthermore, the complexity graph of this experiment presents a very sharp increase in the validation accuracy when the model starts getting pruned. This can be caused because the error function reaches a local minimum before the beginning of the pruning process, and when it starts, it creates and alteration that makes it possible to achieve a new and better minimum.

Late pruning strategy
The complexity graph obtained with this experiment shows the same fact that in the middle strategy. When the model starts getting pruned, it experiences a sharp increase in the validation accuracy and a reduction in the training and validation losses. With this increase, the model achieves a the validation accuracy achieves a maximum value of approximately the 63%, which means a better performance than the middle pruning experiment, but still worse than the early pruning.

The results of this work have proven that, in knowledge distillation, it is possible to obtain a much smaller Student model from a original Teacher model, that mimics its behaviour. In comparison with the teacher, the student have a decrease in the accuracy that can be reduced by selecting a slightly larger and more complex model, adding a certain number of layers or modules per layer. Regarding neural network pruning, it has been proven that, for a practical structured pruning procedure, using different targets of pruned parameters for each layer according to the structure of the model results in a powerful procedure capable of reducing the size of a model in high percentages while maintaining its accuracy.

Chapter 5

Conclusions and future work

5.1 Conclusions

At the beginning of this work it was discussed the problem that currently exists regarding the computation and process of data in small devices and two compression techniques were chosen to deal with this problem: knowledge distillation and neural network pruning.

Along this project it is proven the effectiveness of these techniques by compressing a state-of-the-art deep neural network. Each technique is based on a different principle and therefore, it is possible to combine them in order to obtain a better performance.

The aim of the experiments regarding knowledge distillation have been to achieve a compressed model that shows a suitable performance in a classification problem and understand which parameters or structures should be optimized to improve its accuracy or reduce its size. The neural network pruning experiments, which start from the model obtained from the knowledge distillation, deals with the training process and the size of the model. It has also been proved that adjusting the parameters of these experiments it is possible to improve its efficiency and therefore, achieve a higher compression and a better performance of the model.

To conclude, this project proves that it is possible to deploy software compressive optimization techniques in order to use reliable deep neural networks in devices with low computing power.

5.2 Future work

The purposed future work is related to the use of the two studied optimization techniques in an emulation environment.

Distributed learning is an approach of Machine Learning in which a neural network is trained in different devices. There are different procedures to accomplish this process but the proposed future work should focus in only one of them.

Using Mininet [Fontes et al., 2015], which is presented in the Appendix A, it is possible to create a network in which different nodes can run a computing process while they communicate. Deploying the knowledge distillation online technique as a complete training process, it is possible to train the Teacher model in one station of the network while it communicates with another station, in which the student is trained. This is a Distributed learning technique, since, understanding the training process of the Teacher and the Student models as a whole, the training is fulfilled in two different nodes of a network.

Therefore, the main objective would be to apply the online knowledge distillation to a teacher and a student which are located in different stations and prune them individually while training. Then it would be interesting to measure the quality of the communications between the Teacher and the Student models by using a specialized software such as SCAPY [Biondi and the Scapy community., 2019] (Appendix B).

Appendix A

Mininet

Mininet [Fontes et al., 2015] is a network emulator which allows creating networks using controllers, switches, hosts and links. These networks can be modularized, since hosts and switched work independently. It also provides a Python API, which makes it very simple to create new and customized networks.

Every host included in Mininet run a standard Linux network software, what means they can also run real code. Every link that connect a host and a switch is based are based on virtual ethernet pairs. Therefore, it is possible to run network wide-tests in any node.

In order to create a network, there are several parameters that must be adjusted. Then, Mininet offers a wide range of possibilities to run the network, such as calculating the ping, running the iperf tool, which allows a multiple variety of opportunities or debugging the topology.

Appendix B

SCAPY

SCAPY [Biondi and the Scapy community., 2019] is a powerful interactive packet manipulation program. It is designed for Python and it allows to manage packages from a wide number of protocols. With this program is possible to capture, decode and analyse packets.

