UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

# Compression of Convolutional Neural Networks using Tucker Decomposition

*Author:*
Cristina GRANÉS SANTAMARIA

*Supervisor:*
José M. ÁLVAREZ

*A thesis submitted in fulfillment of the requirements for the degree of Master on Telecommunications*

*at*

DATA 61
CSIRO (Australia)

May 23, 2017

# *Abstract*

Nowadays, deep neural networks are being introduced in mobile devices where memory space and computation speed is limited. Therefore, the need for more compression in those networks has increased. In this work, we explore a way to compress the convolutional neural networks by reducing the number of weights in their convolutional layers. This method consists on the Tucker decomposition of tensors introduced on the kernels of the convolutional layers.

Two different architectures, one with three convolutional layers and another with five, are the base of our study. We employ the Tucker decomposition in different forms, with more or less compression. We compute their compression rates versus the accuracy and loss (training and test) for the CIFAR-10 dataset, in order to see how good the performance of the compressed models is with respect to the original. The decomposition achieves its best results when compressing partially each kernel layer, up to 70%.

Finally, we visualize the probability density function of the filters, contrasting its shape in function of the compression. From those plots, we conclude that, not only we are able to reduce the number of parameters, but also the number of bits needed to store them if we use entropy encoding.

# *Resumen*

Últimamente, se van introduciendo las redes neuronales en dispositivos móviles, cuándo su memoria y velocidad de computación son limitados. Entonces, esto acompaña a la necesidad de comprimir esas redes neuronales. En este proyecto, se explora una forma de comprimir redes neuronales convolucionales reduciendo el número de pesos en sus capas convolucionales. Este método consiste en la descomposición de tensores Tucker, que se introduce en los núcleos de las capas convolucionales.

Dos estructuras diferentes, una con tres capas convolucionales y otra con cinco, se usan como base del estudio. Se emplea la descomposición de Tucker de formas distintas, con más o menos compresión. Se computen las pérdidas y la precisión (en entrenamiento y test) en función del ratio de compresión del modelo para la base de datos CIFAR-10, para ver su rendimiento con respeto al modelo original. La descomposición alcanza sus mejores resultados cuando se comprime parcialmente cada capa, hasta una compresión del $\sim$70%.

Finalmente, se visualiza la forma de la función de densidad de probabilidad de los pesos en función de la compresión. A partir de esto, se conclude que, no solo se puede reducir el número de parámetros sino también el número de bits usados para almacenarlos, si usamos un método de codificación entrópica.

# *Resum*

Últimament es van introduïnt les xarxes neuronals als dispositius móbils, quan la seva capacitat i velocitat de computació estan limitades. Això acompanya a la necessitat d'introduïr la compressió d'aquestes xarxes neuronals. Amb aquest projecte, s'explora una forma de comprimir les xarxes neuronals convolucionals, reduïnt el nombre de pesos a les seves capes convolucionals. Aquest mètode consisteix en la descomposió de tensors anomenada Tucker, que s'introdueix als nuclis de les capes convolucionals.

Dues extructures diferents, una amb tres capes convolucionals i una altra amb cinc, s'usen com a base d'estudi. Es duu a terme la descomposició de Tucker de diferents maneres, amb més o menys compressió. Es computen les pèrdues i precisions (en l'entrenament i el test) de la base de dades CIFAR-10, per veure quin és el seu rendiment en funció del model original. La descomposició aconsegueix millors resultats quan es comprimeix parcialment cada capa, fins a una compressió del $\sim$70%.

Finalment, es visualitza la forma de la funció de densitat de probabilitat dels pesos en funció de la compressió. A partir d'aquí, es conclueix que, no només es poden reduïr el número de paràmetres, sinò també el nombre de bits usats per emmagatzemar-los, si usem un mètode de codificació entròpica.

# *Acknowledgements*

First of all, I must thank my supervisor, Dr. José M. Álvarez for his help and his tolerance during this project.

I thank Dr. Xavier Giró-i-Nieto for his last advise and help.

I should also thank my family and friends for their constant support and motivation.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **CNN** | **C**onvolutional **N**eural **N**etwork |
| **CONV** | **CONV**olutional layer |
| **POOL** | **POOL**ing layer |
| **NORM** | **NORM**alization layer |
| **FC** | **F**ully-**C**onnected layer |
| **ReLU** | **Re**ctified **L**inear **U**nit |
| **BP** | **B**ack-**P**ropagation |
| **SVD** | **S**ingular **V**alue **D**ecomposition |
| **VBMF** | **V**ariational **B**ayesian **M**atrix **F**actorization |
| **RMSProp** | **R**oot **M**ean **S**quare **Prop**agation |
| **RGB** | **R**ed **G**reen **B**lue |
| **PDF** | **P**robabilistic **D**ensity **F**unction |

# Chapter 1

# Introduction

As most of the human population owns smartphones with plenty of applications for different purposes, a lot of companies around the world are centered on introducing to the market their own applications covering new needs and tendencies. Also, new mobile devices to ease the life of people inside their home are introduced, such as *Alexa smart home* device from Amazon. Apart from the personal use of mobile devices, new mobile medical equipment or scanning-detecting devices are being introduced to simplify the operations. The demand of intelligent applications to cover necessities and to impress people is increasing constantly, therefore, the use of deep learning on mobile phone applications is increasing to cover these demands. Currently, the deployment of convolutional neural networks (CNNs) for computer vision tasks on mobile devices is gaining a lot of attention.

In the past years, the tendency in the electronics of all those fields is to decrease its size and increase its performance. So, one of the most critical issues is that mobile devices have strict constraints in terms of computing power, battery and memory capacity. In the past decades, the use of neural networks is mostly external to the mobile devices and usually need a lot of memory space for training, but, in a mobile device, we seek for the minimum space used by the networks. On mobile applications, the training of the network is done on a server and only the test or inference is executed on the mobile devices. Deep neural networks are known to be over-parameterized (to have too many parameters), which facilitates convergence to local minimums of the loss function during training; one of the things that is not desired (Denil et al., 2013). Then, several methods are introduced in order to reduce the number of parameters, also called **compression** of the networks.

This project intends to find a new method to compress the convolutional layers in convolutional neural networks by using the Tucker decomposition (Tucker, 1966) on the set of filters or weights (tensors). This method has been implemented after training by Kim et al., 2015, but the aim of this project is to apply tune parameters that are already compressed. This is supposed to conclude with better results than compressing after training.

The structure of this work is the following: Chapter 2 begins with the state of the art of the compression methods applied in the past few years. In chapter 3, we go through all the important theoretical and mathematical background to understand the project. Then, in chapter 4, we see how to apply the tucker decomposition on the convolutional layers, we go through the dataset and the architectures used in the study and how varying some parameters affects our results, such as the initialization of the parameters. In 5 we see how different types of compression affects our two models and arrive to some conclusions. Finally, the conclusions and future

work can be found in Chapter 6.

# Chapter 2

# State of the Art

The compression of neural networks has been applied plenty of times in different manners in order to reduce the computation time (either training or testing) or the memory needed to store the network architecture.

In Convolutional Neural Networks (CNNs), the first method applied was the introduction of pooling layers after some layers in the architecture. Pooling layers reduce the spatial size of the outputs from the previous layer using the MAX operation (substituting the values in a region or pooling size by the maximum value in that region). The pooling was introduced with the idea to reduce the number of parameters in the last layers, the ones with a higher number of parameters, remembering how they are fully-connected with the previous layer (named fully-connected layers).

Another method tried is the sharing of some parameters between neurons. This makes sense in the image field because the some features can be found in different parts of the image in particular cases (*LeNet tutorial*). But, this method does not always work, as, for example, in cases where images have very centralized particular objects, such as images with centralized human faces or analysis of human bodies in a medical application (each part: heart, brain, lungs, ... , is at the same position in different bodies).

Lin, Chen, and Yan, 2013 introduced the 1x1 convolutions, instead of using $N \times N$ convolutions with $N > 1$. This makes sense because the input images of CNNs have 3-dimensions (height, weight and depth, being the Red-Green-Blue, RGB, color format). With those 1x1 convolutions we have less parameters, as the filters have smaller size (filter of size 1).

Another method with the intention of compression is the introduction of a new hyperparameter in the CONV layers, named *dilatation*. This dilatation allows to increase the size of the effective receptive field or filter size, substituting a larger filter by a smaller filter with spaces between each cell (the size of those spaces is the dilatation). An example of an article analyzing the dilatation would be Yu and Koltun, 2015.

Recently, there are several studies that apply low-rank approximations to speed-up CNNs by exploiting the redundancy of the parameters (Denton et al., 2014, Lebedev et al., 2014, Jaderberg, Vedaldi, and Zisserman, 2014). Those compressions typically focus on the convolutional layers, as they are the ones with a higher computational cost. This introduces the challenge of **network compression**, aiming to

compress the full network.

Denton et al., 2014 showed that the weight matrix of a fully-connected layer can be compressed by applying the truncated singular value decomposition (t-SVD) without significant decrease in the prediction accuracy. Other methods appeared afterwards with better results.

Recently, various methods are being applied, such as methods based on vector quantization and k-means clustering (Gong et al., 2014), hashing techniques (Chen et al., 2015), application of the Generalized Singular Value Decomposition (GSVD) instead of the t-SVD (Zhang et al., 2016) or pruning and Huffman coding on trained quantization (Han, Mao, and Dally, 2015).

Zhang et al., 2016 showed that CONV layers can be accelerated with asymmetric 3d decomposition. They decompose the $D \times D$ receptive fields to three receptive fields of sizes $D \times 1$, $1 \times D$ and $1 \times 1$. In a similar way, 1D convolutions are applied at (Alvarez and Petersson, 2016).

The weights in a CONV layer can be represented as a tensor and, therefore, apply tensor decompositions on such tensors. Some tensor decompositions are CANDECOMP, PARAFAC (Carroll and Chang, 1970, Shashua and Hazan, 2005) and TUCKER (Tucker, 1966, Kim and Choi, 2007).

At Kim et al., 2015, Tucker-2 decomposition is applied after training following the steps:

1. Selection of the rank for the weights tensor with a variational Bayesian matrix factorization.
2. Application of the Tucker-2 decomposition on the weights tensor (decomposing the two last dimensions: input and output).
3. Fine-tuning of the network to recover the accumulated loss of the accuracy.

In this project, the purpose is to analyze the full tucker decomposition on the weights and tune the network once we have fixed this decomposition. Then, the rank selection is done by intuition and the fine-tunning of the network is not necessary.

# Chapter 3

# Theoretical Background

## 3.1 Artificial Neural Networks

Artificial neural networks are computational models commonly used nowadays in computer science and other fields based on the human brain. They are based on a collection of many neural units or artificial neurons connected between each other to form the neural network infrastructure an give a final required output. Each individual neuron has an input and an output, the second one obtained from a particular function applied to the input. Also, the full neural network has an input or several inputs given to a set of neurons (first layer of the network) and an output or outputs (result of the last layer in the network).

The following image shows a scheme of an **artificial neuron** that, as the neurons of a human brain, has an input $\mathbf{x}$ and an output $\mathbf{y}$ function of $\mathbf{x}$. Apart from an input and an output, a neuron has a set of weights and a bias, which are parameters that need to be adjusted in order to obtain the best possible output according to an error function. We have as many weights as inputs (weight or importance we give to each input), which, multiplied to each input, are added to the bias. The result of this addition is transferred to a function $f$, called **activation function**, that gives us the output of the neuron. Examples of the activation function are: the identity, the binary step, the $tanh$, the sigmoid function and the rectifier linear unit (ReLU).