It works by performing two tasks: sending packets and receiving answers. Doing this, it is possible to trace a communication between two or more nodes and analyse its packets. In fact, SCAPY is a powerful tool to compute throughputs at any layer of a protocol or latencies which makes it very interesting to test communications.

Appendix C

Definition of the models

Listing C.1: Definition of the VGG16 Teacher model.

```
class VGG16(nn.Module):
    def __init__(self):
        super(VGG16, self).__init__()
        self.features0 = nn.Conv2d(in_channels=3, out_channels=64,
    kernel_size=3, padding=1, bias=False)
        self.features2 = nn.Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1, bias=False)

        self.features5 = nn.Conv2d(in_channels=64, out_channels=128,
    kernel_size=3, padding=1, bias=False)
        self.features7 = nn.Conv2d(in_channels=128, out_channels=128,
    kernel_size=3, padding=1, bias=False)

        self.features10 = nn.Conv2d(in_channels=128, out_channels=256,
    kernel_size=3, padding=1, bias=False)
        self.features12 = nn.Conv2d(in_channels=256, out_channels=256,
    kernel_size=3, padding=1, bias=False)
        self.features14 = nn.Conv2d(in_channels=256, out_channels=256,
    kernel_size=3, padding=1, bias=False)

        self.features17 = nn.Conv2d(in_channels=256, out_channels=512,
    kernel_size=3, padding=1, bias=False)
        self.features19 = nn.Conv2d(in_channels=512, out_channels=512,
    kernel_size=3, padding=1, bias=False)
        self.features21= nn.Conv2d(in_channels=512, out_channels=512,
    kernel_size=3, padding=1, bias=False)

        self.features24 = nn.Conv2d(in_channels=512, out_channels=512,
    kernel_size=3, padding=1, bias=False)
```

```
        self.features26 = nn.Conv2d(in_channels=512, out_channels=512,
    kernel_size=3, padding=1, bias=False)
        self.features28 = nn.Conv2d(in_channels=512, out_channels=512,
    kernel_size=3, padding=1, bias=False)

        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))

        self.classifier0 = nn.Linear(25088, 4096, bias=False)
        self.classifier3 = nn.Linear(4096, 4096, bias=False)
        self.classifier6 = nn.Linear(4096, 10, bias=False)

    def forward(self, x):
        x = F.relu(self.features0(x))
        x = F.relu(self.features2(x))
        x = self.maxpool(x)
        x = F.relu(self.features5(x))
        x = F.relu(self.features7(x))
        x = self.maxpool(x)
        x = F.relu(self.features10(x))
        x = F.relu(self.features12(x))
        x = F.relu(self.features14(x))
        x = self.maxpool(x)
        x = F.relu(self.features17(x))
        x = F.relu(self.features19(x))
        x = F.relu(self.features21(x))
        x = self.maxpool(x)
        x = F.relu(self.features24(x))
        x = F.relu(self.features26(x))
        x = F.relu(self.features28(x))
        x = self.maxpool(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = F.relu(self.classifier0(x))
        x = F.dropout(x, 0.55)
        x = F.relu(self.classifier3(x))
        x = F.dropout(x, 0.55)
        x = self.classifier6(x)
        return x
```