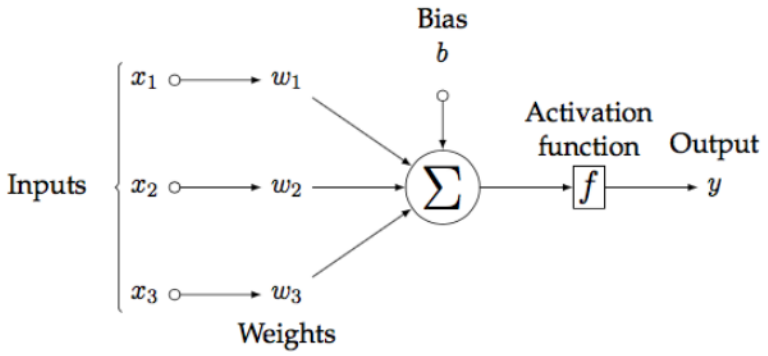


FIGURE 3.1: Artificial Neuron (from Puente, 2016)

As an example, the result $y$ of a neuron, given an input $\mathbf{x} = [x_1, x_2, ..., x_N]$, weights $\mathbf{w} = [w_1, w_2, ..., w_N]$, bias $b$ and the sigmoid as nonlinear function, would be equation 3.1.

$$y = f(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}+b}} \tag{3.1}$$

The result of the neuron can be interpreted as the application of an hyperplane of an hyperspace $\mathcal{R}^N$ with function $w_1x_1 + w_2x_2 + ... + w_Nx_N + b = 0$. Then, the weights can be seen as the parameters that control the rotation and skew of the hyperplane in the hyperspace and the bias can be seen as the translation from the origin of this hyperplane. This hyperplane separates the hyperspace into two different regions, $R_0$ and $R_1$. Therefore, selecting different values of $\mathbf{w}$ and $b$ gives us different separations of the hyper-space $\mathcal{R}^N$. After we apply the nonlinear function, we can interpret the result as splits of probabilities: a point in the hyperspace has a higher probability of forming part of a sub-region depending on where is situated in the hyperspace. Figure 3.2 shows the hyper-space in the 2-dimensional case.

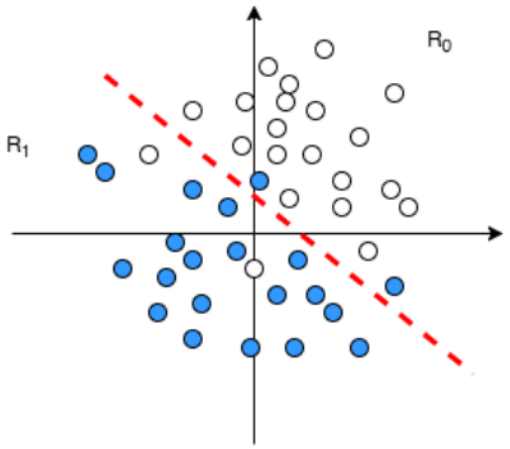


FIGURE 3.2: Example of how the hyperplane (in red) splits the hyper-space into two regions, $R_0$ and $R_1$

If we give outputs $\{0, 1\}$, with $y = 0$ the points in $R_0$ and $y = 1$ the points in $R_1$, we can interpret this separation as a split between classes, being the points situated in region $R_0$ the ones with higher probability of pertaining to class 1 while the ones in $R_1$ would have higher probability of pertaining to class 2. In these terms, we classify or choose whether a point is part of a class or another class. We can obtain the probability of having output 1 given the input with the equation 3.2.

$$P(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}+b}} = \frac{e^{\mathbf{w}^T\mathbf{x}+b}}{1 + e^{\mathbf{w}^T\mathbf{x}+b}} \tag{3.2}$$

The purpose of an artificial neural network is to solve problems in a similar way as the human brain would solve them. For example, as our eyes see a cat and our brain analyzes the image recognizing the cat, we want a neural network to do a similar work in terms of recognition (from the image of a cat, we want the network to give as output the label of class *cat*). Or, as our brain is capable of detecting a particular object in a room, we want a neural network to detect different objects on an image.

By using a *single perceptron*, which consists on a binary split of the hyper-space, we can only classify between two classes. In order to classify $N$ classes, we need an $inputlayer$ that applies several splits (hyperplanes or neurons, at least one for each class) plus an output layer that goes from those splits to the number of outputs we are expecting (for example, the number of classes in the case of classification), as pictured in 3.3.

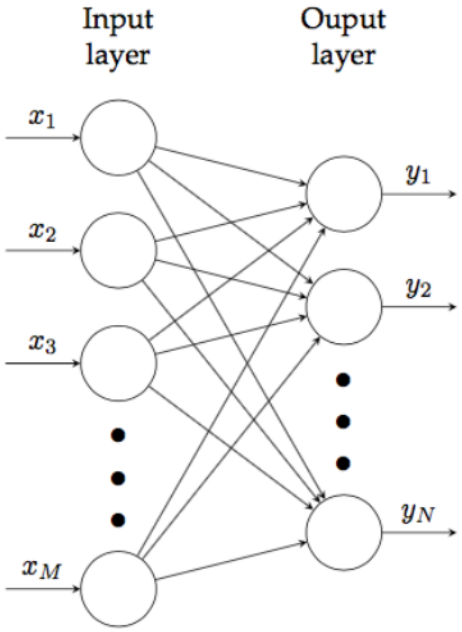


FIGURE 3.3: Logistic Regression (from Puente, 2016)

Given an input $\mathbf{x} = [x_1, x_2, ..., x_M]$ and having $N$ classes, we have an output vector $\mathbf{y} = [y_1, y_2, ..., y_N]$, with $y_n \in (0, 1), \forall n = \{1, 2, ..., N\}$ being the probability of $\mathbf{x}$ forming part of class $n$. We define $Y = k$ as the non-binary output indicating the class with maximum probability, $k$, with $y_k$ the mentioned probability, which consists on the result of the classification of the network. The output activation (output of the output layer) becomes the equation 3.3, named *softmax* function, with $k$ indicating the class $k$ and $P(y = k|$ the probability that the output $y$ indicates that the input is part of class $k$.

$$P(Y = k|\mathbf{x}) = \frac{e^{\mathbf{x}^T \mathbf{w_k}}}{\sum_{n=1}^{N} e^{\mathbf{x}^T \mathbf{w_n}}} \tag{3.3}$$

But this architecture has a limitation, because, using a single hyperplane, we are limited in terms of how the hyperspace is divided. The most common scenario is that classes are not separated by a straight line, as we can see in figure 3.2.

Therefore, we need several hyperplanes (neurons) in order to split the hyperspace with several hyperplanes and give us a better approximation of the real separation between two classes. At least two discriminating units or hyperplanes are required, the first one (*input layer*) to have a first linear split and the second one to apply the non-linear separation of the hyperspace. The second layer is called *hidden*

*layer*, the intermediate set of neurons between the *input layer* and the *output layer*. The network is, then, formed by packs of neurons that form different layers. The lightest architecture possible is the one formed by the minimum number of layers required to have an adequate separation between classes, which consists on 3 layers: *input layer*, *hidden layer* and *output layer*, as seen in figure 3.4.

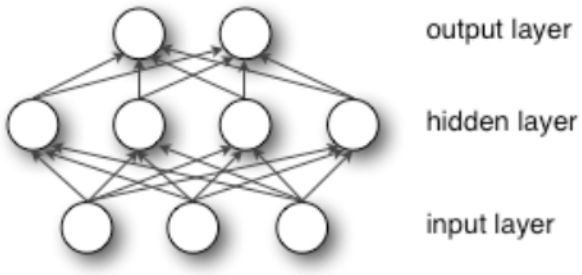


FIGURE 3.4: Multilayer Perceptron (from *Tutorial on Multilayer Perceptron* 2010)

## 3.2 Back-Propagation Algorithm

Once we have decided our architecture in function of the number of classes and inputs, we need to *"teach"* or select the different parameters of the network, which consist on the weights and the biases through the network. We can do this with labeled data, so, having a set of inputs with the corresponding correct outputs. This process is called **training** the network and consists on several steps using a back-propagation (BP) algorithm. To properly select the parameters, we define an *error function* or *cost function* of the network that we want to minimize with the back-propagation algorithm. For BP, the loss function calculates the difference between the input training example and its expected output, after the example has been propagated through the network. This function depends on the weights and the biases of the network, so we select the values of those parameters according to the algorithm. In this section, we will explain the BP algorithm in a similar manner as in the book Bishop, 1995.

The BP algorithm is based on the *backward* propagation of the errors through our network structure. It tries to correct or modify the different weights and biases along the network to minimize the different errors that may appear. We use the labeled data to train the network by iterating between the following two steps:

1. First, we compute the output of the network given an input from the labeled data and analyze the error, by comparing the result with the expected label.

2. We back-propagate through the network and change the weights and biases of each layer to minimize this error. This step has different implementations that lead to different forms of convergence.

In each layer of the network, we have different units or neurons that compute the different activations of the network as in equation 3.4, where $a_j$ is the activation of the neuron $j$, $z_i$ are the different inputs connected to this neuron with weight $w_{ji}$ and

$b_j$ is the bias. We then apply the non-linear function $f$ as in equation 3.5 to obtain the output of the neuron, $z_j$.

$$a_j = \sum_i w_{ji} z_i + b_j \tag{3.4}$$

$$z_j = f(a_j) \tag{3.5}$$

Each unit has an error $E^t$, where $t$ denotes the tuple $(input_t, output_t)$ of the labeled database, that depends on the different outputs of the neurons as $E^t = E^t(z_1, ..., z_P)$. The total error for a concrete input is denoted by the addition of all these errors $E = \sum_t E^t$. As $E$ is obtained from the addition, we can focus on a single tuple $t$. The BP algorithm is based on the Gradient Descent method, which consists on moving through the error function in a direction proportional to the negative of the gradient of the function at the current point. We know that the error depends on the weight $w_j i$ and we can apply the rule of the partial derivatives as in 3.6.

$$\frac{\partial E^t}{\partial w_j i} = \frac{\partial E^t}{\partial a_j} \frac{\partial a_j}{\partial w_j i} \tag{3.6}$$

From 3.4 we have:

$$\frac{\partial a_j}{\partial w_j i} = z_i \tag{3.7}$$

Therefore:

$$\frac{\partial E^t}{\partial w_j i} = \frac{\partial E^t}{\partial a_j} z_i = \delta_j z_i \tag{3.8}$$

The term $\delta_j$ is an error term, as the error is obtained by multiplying this term to the input $z_i$. To evaluate the error, this term has to be computed at each unit. This computation is different for each layer:

- In the output layer, where $z_j = y_j$, the evaluation of $\delta_j$ is as simple as:

$$\delta_j = \frac{\partial E^t}{\partial a_j} = g'(a_j) \frac{\partial E^t}{\partial y_j} \tag{3.9}$$

- In the case of the hidden layers, we use the chain rule for the partial derivatives and obtain:

$$\delta_j = \frac{\partial E^t}{\partial a_j} = \sum_n \frac{\partial E^t}{\partial a_n} \frac{\partial a_n}{\partial a_j} \tag{3.10}$$

  Where the sum is over all the units $n$ towards which the unit $j$ is connected. Applying the previous equations 3.4 and 3.5 we obtain:

$$\delta_j = g'(a_j) \sum_n w_{nj} \delta_n \tag{3.11}$$

The sub-steps for a single tuple $t$ of the BP algorithm are:

1. Inject the inputs $\mathbf{x}$ to the network and propagate the activations through the network (forward step).
2. Evaluate the $\delta_j$ for all the units in the *output layer*.

3. Back-propagate the $\delta$'s to obtain all the $\delta_j$'s of every *hidden* neuron.
4. Evaluate the derivatives with respect to the weights with the $z_i$'s as in equation 3.8.