Listing C.2: Definitioon of the Student model.

```
class StudentNet(nn.Module):
    def __init__(self):
        super(StudentNet, self).__init__()
        self.conv1=nn.Conv2d(in_channels=3,out_channels=8,kernel_size=3,
    padding=1,stride=1,bias=False)
        self.conv2=nn.Conv2d(in_channels=8,out_channels=16,kernel_size=3,
    padding=1,stride=1,bias=False)
        self.conv3=nn.Conv2d(in_channels=16,out_channels=32,kernel_size=3,
    padding=1,stride=1,bias=False)
        self.conv4=nn.Conv2d(in_channels=32,out_channels=64,kernel_size=3,
    padding=1,stride=1,bias=False)
        self.conv4_1=nn.Conv2d(in_channels=64,out_channels=64,kernel_size=3,
     padding=1,stride=1,bias=False)
        self.conv5=nn.Conv2d(in_channels=64,out_channels=128,kernel_size=3,
    padding=1,stride=1,bias=False)
        self.conv5_1=nn.Conv2d(in_channels=128,out_channels=128,kernel_size=
    3, padding=1,stride=1,bias=False)

        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.pool = nn.MaxPool2d(kernel_size=2,stride=2)

        self.fc1=nn.Linear(7*7*128,100,bias=False)
        self.fc2=nn.Linear(100,100,bias=False)
        self.fc2_1=nn.Linear(100,100,bias=False)
        self.fc3=nn.Linear(100,50,bias=False)
        self.fc4=nn.Linear(50,10,bias=False)

    def forward(self,x):
        x=self.pool(F.relu(self.conv1(x)))
        x=self.pool(F.relu(self.conv2(x)))
        x=self.pool(F.relu(self.conv3(x)))
        x=F.relu(self.conv4(x))
        x=self.pool(F.relu(self.conv4_1(x)))
        x=F.relu(self.conv5(x))
        x=self.pool(F.relu(self.conv5_1(x)))
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x=F.relu(self.fc1(x))
        x = F.dropout(x, 0.5)
        x=F.relu(self.fc2(x))
        x = F.dropout(x, 0.5)
        x=F.relu(self.fc2_1(x))
        x = F.dropout(x, 0.5)
        x=F.relu(self.fc3(x))
        x = F.dropout(x, 0.5)
        x=self.fc4(x)
        return x
```

Appendix D

Main code relating the Teacher model

Listing D.1: Tranfer learning of the VGG16 model.

```
bias_teacher_model = models.vgg16(pretrained=True)
no_bias_teacher_model = VGG16()

#We adjust the final layer of the vgg16 pretrained model
input_last_layer = bias_teacher_model.classifier[6].in_features
bias_teacher_model.classifier[6] = nn.Linear(input_last_layer, 10)

#We copy the state dictionary of the original model.
bias_state_dict =  copy.deepcopy(bias_teacher_model.state_dict())
no_bias_state_dict = {}

#We need to tune the name of the modules in order to copy the state
    dictionary without biases into the new VGG16 model
for key, value in bias_state_dict.items():
    if 'bias' not in key:
        if 'features' in key:
            no_bias_state_dict[key.replace("s.","s")] = value
        elif 'classifier' in key:
            no_bias_state_dict[key.replace("r.","r")] = value

no_bias_teacher_model.load_state_dict(no_bias_state_dict)

#Finally, here we have the VGG16 teacher model without biases
teacher_model = no_bias_teacher_model

del bias_teacher_model
del bias_state_dict
```

Listing D.2: Structured pruning loop using the percentage of pruned nodes obtained from the unstructured pruning experiment.

```
if epoch > 10:
    for name, module in model.named_modules():
        if isinstance(module, nn.Conv2d):
            if name == 'features28':
                prune.ln_structured(module, name="weight", amount=0.03, n=1,
 dim=0)
                prune.remove(module, 'weight')
            elif name == 'features26':
                prune.ln_structured(module, name="weight", amount=0.01, n=1,
 dim=0)
                prune.remove(module, 'weight')
        elif isinstance(module, nn.Linear):
            if name == 'classifier0':
                prune.ln_structured(module, name="weight", amount=0.05, n=1,
 dim=0)
                prune.remove(module, 'weight')
            elif name == 'classifier3':
                prune.ln_structured(module, name="weight", amount=0.01, n=1,
 dim=0)
                prune.remove(module, 'weight')
        if epoch % 2 == 0:
            if isinstance(module, nn.Conv2d):
                if epoch < 20:
                    if name == 'features24':
                        prune.ln_structured(module, name="weight", amount=0.
02, n=1, dim=0)
                        prune.remove(module, 'weight')
                if epoch < 18:
                    if name == 'features21':
                        prune.ln_structured(module, name="weight", amount=0.
02, n=1, dim=0)
                        prune.remove(module, 'weight')
                if epoch < 16:
                    if name == 'features19' or name == 'features17' or name
 == 'features14' or name == 'features12' or name == 'features10' or name
 == 'features7' or name == 'features5' or name == 'features2' or name == '
features0':
                        prune.ln_structured(module, name="weight", amount=0.
02, n=1, dim=0)
                        prune.remove(module, 'weight')
```

Appendix E

Main code relating the Student model.

Listing E.1: Distillation loss function of the Student model.

```
def distillation_loss(scores,targets,T=5):
    soft_pred=F.softmax(scores/T,dim=1)
    soft_targets=F.softmax(targets/T,dim=1)
    loss=F.mse_loss(soft_pred,soft_targets)
    return loss
```

Listing E.2: Structured pruning of the Student model following the static approach.

```
if epoch > 200 and epoch < 280:
    if epoch % 7 == 0:
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                    prune.ln_structured(module, name="weight", amount=0.02,
    n=1, dim=0) #before 0.014
                    prune.remove(module, 'weight')
            elif isinstance(module, nn.Linear):
                if name != 'fc4':
                    prune.ln_structured(module, name="weight", amount=0.02,
    n=1, dim=0)
                    prune.remove(module, 'weight')
```

Listing E.3: Early pruning strategy followed in the dynamic structured pruning approach.

```
if epoch > 10 and epoch < 220:
    if epoch % 10 == 0:
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                if name == 'conv5_1':
                    prune.ln_structured(module, name="weight", amount=0.02, n=1, dim=0)
                    prune.remove(module, 'weight')
            elif isinstance(module, nn.Linear):
                if name == 'fc1':
                    prune.ln_structured(module, name="weight", amount=0.02, n=1, dim=0)
                    prune.remove(module, 'weight')
if epoch == 50 or epoch == 150:
    for name, module in model.named_modules():
        if isinstance(module, nn.Conv2d):
            if name == 'conv4' or name == 'conv5':
                prune.ln_structured(module, name="weight", amount=0.02, n=1, dim=0)
                prune.remove(module, 'weight')
        elif isinstance(module, nn.Linear):
            if name == 'fc2':
                prune.ln_structured(module, name="weight", amount=0.02, n=1, dim=0)
                prune.remove(module, 'weight')
if epoch == 60 or epoch == 120 or epoch == 180 or epoch == 240:
    for name, module in model.named_modules():
        if isinstance(module, nn.Conv2d):
            if name == 'conv3':
                prune.ln_structured(module, name="weight", amount=0.02, n=1, dim=0)
                prune.remove(module, 'weight')
        elif isinstance(module, nn.Linear):
            if name == 'fc2_1' or name == 'fc3':
                prune.ln_structured(module, name="weight", amount=0.02, n=1, dim=0)
                prune.remove(module, 'weight')
if epoch == 50 or epoch == 100 or epoch == 150 or epoch == 200 or epoch == 250:
    for name, module in model.named_modules():
        if isinstance(module, nn.Conv2d):
            if name == 'conv4_1':
                prune.ln_structured(module, name="weight", amount=0.02, n=1, dim=0)
                prune.remove(module, 'weight')
```

Listing E.4: Middle pruning strategy followed in the dynamic structured pruning approach.

```
if epoch > 100:
    if epoch % 10 == 0:
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                if name == 'conv5_1':
                    prune.ln_structured(module, name="weight", amount=0.02,
   n=1, dim=0)
                    prune.remove(module, 'weight')
            elif isinstance(module, nn.Linear):
                if name == 'fc1':
                    prune.ln_structured(module, name="weight", amount=0.02,
   n=1, dim=0)
                    prune.remove(module, 'weight')
    if epoch == 101 or epoch == 201:
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                if name == 'conv4' or name == 'conv5':
                    prune.ln_structured(module, name="weight", amount=0.02,
   n=1, dim=0)
                    prune.remove(module, 'weight')
            elif isinstance(module, nn.Linear):
                if name == 'fc2':
                    prune.ln_structured(module, name="weight", amount=0.02,
   n=1, dim=0)
                    prune.remove(module, 'weight')
    if epoch == 120 or epoch == 160 or epoch == 200 or epoch == 240:
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                if name == 'conv3':
                    prune.ln_structured(module, name="weight", amount=0.02,
   n=1, dim=0)
                    prune.remove(module, 'weight')
            elif isinstance(module, nn.Linear):
                if name == 'fc2_1' or name == 'fc3':
                    prune.ln_structured(module, name="weight", amount=0.02,
   n=1, dim=0)
                    prune.remove(module, 'weight')
    if epoch == 110 or epoch == 140 or epoch == 170 or epoch == 200 or epoch
    == 230:
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                if name == 'conv4_1':
                    prune.ln_structured(module, name="weight", amount=0.02,
   n=1, dim=0)
                    prune.remove(module, 'weight')
```