We need to repeat the previous steps for each tuple $t$ in the database, so that we can finally obtain $E$ for all the labeled data:

$$\frac{\partial E}{\partial w_{ji}} = \sum_t \frac{\partial E^t}{\partial w_{ji}} \tag{3.12}$$

We repeat the operations to obtain $E$ several times and we keep updating the weights of the network each time or iteration $k$ with the Gradient Descent method. The updating of the weight $w_{ji}$ at the iteration $k$, applying 3.12 and 3.8, can be expressed as:

$$w_{ji}^{(k+1)} = w_{ji}^{(k)} - \Delta w_{ji}^{(k)} = w_{ji}^{(k)} - \eta_0 \frac{\partial E}{\partial w_{ji}} = w_{ji}^{(k)} - \eta_0 \sum_t \delta_j^t z_i^t \tag{3.13}$$

Being $w_{ji}^{(k+1)}$ the weight $w_{ji}$ in the $k+1$ step, $w_{ji}^{(k)}$ the weight $w_{ji}$ in the previous step $k$ and $\Delta w_{ji}^{(k)}$ the modification applied to the weight, which is given by the amount of error we subtract. The constant $\eta_0$ is called **learning rate** and is chosen before starting the training, along with $t$, the topology of the network and the number of iterations to update the weights. Those parameters selected ahead are called **hyperparameters** are they are usually tunned with intuition.

The training iterations can be separated into batches or a set of training examples in one *forward-backward* pass (one forward pass plus one backward pass), which leads to the definition of **batch size** or number of training examples in a batch. Once we have trained with all the training examples available in our dataset, we say we finished one *epoch*, that is, the ending of a *forward-backward* pass on all the examples. So, we can differentiate between an **iteration**, which is a *forward-backward* pass with *batch size* examples from the training set, and an **epoch**, which contains all the iterations needed to complete the *forward-backward* pass on all the training examples we have.

Regarding the learning rate $\eta_0$, we have to set it to a proper value that lets the model learn in a manner to converge as well as possible, but not too slowly. Typical values for this are in the range $(0.1, 0.001)$. If we select a high value, if may lead to divergence, increasing the training loss rather than minimizing it. On the other hand, a low learning rate will probably stuck the algorithm at a local minimum rather than going further to a better solution during the optimization.

Also, the initialization of the weights and biases is important for the convergence of the BP algorithm. There are different ways to initialize all the variables, but all of them consist on a random initialization so that the algorithm has a higher flexibility to converge.

When training a network we need to take into account the problem of **overfitting**, which consists on an over-training of the network. If we over-train the network, we will have a higher error when we want to apply the network on a set of data, different from the training set, because the over-training will specialize too

much our network on the specific training set. For this reason, it is important to set correctly the number of iterations and to stop the training process before over-fitting it. There are different methods that can be applied apart from interrupting the training after a number of iterations, but this is out of scope for this project.

Then, once we have selected the parameters (weights and biases) in the training step, we can **test** or analyze the error of our network using another group of labeled data, different from the one used in the training step. With this step, we can somehow evaluate how good our network performs on that type of data. As a consequence, the data is usually separated between two (train and test) or three (train, validation and test) groups, one is the one used to train and the others are used to validate or test the results. The validation set is used sometimes to set other parameters and train again.

## 3.3 Convolutional Neural Networks

After seeing how the *hidden layer* gives us the opportunity to have a non-linear separation in the hyperspace, the Deep Neural Network (DNN) was introduced, which consists on stacking many hidden layers to make the model deeper. This idea was introduced several years ago but not tried until recently due to the lack of computation capabilities applying the back-propagation algorithm on deeper architectures, as it led to local minimums that are more frequent with deeper architectures because we introduce more complex nonlinearity functions with much more "up's" and "down's", introducing more local minimums. Also, the computation capabilities in the past were not the same as nowadays and it took too long to train those type of networks.

With the introduction of the Internet, the amount of data has increased incredibly, and with the introduction of Graphical Processing Units (GPUs), the complex matrix operations are solved much faster and efficiently. New learning or training algorithms, such as the Newton's method or the Levenberg-Marquardt algorithm, are more efficient that the basic back-propagation algorithm and give a better propagation of the gradients, avoiding local minimums.

Convolutional Neural Networks (CNNs) are deep neural networks whose neurons perform a dot product with some inputs, operation that is usually followed by a non-linear function. The input consists on raw pixels of images and the output gives class scores. This type of network was first introduced in the 1990's by Yann LeCun, with his network LeNet, and his paper 'Gradient-based learning applied to document recognition', LeCun et al., 1998 is the documentation of the first applied Convolutional Neural Network LeNet-5.

While regular neural networks receive an input and transform it through certain hidden layers and a fully-connected layer as the output layer, these type of neural networks are not useful with images, as we would need too many parameters that may lead to over-fitting. Meanwhile, convolutional neural networks take advantage of the fact that the input is a set of images and form layers arranged in 3 dimensions (weight, height and depth). In this section, we will explain the architecture on CNNs

plus some characteristics. A good reference to understand about them is the *Stanford course on CNNs*, from which some parts are based on.

### 3.3.1 Architecture of a CNN

CNN have three different types of layers: convolutional layer, pooling layer and fully-connected layer. The simplest architecture of a CNN would be: an input (raw pixel values of an image, 3D-dimensional input), a convolutional layer, a ReLU (non-linear function), a pooling layer and the output fully-connected layer.

The **convolutional layer** is the main layer, whose parameters consist on a set of trainable filters (which are small along width and height compared to the input, but have the same depth as the input). In these layers, the weights correspond to the values of the filters and each output is obtained by sliding each filter along the input at any position, so that each filter gives a 2D-dimensional output given by the dot product and sliding of the filter. In other words, as we slide over the weight and height of the input, a 2-dimensional activation map is formed giving the response of that filter at every spatial position. The number of filters in the convolutional layer determines the depth of the output. The convolution operation is depicted on equation 3.14 and on figure 3.5.

As it is impractical to connect all neurons to every single pixel in the input images, we have a *local connectivity*, connecting each neuron to only a local region of the input volume. The extend of this region is the receptive field of the neuron, which is equal to the size of the filter.

Given an input $I$ of shape $[H, W, S]$ and a set of convolutional filters $K$ of shape $[N, N, S, T]$, which corresponds to $T$ filters of size $NxN$, the result $y$ of a simple convolution (with stride 1 and without padding) would be:

$$y_{h',w',t} = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{s=1}^{S} K_{i,j,s,t} I_{h,w,s} \tag{3.14}$$

Where $y_{h',w',t}$ is the value of $y$ at the position $(h', w', t)$.

The output size $[H', W', T]$ of a convolutional layer (CONV) is controlled by 3 parameters: depth, stride and zero-padding. The depth is determined by the number of filters in the layer, each learning to look at something different in the input. The stride is how we move or slide the filter along the pixels, so if the stride is 1, we move the filters one pixel at a time, and if the stride is $n > 1$, the filter jumps n pixels at a time. Therefore, the higher the stride, the smaller the output volume. Sometimes, it is convenient to pad the input with zeros around the border and the number of zeros or size of the zero-padding also affects the size of the output. We can take advantage of the zero-padding in order to obtain the same weight and height of the input at the output, for example. In total, the spatial size of the output, given an input volume size $V$, receptive field or filter size $F$, stride applied $S$ and zero-padding used $P$, would be given by: $(V - F + 2P)/S + 1$. In general, by setting zero-padding to be $P = (F-1)/2$ when the stride is 1 we ensure to have the same size $V$ at the output.

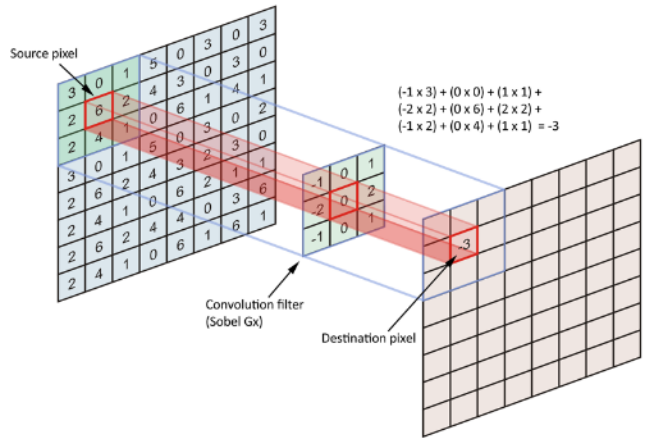


FIGURE 3.5: An example of a convolution filter (from *Accelerating Floyd-Steinberg on the Mali GPU* 2014). Each value of the output is created by multiplying its current location block and the coefficients of the filter.

The convolution operation can be performed as a matrix multiplication, because the operations between the filters and the input are dot products of the filters on local regions of the input. Having the same input $I$ and filters $K$ as in 3.14, we take $N \times N \times D$ sub-blocks of pixels from the input in each dot product with the filters. Then, we can extend each sub-block $N \times N \times D$ of the image into a column vector and create a matrix with $N \times N \times D$ rows and as many columns as locations in which we apply the filters. If we call that matrix $M$, and we also extend into a vector of size $N \times N \times D$ the filters, the convolution of the image with one filter is simply a matrix times a vector. For example, if the input is 227x227x3 and we have $11 \times 11 \times 3$ filters with stride 4, we obtain $(227 - 11)/4 + 1 = 55$ locations (width and height) and the matrix $M$ would have size $[11 \cdot 11 \cdot 3,\ 55 \cdot 55] = [363,\ 3025]$.

Moreover, we can stretch all the weights of the CONV layer are also stretched into rows and we obtain a matrix of size $[T,\ N \times N \times D]$, each row containing the weights of a filter. In the example, if we have $T = 96$ filters, we obtain a $96 \times 363$ matrix. The result of the convolution would be the matrix multiplication (matrix of the input times matrix of the filters), giving an output [T, number of locations] ($96 \times 3025$ in the example). Afterwards, we need to reshape it back to [locations weight, locations height, T] ($55 \times 55 \times 96$ in the example). This implementation has the disadvantage that it may use a lot of memory, but the benefit is that the operation is as simple as a matrix multiplication and it is useful to understand the mathematical operation of the convolution.

As in other deep neural networks, we use the BP algorithm to train the weights or filter coefficients. In this case, the backward pass of the algorithm would consist on another convolution.

A **ReLU layer** is usually applied after the CONV layer, which consists on a set of neurons that apply the non-saturating activation function called Rectified Linear Unit (ReLU). The mathematic expression of the function is $f(x) = max(0, x)$. This layer increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer. Other

functions can be used to increase the nonlinearity, such as the hyperbolic tangent $f(x) = \tanh(x)$ or the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$. Compared to other functions the usage of ReLU is preferable, because training the neural network results several times faster (Krizhevsky, Sutskever, and Hinton, 2012), without making a significant difference to generalization accuracy.

A **pooling layer** (POOL) is commonly introduced between CONV layers, so that the spatial size is progressively reduced in order to reduce the number of parameters and computations in the network. The way to do it is using the $MAX$ *operation* or selection of the maximum value within a window, independent in the depth axis (so, for each depth slice we select the maximum in that slice, not taking into account the other depth slices). We first select the pooling size or window size ($P_h \times P_w$) and we apply the $MAX$ operation every $P_h \times P_w$ block, substituting the $P_h \times P_w$ block by a single value corresponding to the maximum value in the block. In this case, we also have a stride parameter $S$ apart from the spatial extend $P$. From an input volume $W \times H \times D$, we obtain an output volume size of $W_o = (W - P)/S + 1, H_o = (H - P)/S + 1, D_o = D$.
In a pooling layer, we do not introduce new parameters nor zero-padding, it is a fixed function of the input that reduces its size. Meanwhile, not all the convolutional networks found in the past work use pooling layers and use only CONV layers (example: as in Springenberg et al., 2014) and, in those cases, we could use large stride in some layers to reduce the size. In the pooling layer, the back-propagation step would be like routing the gradient to the input that had the highest value in the forward pass, so the index of the position of the maximum value is useful in the BP.

**Normalization layers** (NORM) are sometimes applied in order to keep the values of the activations between 0 and 1, but their contribution is minimal in convolutional networks. The main idea is to maintains the mean activation close to 0 and the activation standard deviation close to 1. The normalization was inspired by computational neuroscience, which can be checked at DiCarlo, Pinto, and Cox, 2008.