Listing E.5: Late pruning strategy followed in the dynamic structured pruning approach.

```
if epoch > 200:
    if epoch % 5 == 0:
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                if name == 'conv5_1':
                    prune.ln_structured(module, name="weight", amount=0.02,
    n=1, dim=0)
                    prune.remove(module, 'weight')
            elif isinstance(module, nn.Linear):
                if name == 'fc1':
                    prune.ln_structured(module, name="weight", amount=0.02,
    n=1, dim=0)
                    prune.remove(module, 'weight')
    if epoch == 210 or epoch == 220:
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                if name == 'conv4' or name == 'conv5':
                    prune.ln_structured(module, name="weight", amount=0.02,
    n=1, dim=0)
                    prune.remove(module, 'weight')
            elif isinstance(module, nn.Linear):
                if name == 'fc2':
                    prune.ln_structured(module, name="weight", amount=0.02,
    n=1, dim=0)
                    prune.remove(module, 'weight')
    if epoch == 210 or epoch == 230 or epoch == 250 or epoch == 270:
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                if name == 'conv3':
                    prune.ln_structured(module, name="weight", amount=0.02,
    n=1, dim=0)
                    prune.remove(module, 'weight')
            elif isinstance(module, nn.Linear):
                if name == 'fc2_1' or name == 'fc3':
                    prune.ln_structured(module, name="weight", amount=0.02,
    n=1, dim=0)
                    prune.remove(module, 'weight')
    if epoch == 220 or epoch == 230 or epoch == 240 or epoch == 250 or epoch
    == 260:
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                if name == 'conv4_1':
                    prune.ln_structured(module, name="weight", amount=0.02,
    n=1, dim=0)
                    prune.remove(module, 'weight')
```

Bibliography

S. Anwar, K. Hwang, and W. Sung. Structured pruning of deep convolutional neural networks, 2015. URL `https://arxiv.org/abs/1512.08571`.

P. Biondi and the Scapy community. Scapy. `https://scapy.readthedocs.io/en/latest/introduction.html`, 2019. [Online; accessed 14-dec-2021].

D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag. What is the state of neural network pruning?, 2020. URL `https://arxiv.org/abs/2003.03033`.

A. Bragagnolo and C. A. Barbano. Simplify: A python library for optimizing pruned neural networks. *SoftwareX*, 17, 2022. URL `https://doi.org/10.1016/j.softx.2021.100907`.

C. Bucilua, R. Caruana, and A. Niculescu-Mizil. *Model compression*. Association for Computing Machinery, 2006. URL `https://doi.org/10.1145/1150402.1150464`.

T. Chen, B. Ji, T. Ding, B. Fang, G. Wang, Z. Zhu, L. Liang, Y. Shi, S. Yi, and X. Tu. *Only Train Once: A One-Shot Neural Network Training And Pruning Framework*, volume 34. Curran Associates, Inc., 2021. URL `https://proceedings.neurips.cc/paper/2021/file/a376033f78e144f494bfc743c0be3330-Paper.pdf`.

R. R. Fontes, S. Afzal, S. H. B. Brito, M. Santos, and C. E. Rothenberg. Mininet-wifi: Emulating software-defined wireless networks. *2nd International Workshop on Management of SDN and NFV Systems 2015*, 2015. URL `https://ieeexplore.ieee.org/document/7367387`.

A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. A survey of quantization methods for efficient neural network inference, 2021. URL `https://arxiv.org/abs/2103.13630`.

F. Girosi, M. Jones, and T. Poggio. Regularization theory and neural networks architectures. *Neural Computation*, 7(2):219–269, 03 1995. ISSN 0899-7667. doi: 10.1162/neco.1995.7.2.219. URL `https://doi.org/10.1162/neco.1995.7.2.219`.