The **fully-connected layer** (FC) at the end of the architecture has connections to all the activations in the previous layer. It differs from a CONV layer because the CONV layers are connected only to a local region in the input and many neurons in the CONV layer share parameters; but both layers compute dot products. Therefore, we can convert between FC and CONV layers. The FC layer (or layers) does the high-level reasoning in the neural network. Their activations can be computed with a matrix multiplication followed by a bias offset.

In these architectures, the weights and biases are also trained using a BP algorithm, as mentioned in the previous section, on a set of labeled data (in this case, images labeled pixel by pixel).

### 3.3.2 Visualization of the filters

The understanding of CNNs is not easy, so there exist different approaches trying to visualize what is going on and understand how the network trains. Some of them are: visualization of the activations and the filters, retrieving images that maximally activate a neuron, embedding the codes for example with t-Distributed Stochastic Neighbor Embedding (t-SNE) or including occlusions in different parts of the input

images. In the scope of this project, we will deal with the first approach.

As explained in the previous subsection, each filter focuses on a different section in the image or on a different structure. For example, one filter may focus on the vertical lines appearing in the image while another one would focus on the horizontal lines. To try to understand how the filters are trained and what they focus on, we can either visualize the activations in the forward pass or visualize the weights.

When picturing the activations with ReLU (3.6), we can see the difference between the different CONV layers in our architecture and the differences in each step of the training process. When training, the activations start out looking relatively vague and dense and, as the training keeps advancing, they usually become more sparse and localized to the point where we can see what gives an activation in each layer and for each filter. But, this way to visualize the network is difficult to understand and it may be confusing, as some activation maps may be all zero for many different inputs or with very little activations. If an all zero activation appears in a lot of inputs, it may indicate dead filters, which can be due to an abusive high learning rate.

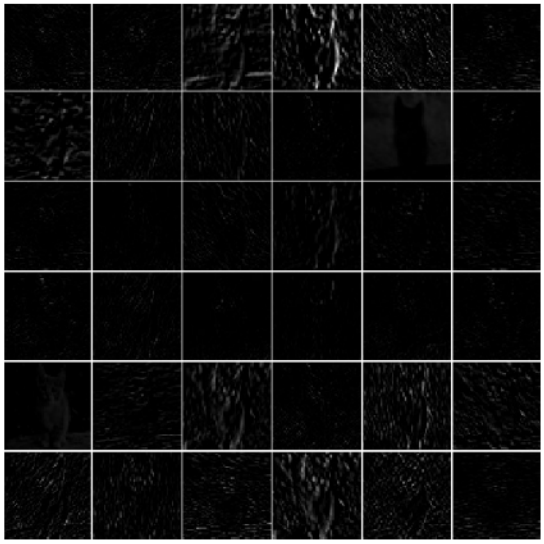

FIGURE 3.6: Activations of the first CONV layer (from *Visualizing what ConvNets learn*). As it can be appreciated, most of the activations are zero (black) and it is difficult to understand what is going on.

When visualizing the weights, we can look up a bit more of what is going on. Visualizing the filters is usually only interpretable on the first CONV layer, which is looking directly at the raw pixel data, but it can be possible at deeper layers in the network sometimes. The weights are useful to visualize because, if the network is well-trained, we are able to display nice and smooth filters without any noisy patterns. If we see any noisy patterns, it can be due to the lack of training of the network or the possibility of a very low regularization strength that may have led to over-fitting. In order to visualize the weights, we simply inject an input with a single 1 at the center, so that, after convolving it with a filter, what we visualize are the weights. As an example, we can see how the values of the filters or weights on

the first CONV layer have very defined patterns in figure 3.7.

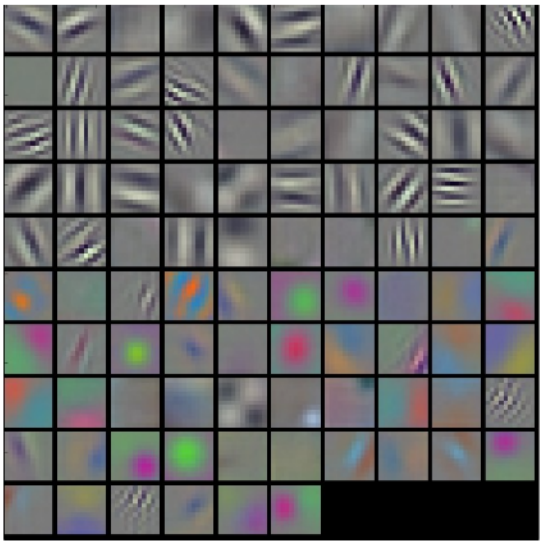

FIGURE 3.7: Typical-looking filters on the first CONV layer (from *Visualizing what ConvNets learn*). The weights are very nice and smooth, indicating nicely converged network.

### 3.3.3 Reduction of the parameters in a CNN

In CNNs, there are different methods that have been applied with the purpose of reducing the number of parameters and, as a consequence, the memory needed. Some of the basic methods, already introduced in chapter 2, will be explained in this section.

In order to reduce the number of parameters in a layer, we can share parameters between neurons. For example, if we have $5 \times 5 \times 64$ filters, we can share the $5 \times 5$ parameters in all the 64 depths. Constraining the neurons in each depth to use the same weights and bias is feasible because, a feature analyzed at a particular position (x, y, 0) of the input, should be also useful at different positions on the depth, that is every (x, y, z). The reason behind that is how inputs are prepared as a matrix of pixels for each dimension on the color space, which is usually the Red-Green-Blue (RGB) color model (in this case, the input of the network will have size $H \times W \times 3$, being $H$ and $W$ the height and width of the image, respectively). In practice, we would compute the gradient of the weights for every neuron in the volume in the BP algorithm, but the gradients would be added up across each depth slice and only update a single set of weights per slice. In the forward pass, we can compute the convolution in each depth slice as a convolution of the neuron's weights with the input volume.

In some situations, when there exist input images with specific structure, it may not make sense to share the parameters. For example, when the input images have a centered human face. In that case, it is not reasonable to search for eye-specific or nose-specific features in different spatial locations, for example. In those cases, we

share less parameters and we have a locally-connected layer.

The pooling layer, mentioned in the previous subsection, is another method to reduce the number of parameters. Although it does not reduce the parameters in the CONV layers, as it reduces the height and weight of their outputs through the network, we end up having less trainable parameters in the FC layers. This result is of great importance, as the FC layers are the ones with a higher number of parameters due to the fully-connectivity to the activations of the previous layer.

Another case in which our network would have less parameters is the application of $1 \times 1$ convolutions, instead of having a filter size $N \times N, N > 1$. As in CNNs we have a 3-dimensional input, with 3-dimensional dot products, so it makes sense to have $1 \times 1$ convolutions. These type of CONV layers where first investigated in Lin, Chen, and Yan, 2013.

On the other hand, we can introduce another hyperparameter to the CONV layer called dilatation. The dilatation is added in the CONV layers in order to have filters with spaces between each cell (dilatation of the filters). For example, we can have $3 \times 3$ filters with dilatation 1, so the block analyzed in the image will be of size $5 \times 5$. For example, in one dimension with filter $v$ of size 3 and input $x$, the first step of the convolution (first value in the output) would be $v[0]\,x[0] + v[1]\,x[1] + v[2]\,x[2]$ and, if we add dilatation of size 1, we would have $v[0]\,x[0] + v[1]\,x[2] + v[2]\,x[4]$ instead. As it can be interpreted as having $5 \times 5$ filters with zeros, we would say that the effective receptive field of the neurons is $5 \times 5$. Therefrom, if we use dilated convolutions, the effective receptive field grows much quicker. The example article mentioned in chapter 2 is Yu and Koltun, 2015. We can see the dilatation as a form to reduce the number of parameters by substituting a larger filter ($5 \times 5$ in the example) by a smaller filter ($3 \times 3$ in the example).

## 3.4 Tensor decomposition

A Tensor is a geometric object, which can be defined as a multidimensional array. In the same way a vector is an $n$-dimensional array of length $n$ with respect to a given vectorial basis (independent set of vectors with which we can represent the array), we can represent a tensor with respect to a basis with a multidimensional array.

A *nth*-rank tensor in a $m$-dimensional space is a mathematical object that has $n$ indices, $m^n$ components and obeys certain transformation rules similar to the rules for vectors (properties of the product, properties of the addition, identity elements, ...). Each index of a tensor ranges over the number of dimensions of the space. For example, a scalar would be a tensor of $rank$ 0 and a vector is a tensor of $rank$ 1.

Figure 3.8 shows an example of a 3-dimensional tensor $T$ of size $X \times Y \times Z$. Given a CONV layer of $K$ filters of size $N \times N \times D$, we can see the weights in that layer, $\mathbf{W}$, as a tensor of dimension $N \times N \times D \times K$.

In mathematics, there are several ways to express the tensors in terms of scalars, vectors, matrices and other tensors of different sizes, named decompositions of a

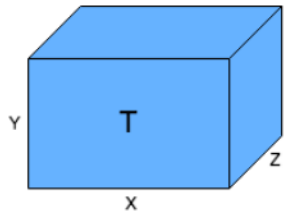


FIGURE 3.8: 3-dimensional Tensor.

tensor. There exist a lot of different decompositions: INDSCAL, PARAFAC, CANDELINC, DEDICOM, PARATUCK, TUCKER, ... We can find a discussion of some decompositions and their applications at Kolda and Bader, 2009. For example, Tucker decomposition, which is the one applied in this project, decomposes a tensor into a set of matrices and a central smaller tensor.

### 3.4.1 Singular Value Decomposition (SVD)

First, we will explain about Singular Value Decomposition (SVD), a factorization of a matrix, so that we can understand better how Tucker decomposition works. Given a $m \times n$-matrix, $\mathbf{M}$, whose elements come from the field $A$, the SVD decomposition is given by:

$$\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{3.15}$$

Where $\mathbf{U}$ is a $m \times m$ unitary matrix and $\mathbf{V}$ is a $n \times n$ unitary matrix, in the case $A = R$ we would have orthogonal matrices. $\mathbf{V}^*$ is the conjugate transpose of $\mathbf{V}$. $\mathbf{\Sigma}$ is a diagonal matrix with non-negative real numbers, with a diagonal corresponding to the singular values of $\mathbf{M}$.

First of all, we will go through some definitions:

1. A *diagonal matrix* is a matrix whose values are zero except the ones in the diagonal. In the case where all the elements in the diagonal are equal to 1, it is called *identity matrix*.

2. Given a matrix $\mathbf{A}$, its *transpose* is an operator which flips a matrix over its diagonal, that is it switches the row and column indices of the matrix by producing another matrix $\mathbf{A}^T$ (reflect $\mathbf{A}$ over its main diagonal, the rows of $\mathbf{A}$ become the columns of $\mathbf{A}^T$ and the columns of $\mathbf{A}$ become the rows of $\mathbf{A}^T$).

3. Given a matrix $\mathbf{A}$ from the complex space, its *complex conjugate*, $\overline{\mathbf{A}}$, is the conversion of each number to another value with equal real part and imaginary part equal in magnitude but opposite in sign.

4. The *conjugate transpose* of a $m \times n$-matrix $\mathbf{A}$ with complex values is denoted as $M^*$ and it is obtained from $\mathbf{A}$ by taking the transpose and then taking the complex conjugate of each entry (i.e., negating their imaginary parts but not their real parts). The mathematical expression of this definition is $\mathbf{A}^* = (\overline{\mathbf{A}})^T = \overline{\mathbf{A}^T}$, with $\mathbf{A}^T$ being the transpose and $\overline{\mathbf{A}}$ the complex conjugate.

5. A quadratic matrix $\mathbf{A}$ is *unitary* if $\mathbf{A}\mathbf{A}^* = \mathbf{A}^*\mathbf{A} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. If $\mathbf{A}$ is from the real space, we say that the matrix is *orthogonal* and it accomplishes $\mathbf{A}\mathbf{A}^T = \mathbf{A}^T\mathbf{A} = \mathbf{I}$. In those two cases, we have that the rows or

columns of the matrices are unitary vectors or orthogonal vectors, respectively, between each other.

6. A non-negative real number $\lambda$ is a *singular value* for $\mathbf{M}$ of a space $K^{m\times n}$ if and only if there exist unit-length vectors $\vec{u}$ in $K^m$ and $\vec{v}$ in $K^n$ such that $\mathbf{M}\vec{v} = \lambda\vec{u}$ and $\mathbf{M}^*\vec{u} = \lambda\vec{v}$. The vectors $\vec{u}$ and $\vec{v}$ are called left-singular and right-singular vectors for $\lambda$, respectively.

Going back to the definition of the SVD, 3.15, we have that the values in the diagonal of $\mathbf{\Sigma}$ are the *eigen-values* or *singular values* $\lambda_i$, $i = 1, ..., k$ in $\mathbf{\Sigma}$ obtained by two steps in an iterative manner (first, the matrix is reduced to a bidiagonal matrix and, after, the SVD of the bidiagonal matrix is computed using QR algorithm, for example). The first $k = min(m, n)$ columns of $\mathbf{U}$ and $\mathbf{V}$ are, respectively, left-singular vectors and right-singular vectors for the corresponding singular values. Therefore, the SVD theorem implies that:

- An $m \times n$ matrix $\mathbf{M}$ has at most $k = min(m, n)$ distinct singular values.
- It is always possible to find a unitary basis $\mathbf{U}$ for $K^m$ with a subset of basis vectors spanning the left-singular vectors of each singular value of $\mathbf{M}$.
- It is always possible to find a unitary basis $\mathbf{V}$ for $K^n$ with a subset of basis vectors spanning the right-singular vectors of each singular value of $\mathbf{M}$.

The SVD is used to solve homogeneous linear equations, to find the pseudo-inverse of the matrix and to obtain the range and null space of the matrix. Therefore, it is commonly used in different fields of research, such as machine learning and image processing. For example, the SVD can be used to find the decomposition of an image processing filter into separable horizontal and vertical filters.

### 3.4.2 Tensor Mathematical Operations

Given a tensor $\mathbf{X}$ of $K^{I\times J\times K}$, a matrix $\mathbf{B}$ of $K^{M\times J}$ and a vector $\mathbf{a}$ of $K^I$. Using the notation $x_{i,j,k}$ for the element in the position $(i, j, k)$ in $\mathbf{X}$, we define the following operations:

- Tensor times matrix at axis $i$ is given by $\mathbf{X} \times_i \mathbf{B}$, the following equations 3.16 and 3.17 and the figure 3.9 give an example for the case $i = 1$ (first axis).

$$\mathbf{Y} = \mathbf{X} \times_1 B,\ \mathbf{Y} \in K^{M\times J\times K} \tag{3.16}$$

So each element $(m, j, k)$ of $\mathbf{y}$ is obtained by:

$$\mathbf{y}_{m,j,k} = \sum_i x_{i,j,k} \cdot b_{m,j} \tag{3.17}$$

We can represent the operation with $\mathbf{Y}_{(i)} = \mathbf{B} \cdot \mathbf{X}_{(i)}$, where $\mathbf{M}_{(i)}$ of a tensor $\mathbf{M}$ represents the mode-$i$ unfolding of the tensor. The mode-$i$ unfolding of a tensor is the unfolding of the tensor into a matrix so that the mode-$i$ fibers are rearranged to be the columns of a matrix. In figure 3.10, we can see how a fiber and a slice is defined in a 3-dimensional tensor.

- Tensor times vector operation in the $i$ axis is represented by $\overline{\times}_i$. Equations 3.18 and 3.19 and figure 3.11 show the example of the operation in the first axis ($i = 1$).

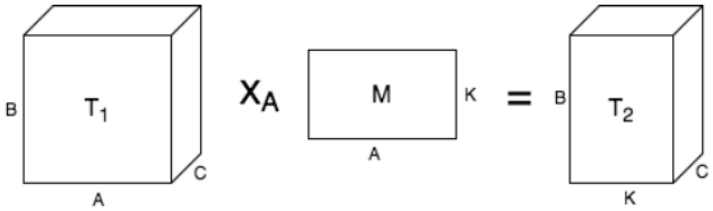


FIGURE 3.9: 3-dimensional example of the tensor times matrix operation.

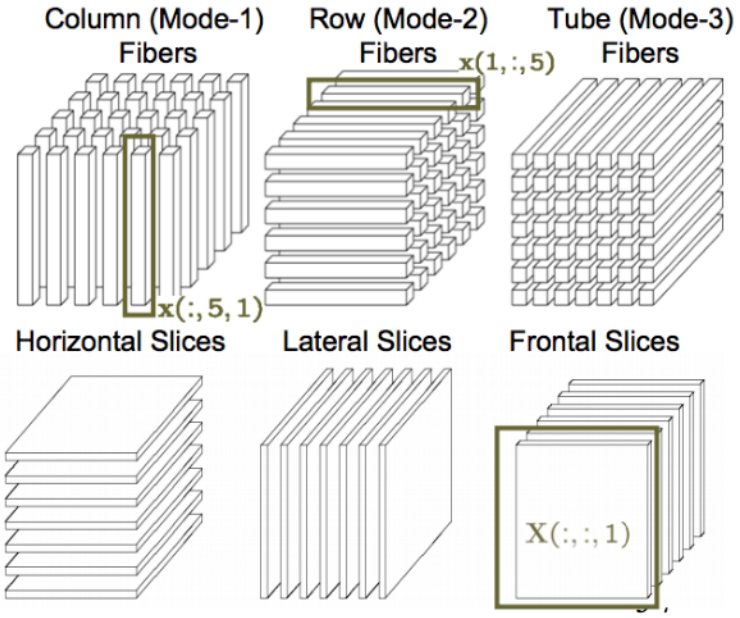


FIGURE 3.10: Fibers and slices of a 3-dimensional tensor (from *Mining Large Time-evolving Data Using Matrix and Tensor Tools. SIGMOD 2007 tutorial* 2007).

$$\mathbf{Y} = \mathbf{X} \overline{\times}_1 \mathbf{a},\ \mathbf{Y} \in K^{J \times K} \tag{3.18}$$

$$\mathbf{y}_{j,k} = \sum_i x_{i,j,k} \cdot a_i \tag{3.19}$$

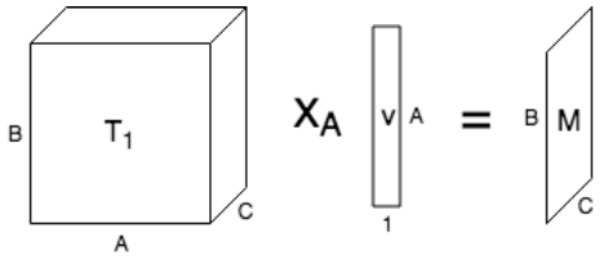


FIGURE 3.11: 3-dimensional example of the tensor times vector operation.

- 3-way outer product of three vectors, $\mathbf{a}$ of $K^I$, $\mathbf{b}$ of $K^J$, $\mathbf{c}$ of $K^K$, is defined as:

$$\mathbf{Y} = \mathbf{a} \circ \mathbf{b} \circ \mathbf{c},\ \mathbf{Y} \in K^{I \times J \times K} \tag{3.20}$$

$$\mathbf{y}_{i,j,k} = a_i \cdot b_j \cdot c_k \tag{3.21}$$

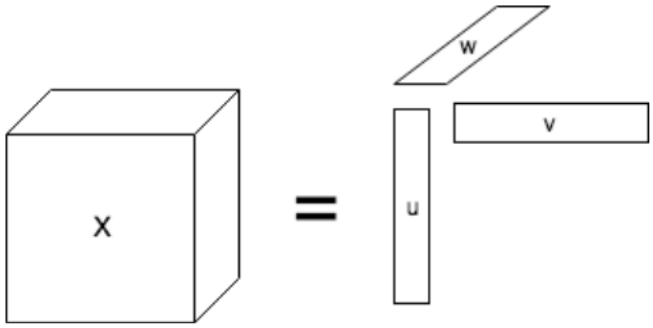


FIGURE 3.12: Example of the 3-way outer product.

- Matrix Kronecker product of two matrices $\mathbf{A} \in K^{M \times N}$ and $\mathbf{B} \in K^{P \times Q}$ is defined as:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{1,1}B & a_{1,2}B & \cdots & a_{1,N}B \\ a_{2,1}B & a_{2,2}B & \cdots & a_{2,N}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{M,1}B & a_{M,2}B & \cdots & a_{M,N}B \end{bmatrix} = [a_1 \otimes b_1,\ a_2 \otimes b_2,\ \cdots,\ a_N \otimes b_Q] \in K^{MP \times NQ} \tag{3.22}$$

- Matrix Khatri-Rao product, given a matrix $\mathbf{A} \in K^{M \times R}$ and a matrix $\mathbf{B} \in K^{N \times R}$ is defined as:

$$\mathbf{A} \odot \mathbf{B} = [a_1 \otimes b_1, a_2 \otimes b_2, \cdots, a_N \otimes b_Q] \in K^{MN \times R} \tag{3.23}$$

Comparing the equation 3.23 with the Kronecker Matrix 3.22, we can appreciate that, given a vector $\mathbf{a}$ and a vector $\mathbf{b}$, both products $\mathbf{a} \otimes \mathbf{b}$ and $\mathbf{a} \odot \mathbf{b}$ have the same elements, but one is shaped into a matrix and the other into a vector, respectively.

### 3.4.3 Kruskal Tensor

The Kruskal tensor (Coppi and Bolasco, 1989) is a way to decompose a tensor into an addition of 3-way outer products of vectors, as pictured in 3.13. Given a tensor $\mathbf{X}$, some vectors $\mathbf{u}_i$, $\mathbf{v}_i$, $\mathbf{w}_i$ and scalars $\mu_i$ with $i = 1, ..., R$, we can obtain a Kruskal tensor with $R$ vectors as the one in figure 3.13.

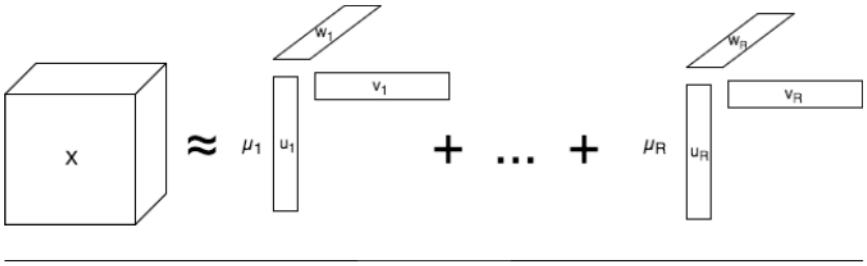


FIGURE 3.13: 3-dimensional Kruskal Tensor.

The mathematical expression of the decomposition is:

$$\mathbf{X} = \sum_r \mu_r \, \mathbf{u}_r \circ \mathbf{v}_r \circ \mathbf{w}_r \tag{3.24}$$

If we put all the vectors in matrices $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ and the scalars in a diagonal matrix $\mathbf{\Lambda}$, we can represent the Kruskal tensor in a matrix form with any of the following equations:

$$\mathbf{X}_{(1)} = \mathbf{U}\mathbf{\Lambda}(\mathbf{W} \odot \mathbf{V})^T \tag{3.25}$$

$$\mathbf{X}_{(2)} = \mathbf{V}\mathbf{\Lambda}(\mathbf{W} \odot \mathbf{U})^T \tag{3.26}$$

$$\mathbf{X}_{(1)} = \mathbf{W}\mathbf{\Lambda}(\mathbf{V} \odot \mathbf{U})^T \tag{3.27}$$

And, if we unfold the tensor as a vector $vec(\mathbf{X})$ and the scalars $\mu_r$ as a vector $\mu$ we can represent the Kruskal tensor as:

$$vec(\mathbf{X}) = (\mathbf{W} \odot \mathbf{V} \odot \mathbf{U})\mu \tag{3.28}$$

### 3.4.4 Tucker Tensor

The Tucker tensor is a bit different from the Kruskal tensor in the sense that, instead of having an addition of 3-way outer products of vectors, we have more than one sum of those products, which can be represented with matrix times tensor products as in figure 3.14.