J. Gou, B. Yu, S. J. Maybank, and D. Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, mar 2021. doi: 10.1007/s11263-021-01453-z. URL `https://doi.org/10.1007/2Fs11263-021-01453-z`.

S. Han, J. Tran, and W. Dally. Learning both weights and connections for efficient neural networks. *Journal of Systems Architecture*, 122:357–362, 2022. URL `https://doi.org/10.1016/j.sysarc.2021.102336`.

C. Harris, K. Millman, and S. Van der Walt. Array programming with numpy. *Nature*, 585:357–362, 2020. URL `https://doi.org/10.1038/s41586-020-2649-2`.

G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network, 2015. URL `https://doi.org/10.48550/arXiv.1503.02531`.

Y. Huang and Y. Chen. Autonomous driving with deep learning: A survey of state-of-art technologies, 2020. URL `https://arxiv.org/abs/2006.06091`.

Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1: 541–551, 1989.

X. Li, W. Hu, C. Li, T. Jiang, H. Sun, X. Li, X. Huang, and M. Grzegorzek. A state-of-the-art survey of artificial neural networks for whole-slide image analysis:from popular convolutional neural networks to potential visual transformers, 2021. URL `https://arxiv.org/abs/2104.06243`.

G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. van der Laak, B. van Ginneken, and C. I. Sánchez. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 42:60–88, dec 2017. doi: 10.1016/j.media.2017.07.005. URL `https://doi.org/10.1016/2Fj.media.2017.07.005`.

Y. Matsubara, M. Levorato, and F. Restuccia. Split computing and early exiting for deep learning applications: Survey and research challenges, 2021. URL `https://arxiv.org/abs/2103.04505`.

T. Mitchell. *Machine Learning*. McGraw Hill, 1997. ISBN 0-07-042807-7.

P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference, 2016. URL `https://arxiv.org/abs/1611.06440`.

M. Paganini and J. Forde. On iterative neural network pruning, reinitialization, and the similarity of masks, 2020. URL `https://arxiv.org/abs/2001.05050`.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killen, and Z. Lin. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8024–8035, 2019.

M. Riera, J. Arnau, and A. González. Dnn pruning with principal component analysis and connection importance estimation. *NIPS*, pages 357–362, 2015. URL `https://doi.org/10.1016/j.sysarc.2021.102336`.

V. Roger, J. Farinas, and J. Pinquier. Deep neural networks for automatic speech processing: A survey from large corpora to limited data, 2020. URL `https://arxiv.org/abs/2003.04241`.

H. Schulz and S. Behnke. Deep learning. *KI Kunstliche Intelligenz*, pages 357–363, 2012. URL `doi:10.1007/s13218-012-0198-z`.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2015.

R. K. Sinha, R. Pandey, and R. Pattnaik. Deep learning for computer vision tasks: A review, 2018. URL `https://arxiv.org/abs/1804.03928`.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

H. Tessier. Neural network pruning 101: All you need to know not to get lost. *Towards Data Science*, 2021. URL `https://towardsdatascience.com/neural-network-pruning-101-af816aaea61`.

A. Torfi, R. A. Shirvani, Y. Keneshloo, N. Tavaf, and E. A. Fox. Natural language processing advancements by deep learning: A survey, 2020. URL `https://arxiv.org/abs/2003.01200`.

P. J. Werbos. *The Roots of Backpropagation : From Ordered Derivatives to Neural Networks and Political Forecasting.* John Wiley and Sons, 1994. ISBN 0-471-59897-6.

J. West, D. Ventura, and S. Warnick. Spring research presentation: A theoretical foundation for inductive transfer, 2007.

S. Yang, Y. Wang, and X. Chu. A survey of deep learning techniques for neural machine translation, 2020. URL `https://arxiv.org/abs/2002.07526`.

M. Zhang, F. Zhang, N. D. Lane, Y. Shu, X. Zeng, B. Fang, S. Yan, and H. Xu. Deep learning in the era of edge computing: Challenges and opportunities, 2020. URL `https://arxiv.org/abs/2010.08861`.