The mathematical expression of the Tucker tensor of a tensor $\mathbf{X} \in \mathbb{R}^{I \times J \times K}$ is:

$$\mathbf{X} = \sum_r \sum_s \sum_t g_{r,s,t}\, \mathbf{u}_r \circ \mathbf{v}_s \circ \mathbf{w}_t = \mathbf{G} \times_1 \mathbf{U} \times_2 \mathbf{V} \times_3 \mathbf{W} \tag{3.29}$$

Where $\mathbf{U}$ is a $I \times R$-matrix, $\mathbf{V}$ is a $J \times S$-matrix, $\mathbf{W}$ is a $K \times T$-matrix and $\mathbf{G}$ is a $R \times S \times T$-tensor, with $R < I$, $S < J$ and $T < K$.

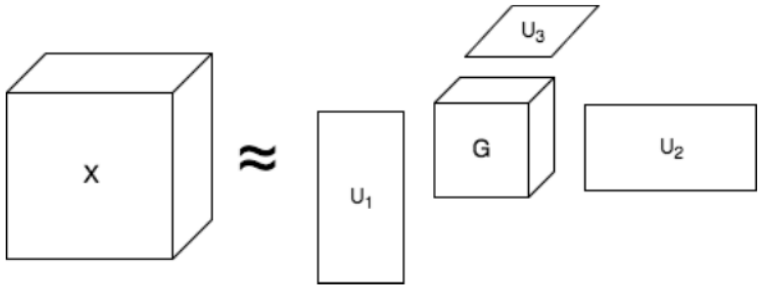


FIGURE 3.14: 3-dimensional Tucker Tensor.

As in the Kruskal tensor, we can obtain a matrix representation of the Tucker tensor, as in the following equations:

$$\mathbf{X}_{(1)} = \mathbf{U}\mathbf{G}_{(1)}(\mathbf{W} \otimes \mathbf{V})^T \tag{3.30}$$

$$\mathbf{X}_{(2)} = \mathbf{V}\mathbf{G}_{(2)}(\mathbf{W} \otimes \mathbf{U})^T \tag{3.31}$$

$$\mathbf{X}_{(1)} = \mathbf{W}\mathbf{G}_{(2)}(\mathbf{V} \otimes \mathbf{U})^T \tag{3.32}$$

And unfolding it into a vector:

$$vec(\mathbf{X}) = (\mathbf{W} \otimes \mathbf{V} \otimes \mathbf{U})vec(\mathbf{G}) \tag{3.33}$$

Notice that the difference between the Kruskal and the Tucker tensor is the type of product we apply ($\odot$ and $\otimes$), apart from having a tensor $\mathbf{G}$ in the Tucker tensor rather than the $\mathbf{\Sigma}$ matrix in the Kruskal tensor.

We can see the Tucker tensor as similarity with a SVD in each dimension, being $\mathbf{G}$ the singular values matrix and $\mathbf{U}, \mathbf{V}, \mathbf{W}$ the matrices with the singular vectors of each dimension. From this and knowing that some filters in the same CONV layer have dependent or related parameters (because some of them analyze the same or similar features), we can intuit that this tensor decomposition is feasible in the convolutional filters.

## 3.5 Summary

In this chapter, we first review the basics on Neural Networks, from the artificial neuron and the basic architecture in 3.1 to the BP algorithm in 3.2. We explain how two layers of neurons limits a lot the applications, thus, we require a hidden layer. With the addition of several hidden layers we have a Deep Neural Network. When disclosing the BP algorithm, we see how to train or set the different weights of the networks to handle different functions.

Afterwards, we review some basics on CNNs in 3.3, seeing the different layers and how they work plus the convolution formulation. We then figure out how to

visualize the filters of weights of the CNNs and why, plus some methods that have been used to reduce the number of parameters (sharing values between neurons, pooling the outputs of each or some CONV layer, applying $1 \times 1-$convolutions and dilatation of the filters).

Finally, we go through all the basic mathematics useful to understand the tucker decomposition (definition of a tensor, SVD and tensor operations), which is the basis of this project, and analyze another decomposition (the Kruskal decomposition) that is related to the Tucker decomposition.

# Chapter 4

# Tucker decomposition in CNN

## 4.1 Application of Tucker decomposition in CONV layers

As mentioned in the previous chapter (3), the CONV layers are formed by a set of weights corresponding to a set of $I \times J \times S$-filters. Given an input $\mathbf{X}$ of size $H \times W \times S$ and a CONV layer of $T$ filters of size $I \times J \times S$, the weights of that layer can be represented in a tensor $\mathbf{K}$ of size $I \times J \times S \times T$, denominated kernel of the CONV layer, and the convolution between the input and the kernel $\mathbf{Y}$ (of size $H' \times W' \times T$) would be:

$$y_{h',w',t} = \sum_i \sum_j \sum_s \mathbf{K}_{i,j,s,t}\, x_{h(i),w(j),s} \tag{4.1}$$

Note that $x_{h(i),w(i),s}$ has positions as a function of $i$ and $j$, which is dependent on the stride and padding we use for the convolution. We want to apply the Tucker decomposition to the kernel tensor $\mathbf{K}$. This decomposition, by following the steps on the previous chapter, would be:

$$\mathbf{K} = \mathbf{G} \times_1 \mathbf{A}^1 \times_2 \mathbf{A}^2 \times_3 \mathbf{A}^3 \times_4 \mathbf{A}^4 \tag{4.2}$$

Being $\mathbf{G}$ the smaller $R_1 \times R_2 \times R_3 \times R_4$-kernel ($R_1 < I$, $R_2 < J$, $R_3 < S$ and $R_4 < T$) and $\mathbf{A}^1$, $\mathbf{A}^2$, $\mathbf{A}^3$, $\mathbf{A}^4$ the matrices in the decomposition with sizes $I \times R_1$, $J \times R_2$, $K \times R_3$ and $T \times R_4$, respectively. This convolution can also be represented as the sum in 4.3.

$$\mathbf{K}_{i,j,s,t} = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} \mathbf{G}_{r_1,r_2,r_3,r_4} \mathbf{A}^1_{i,r_1} \mathbf{A}^2_{j,r_2} \mathbf{A}^3_{s,r_3} \mathbf{A}^4_{t,r_4} \tag{4.3}$$

If we substitute 4.3 into 4.1 we obtain 4.4.

$$y_{h',w',t} = \sum_i \sum_j \sum_s \left( \sum_{r_1} \sum_{r_2} \sum_{r_3} \sum_{r_4} \mathbf{G}_{r_1,r_2,r_3,r_4} \mathbf{A}^1_{i,r_1} \mathbf{A}^2_{j,r_2} \mathbf{A}^3_{s,r_3} \mathbf{A}^4_{t,r_4} \right) x_{h(i),w(j),s} \tag{4.4}$$

$$\mathbf{K}_{i,j,s,t} = \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} \mathbf{G}_{i,j,r_3,r_4} \mathbf{A}^1_{i,r_1} \mathbf{A}^2_{j,r_2} \mathbf{A}^3_{s,r_3} \mathbf{A}^4_{t,r_4} \tag{4.5}$$

In the case of Tucker-2 decomposition (decomposition of the last two dimensions as in 4.5), the convolution can be separated in three different steps (as mentioned in Kim et al., 2015):

1. We first apply the matrix in the third dimension to the input. This will decrease the third dimension from $S$ to $R_3$.

$$z_{h,w,r_3} = \sum_s \mathbf{A}^3_{s,r_3} \, x_{h,w,s} \tag{4.6}$$

2. Then, we convolve the result with the kernel $\mathbf{G}$.

$$z'_{h',w',r_4} = \sum_i \sum_j \sum_{r_3} \mathbf{G}_{i,j,r_3,r_4} z_{h(i),w(j),r_3} \tag{4.7}$$

3. Finally, we re-size the last dimension from $R_4$ to $T$.

$$y_{h',w',t} = \sum_{r_4} \mathbf{A}^4_{t,r_4} \, z'_{h',w',r_4} \tag{4.8}$$

Equations 4.6 and 4.8 can be seen as $1 \times 1$ convolutions, thus, convolutions with kernel sizes $1 \times 1 \times S \times R_3$ and $1 \times 1 \times R_4 \times T$, respectively. And equation 4.7 is the common convolution that would take the same convolution hyperparameters as in the initial convolution 4.1 (same stride, padding, dilatation), because it is the step that affects the spatial locations of the input.

That is, we first apply a $1 \times 1$ convolution to reduce the depth from $S$ to $R_1$, we then apply a smaller kernel for the actual convolution and, finally, we apply a $1 \times 1$ convolution in order to resize the output back to the expected size $T$ (from $R_4$).

In Kim et al., 2015 they apply the tucker-2 compression to an already trained CNN. They first select the rank or values for $(R_3,\ R_4)$ applying the variational Bayesian matrix factorization (VBMF) (Nakajima et al., 2012), which is able to find noise variance, rank and provide a theoretical condition to recover the rank. They select the rank by applying the VBMF to the kernel tensor $K$. Once the rank is selected with this method, they apply the tucker decomposition to the trained network and fine-tune the resulting compressed network in order to recover as much as possible the accuracy.

Instead of applying the tucker decomposition on an already trained network, we select a compression rate (we select $R_3$ and $R_4$) and train the network. The compression rate for the tucker-2 decomposition is showed in equation 4.9.

$$C_r = \frac{IJST}{SR_3 + IJR_3R_4 + TR_4} \tag{4.9}$$

We can also obtain the speed-up ratio by adding the number of operations required in equations 4.6, 4.7 and 4.8. If we denote the size of the output $\mathbf{Y}$ as $H' \times W' \times T$, the speed-up ratio is:

$$S_r = \frac{IJSTH'W'}{SR_3HW + IJR_3R_4H'W' + TR_4H'W'} \tag{4.10}$$

We want to decompose the complete tucker decomposition (in all dimensions) into different steps as in 4.6, 4.7 and 4.8. This is not as easy as the tucker-2 decomposition, as it is related to the spatial dimensions, where the actual convolution is done (with strides, padding...). So it is important to analyze which are the different hyperparameters (stride, padding...) for each operation.

First of all, we analyze the case in which $R_1 = 1$, $R_2 = 1$, $R_3 = S$ and $R_4 = T$. In this case it is easily seen that the tucker decomposition is reduced to 3 steps: a 1-dimensional convolution with kernel $I \times 1 \times S \times S$, a 1-dimensional convolution with kernel $1 \times J \times S \times S$ and a $1 \times 1$-convolution with kernel $1 \times 1 \times S \times T$. In the first two convolutions the $S$ filters would share the $I, J$ weights and, also, the weights would be the same for all the depth slices $S$. The mathematical expressions for that are:

1. First 1-dimensional convolution with $\mathbf{A}^1$, with output size $H' \times W \times S$:

$$z_{h',w,s} = \sum_i \mathbf{A}^1_i \, x_{h(i),w,s} \tag{4.11}$$

   In this convolution, the hyperparameters of the original convolution (padding, strides...) for the height would be the hyperparameters in this convolution.

2. Second 1-dimensional convolution with $\mathbf{A}^2$, with output size $H' \times W' \times S$:

$$z'_{h',w',s} = \sum_j \mathbf{A}^2_j \, z_{h',w(j),s} \tag{4.12}$$

   In this convolution, the hyperparameters of the original convolution for the weight would be the hyperparameters for this convolution.

3. Last convolution with output size $H' \times W' \times T$:

$$y_{h',w',t} = \sum_s \mathbf{G}_{1,1,s,t} z'_{h',w',s} \tag{4.13}$$

   In this step, there would be no padding, no strides nor dilatation.

If we also apply the decomposition on the other two dimensions, we can apply the decomposition of the first dimension and reduce the number of operations ($S$ to $R_3$). The steps applying the decomposition on all the dimensions would be:

1. First, we reduce the depth applying a $1 \times 1-$convolution with $\mathbf{A}^3$ (output size $H \times W \times R_3$:

$$z^1_{h,w,r_3} = \sum_s \mathbf{A}^3_{s,r_3} \, x_{h,w,s} \tag{4.14}$$

2. We apply the 1-dimensional convolution with $\mathbf{A}^1$ as in 4.11, obtaining an output size $H' \times W \times T$:

$$z^2_{h',w,r_3} = \sum_i \mathbf{A}^1_i \, z^1_{h(i),w,r_3} \tag{4.15}$$

3. We apply the 1-dimensional convolution with $\mathbf{A}^2$ as in 4.12, obtaining an output size $H' \times W' \times T$:

$$z^3_{h',w',r_3} = \sum_j \mathbf{A}^2_j \, z^2_{h',w(j),r_3} \tag{4.16}$$

4. We apply the convolution with the smaller kernel obtaining an output size $H' \times W' \times R_4$:

$$z^4_{h',w',r_4} = \sum_{r_3} \mathbf{G}_{1,1,r_3,r_4}\, z^3_{h',w',r_3} \tag{4.17}$$

   In this step, there would be no padding, no strides nor dilatation as in 4.13.

5. We resize the output to $H' \times W' \times T$ with $\mathbf{A}^4$

$$y_{h',w',t} = \sum_{r_4} \mathbf{A}^4_{t,r_4}\, z^4_{h',w',r_4} \tag{4.18}$$

If we apply all the matrices as in the previous steps (4.14, 4.15, 4.16, 4.17, 4.18), the compression ratio and the speed-up ratio would be the ones in 4.19 and 4.20.

$$C_r = \frac{IJST}{SR_3 + I + J + R_3R_4 + TR_4} \tag{4.19}$$

$$S_r = \frac{IJSTH'W'}{SR_3HW + IH'W + JH'W' + R_3R_4H'W' + TR_4H'W'} \tag{4.20}$$

In general, 4.21 and 4.22 show the compression and speed-up ratios for the case $\{R_1,\, R_2,\, R_3,\, R_4\} > \{1,\, 1,\, 1,\, 1\}$.

$$C_r = \frac{IJST}{SR_3 + IR_1 + JR_2 + R_1R_2R_3R_4 + TR_4} \tag{4.21}$$

$$S_r = \frac{IJSTH'W'}{SR_3HW + IR_1H'W + JR_2H'W' + R_1R_2R_3R_4H'W' + TR_4H'W'} \tag{4.22}$$

If we analyze a single matrix involved in the spatial convolution, for example $\mathbf{A}^1$ with $I > R_1 > 1$, the convolution would be:

$$y_{h',w',t} = \sum_i \sum_j \sum_s \left( \sum_{r_1} \mathbf{G}_{r_1,j,s,t} \mathbf{A}^1_{i,r_1} \right) x_{h(i),w(j),s} \tag{4.23}$$

Which is equivalent to:

$$y_{h',w',t} = \sum_{r_1} \sum_j \sum_s \left( \sum_i \mathbf{A}^1_{i,r_1} x_{h(i),w(j),s} \right) \mathbf{G}_{r_1,j,s,t} \tag{4.24}$$

Then, we can first do the operation between brackets, which gives us $R_1$ results and store them for the following $R_1$ operations to obtain $\mathbf{Y}$. But, this increases the need for memory to operate. On the other hand, if we first apply the matrix $\mathbf{A}^1$ to the smaller kernel $\mathbf{G}$ and, then, apply the convolution with the resulting kernel, we decrease the number of results to be stored on memory when doing the operations but we end up with more operations. As the purpose of this project is to decrease the memory storage, we will use the second method for matrices $\mathbf{A}^1$ and $\mathbf{A}^2$. An interesting future work would be to find the best and more efficient algorithm to

apply the full convolution with the tucker decomposition.

The final steps for the convolution with the full tucker decomposition (with $\{R_1, R_2, R_3, R_4\} > \{1, 1, 1, 1\}$) would be:

1. Apply the matrix $\mathbf{A}^3$ to the input $\mathbf{X}$ as in 4.14.
2. Compute a kernel $\mathbf{G}^2$ from the small kernel $\mathbf{G}$ and matrices $\mathbf{A}^1$, $\mathbf{A}^2$ as $\mathbf{G}^2 = \mathbf{G} \times_1 \mathbf{A}^1 \times_2 \mathbf{A}^2$.
3. Convolve the output from the first step with the kernel $\mathbf{G}^2$ as in 4.7.
4. Apply the matrix $\mathbf{A}^4$ to the result from the previous convolution as a $1 \times 1-$convolution (4.18).

## 4.2 Experimental setup

In this section, we break down the dataset and the different studied architectures in order to find out the best way to implement the tucker decomposition in different models. We chose the CIFAR-10 dataset so that the experiments took less time to train and implement the models, training and test with TensorFlow (Abadi et al., 2015).

### 4.2.1 Dataset: CIFAR-10

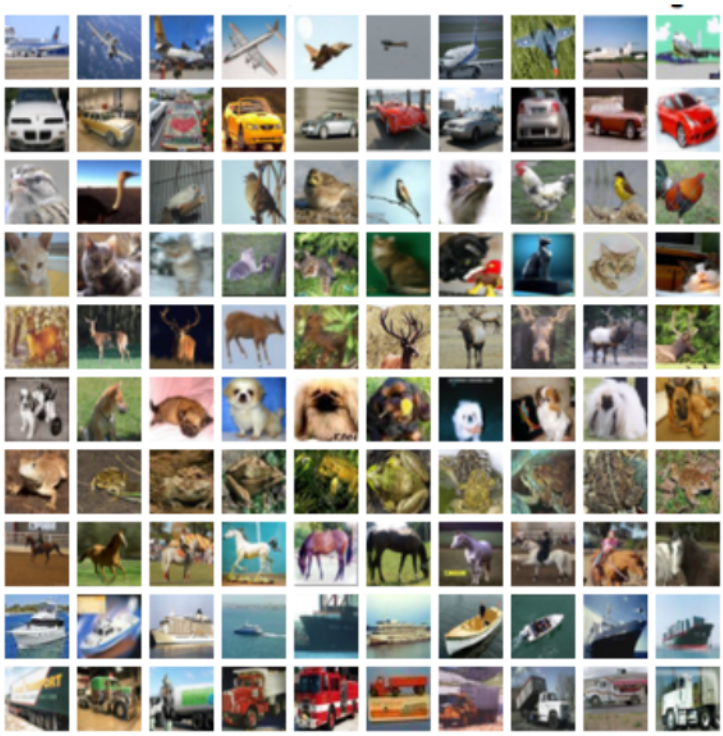

FIGURE 4.1: Example of some images in *CIFAR-10* dataset (from *Cifar datasets* 2009).

Different architectures with different compressions will be studied on the *CIFAR-10* dataset for classification (dataset available at *Cifar datasets* 2009). The *CIFAR-10*

dataset consists of 60000 color images of size $32 \times 32$ pixels that can be separated into 10 different classes, with 6000 images per class. 50000 of the images will be used for the training process and 10000 will be used to test the network. This dataset is small compared to other datasets (such as *ImageNet*), but it allows us to try a lot of different architectures in less time.

The images will be divided into batches or packs for the training process. Then, in each iteration we will train the network with as many images as the ones inside the batch (batch size) for that iteration. The training batches contain the images in random order, so some training batches may contain more images from one class than another. The important values of loss and accuracy will be the ones obtained after all the 50000 images for the training set are injected in the BP algorithm, thus, every epoch. Those values are the ones that will be stored during the training process. Also, every epoch, we will test the network with the remaining 10000 images to see how its error and accuracy evolves as the network is being trained.

The 10 classes in the dataset are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. All those classes are completely mutually exclusive, not as in other datasets. Some examples of those classes can be seen at figure 4.1.

### 4.2.2 Architectures

Two architectures with different compressions (applying both tucker-2 and full tucker decompositions) will be studied in the following sections: *cifar quick* architecture and an adaptation of the famous *AlexNet* (Krizhevsky, Sutskever, and Hinton, 2012).

The first one is a quick way to obtain good results on *CIFAR-10* dataset based on the architecture found at Jia et al., 2014 on CAFFE. The architecture can reach the ~75% test accuracy after 5000 iterations of training, as explained in *Cifar quick implementation on Caffe* 2014. Table 4.1 summarizes the structure of this network.

| LAYER | CHARACTERISTICS |
|---|---|
| CONV layer | 32 filters $5 \times 5$, padding 2, stride 1 |
| ReLU layer | - |
| POOL layer | pool size $[3,\ 3]$, stride 2 |
| CONV layer | 32 filters $5 \times 5$, padding 2, stride 1 |
| ReLU layer | - |
| POOL layer | pool size $[3,\ 3]$, stride 2 |
| CONV layer | 64 filters $5 \times 5$, padding 2, stride 1 |
| ReLU layer | - |
| POOL layer | pool size $[3,\ 3]$, stride 2 |
| FC layer | 64 units |
| FC layer | 10 units |
| FC layer | Number of classes ($= 10$) units |

TABLE 4.1: Table of the Cifar quick model.

The second one is a more complex architecture, with more CONV layers and with larger filters ($11 \times 11$ and $5 \times 5$ filters). Table 4.2 summarizes the structure of this model. The NORM layers use local response normalization and the FC layers have smaller units than the ones in the original *AlexNet* model (4096), as *CIFAR-10* images are much smaller than *ImageNet* dataset images in which the model was originally applied ($32 \times 32$ compared to $224 \times 224$ images). In this case, there are no reliable records of how the architecture deals with the *CIFAR-10* dataset. Either way, in both cases, we will first train and test the non-compressed models so that we are able to compare them better using the same *hyperparameters* with and without compression.

| LAYER | CHARACTERISTICS |
|---|---|
| CONV layer | 96 filters $11 \times 11$, padding 0, stride 1 |
| ReLU layer | - |
| POOL layer | pool size $[3,\ 3]$, stride 2 |
| NORM layer | depth radius 5, bias 2, alpha 0.0001, beta 0.75 |
| CONV layer | 256 filters $5 \times 5$, padding 2, stride 1 |
| ReLU layer | - |
| NORM layer | depth radius 5, bias 2, alpha 0.0001, beta 0.75 |
| CONV layer | 384 filters $3 \times 3$, padding 1, stride 1 |
| ReLU layer | - |
| CONV layer | 384 filters $3 \times 3$, padding 1, stride 1 |
| ReLU layer | - |
| CONV layer | 256 filters $3 \times 3$, padding 1, stride 1 |
| ReLU layer | - |
| POOL layer | pool size $[3,\ 3]$, stride 2 |
| FC layer | 256 units |
| FC layer | 256 units |
| FC layer | Number of classes ($= 10$) units |

TABLE 4.2: Table of the *AlexNet* model adaptation.

## 4.3 Weights initialization

The initialization of the weights and the optimizer we use can affect the results and the convergence of the network, as they are the two main tools that are able to change how the BP algorithm starts and ends through the loss or error function.

An optimizer is a method to minimize the loss function, that is to perform the BP algorithm in 3. The differences between optimizers is how they apply the gradient to descent through the loss function. Some of them move more randomly around the function and others move following a more smooth descending and some of them may arrive quicker to the solution than others, for example. Some classic optimization algorithms would be the *Gradient Descent* and *Adagrad*.

The first tries where using the *Gradient Descent Optimizer* (explained in Hasdorff, 1976) with an exponential decay of the learning rate, that is a variable learning rate following exponential tendency. The chosen optimizer, in the end, was the *Root Mean Square Propagation (RMSProp) optimizer* with the default values of decay, momentum and epsilon in the TensorFlow function. This optimizer is an adaptative learning rate method proposed in *Geoff Hinton Coursera class, lecture 6e.* 2014.

In this section we will see how a bad the initialization of the weights can affect our training until the point that it gets stuck to a local minimum very far away from the actual minimum. In all the cases, the initialization of the biases was a simple constant initialization.

### 4.3.1 Using truncated normal initializer

First of all, we started with a truncated normal initialization for all the weights. The tensorflow function `tf.truncated _normal_initializer`, which has the mean and standard deviation as tunable parameters. The truncated normal initialization consists of initializing the variables so that they follow a normal distribution with specified mean and standard deviation, but dropping and re-picking the values whose magnitude is more than 2 standard deviations from the mean (called truncation of the values).

Applying this initialization to the first architecture, it worked for the without compression, as we could obtain an accuracy of 70% for train and ~77% for test in 100 epochs. But, when compressing just a bit the network, the results were always stagnant at a certain point, far away from the desired point. Even changing different *hyperparameters*, such as the learning rate or the standard deviation of the initialization, the algorithm kept reaching a dead point. Figure 4.2 and 4.3 show some examples of the cifar quick model compressing only the first CONV layer. The compression applied is from a $5 \times 5 \times 3 \times 32-$kernel to three kernels of sizes $1 \times 1 \times 3 \times 1$, $5 \times 5 \times 1 \times 1$ and $1 \times 1 \times 1 \times 32$, that is from $5 \cdot 5 \cdot 3 \cdot 32 = 2400$ weights to $3 + 5 \cdot 5 + 32 = 60$ weights in the first CONV layer.

In 4.2, we can see how the loss decreases just a bit until it gets stiffed around a certain value and in 4.3 we see how the accuracy, as a consequence, does not increase but keeps jumping between the same values.

### 4.3.2 Using Xavier initialization

Once trying a simple architecture with a small compression in one CONV layer with the *Xavier initialization* the results where the expected ones. In TensorFlow, it corresponds to the function `tf.contrib.layers.xavier_initializer`, with uniform distributed random initialization.

In Glorot and Bengio, 2010 they try to understand better why standard gradient descent from random initialization gives poorly results with deep neural networks and they propose a new initialization scheme, *Xavier initialization*, that brings substantially faster and better convergence. *Xavier initialization* makes sure the weights are "just right" (not too large neither too small), keeping them in a reasonable range of values through the layers. The method is to give the weights a variance so that, in both the forward-propagation and the BP, the information keeps flowing. Basically,

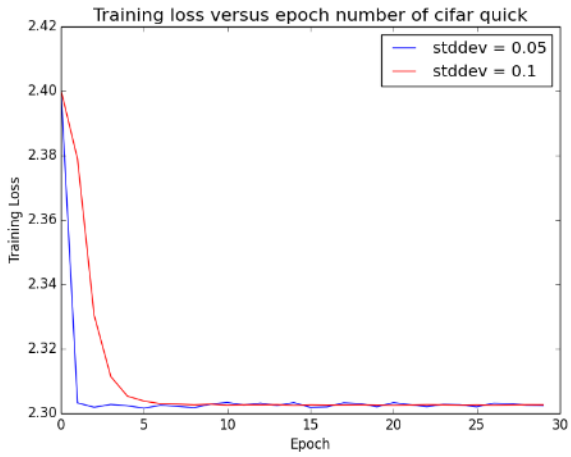


FIGURE 4.2: Training loss versus epochs of the cifar quick model with compression on the first CONV layer. Truncated normal initialization of the weights with different standard deviations: 0.1 in red and 0.05 in blue.

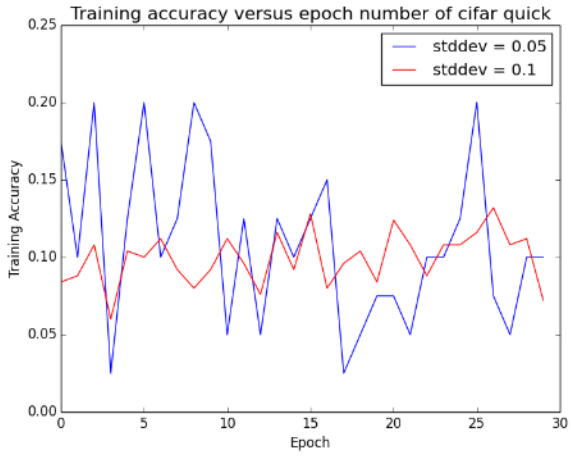


FIGURE 4.3: The same as in 4.2 with the training accuracy.

the *Xavier initialization* selects the adequate standard deviation in order to avoid the explosion or vanishing of the gradients in the BP algorithm.

In figures 4.4 and 4.5 we can see how, with Xavier initialization, we are able to decrease the loss and increase the accuracy of the compressed model as in the previous subsection, with the same tendency as with the model without compression.

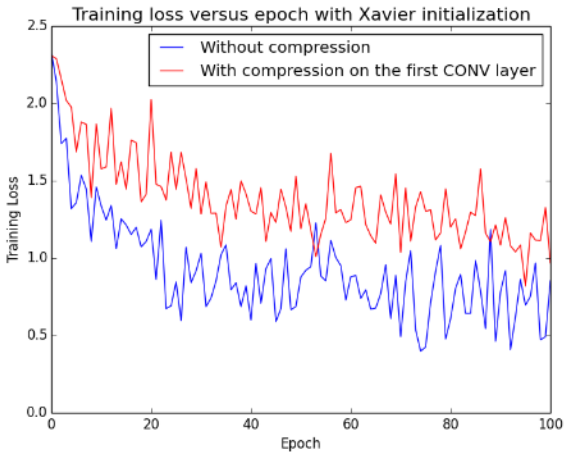


FIGURE 4.4: Training loss versus epochs of the cifar quick model with compression on the first CONV layer applying Xavier initialization. The blue curve corresponds to the model without compression and the red curve corresponds to the model with compression.

The final training loss and accuracy for the non-compressed model in 100 epochs are 0.853 and 70%, respectively, whereas the values for the compressed model are 0.969 and 65%.

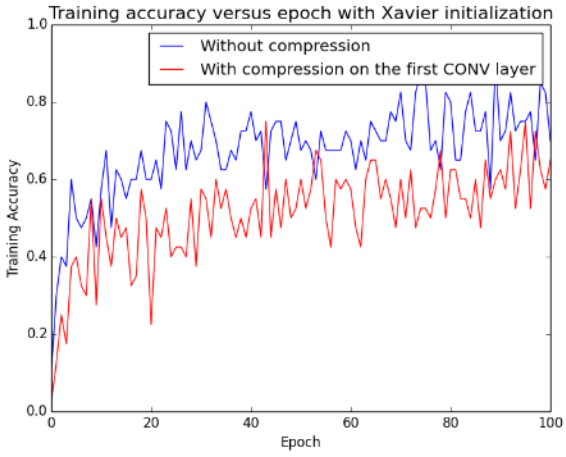


FIGURE 4.5: The same as in 4.4, with the training accuracy.

## 4.4 Summary

In this chapter we have seen how to apply the Tucker decomposition on the kernels of the convolutional layers in 4.1 by decomposing a single convolution into several smaller convolutions. We also analyze different cases of the Tucker decomposition to understand better its functioning. In 4.2 we introduce the dataset (CIFAR-10) and the two different architectures used in the experiments (*cifar quick* and *AlexNet*).

Before starting with the experiments, we first see how the initialization of the weights in compressed models affected their convergence in the BP algorithm (4.3) and we end up selecting Xavier initialization for our experiments.

# Chapter 5

# Tucker decomposition study

In the previous chapter (4) we have seen how we can apply the Tucker decomposition on the different kernels of the CONV layers and we have introduced the dataset and architectures we will study. In this chapter, we will implement the two architectures (*cifar quick* and *AlexNet*) with different decompositions, in order to analyze how much we can compress a network or a CONV layer. We will analyze the different models in terms of compression, not in terms of speed-up ratio.

## 5.1 Study of different architectures

In this section, we will analyze the different architectures, which differ on number of CONV layers, by applying different tucker decompositions and compressions. We will evaluate the different performances of each model in terms of their loss and accuracy on training and test and their compression ratios, with CIFAR-10 dataset.

We will compute both, the top-*1* and top-*5* accuracies, as, in some cases, the top-*5* accuracy is of great consideration (when the application of the network can have a bit of error, such that the correct option is contained between the first 5 results). This *top-n accuracy* is simply the result obtained from the times a predicted label matched the targeted label in the *n* predictions, divided by the number of data-points evaluated.

### 5.1.1 Cifar Quick Model

We will apply **tucker-2 compressions** in different CONV layers of the model in 4.1. First, we compare the model with tucker-2 and maximum compression in the first layer and the model with the same type of compression but in the last layer. We refer to tucker-2 maximum compression as the tucker-2 decomposition with both $R_3$ and $R_4$ equal to 1. Compressing the first CONV, which is the one applied directly to the input image, may give us worse results than compressing the last layer (applying compression after two layers without compression). See table 5.1 for the compression ratios, compression percentage of the layer ($C_{layer}$) and the total compression with respect to the non-compressed model (in %, $C_{model}$).

Table 5.2 shows how, when compressing only the second layer (which means compressing to more than a half of the initial model, as table 5.1 shows), the accuracy and loss is not far away from the first compressed model (which has only a 3% of compression). Therefore, it really does affect where we apply the compression. The first CONV layer is directly connected to the input image, so, if we compress already on that layer, the loss will be added in the end of the network. On the other hand, if we compress after two CONV layers have been convolved with the input, as in the second compressed model, we can compress even more to obtain near results.

| MODEL | LAYER | Decomposition | $C_r$ | $C_{layer}$ | $C_{model}$ |
|---|---|---|---|---|---|
| First compressed model | 1st CONV layer | kernel 5x5x1x1, plus two convolutions of 1x1x3x1 and 1x1x1x32 | 40 | 97.5% | 3% |
| Second compressed model | 3rd CONV layer | kernel 5x5x1x1, plus two convolutions of 1x1x32x1 and 1x1x1x64 | 423.14 | 99.77% | 53% |

TABLE 5.1: Table of the compression ratios and percentages with compression in a single CONV layer of the *cifar quick* model.

Therefore, it is better to compress the last layers, as they are more *abstract*.

| MODEL | $C_{model}$ | Training loss | Training accuracy | Training top-5 accuracy |
|---|---|---|---|---|
| Non-compressed model | 0% | 0.853 | 0.7 | 0.95 |
| First compressed model | 3% | 0.969 | 0.65 | 0.95 |
| Second compressed model | 53% | 1.214 | 0.525 | 0.875 |
| **MODEL** | $C_{model}$ | **Testing loss** | **Testing accuracy** | **Testing top-5 accuracy** |
| Non-compressed model | 0% | 1.128 | 0.825 | 0.975 |
| First compressed model | 3% | 2.233 | 0.675 | 0.875 |
| Second compressed model | 53% | 2.393 | 0.65 | 0.925 |

TABLE 5.2: Table of the loss, top-*1* and top-*5* accuracies for train and test without compression and with compression in a single CONV layer of the *cifar quick* model, after 100 epochs.

We consider now the models with compression in all the CONV layers of the network. The third compressed model compresses all the three CONV layers in *half* and the fourth model compresses as much as possible the three layers (with $R_3 = 1$ and $R_4 = 1$). To interpret the third compressed model, we define the compression in *half tucker-2* of a CONV layer with kernel size $D \times D \times S \times T$ as the tucker-2 decomposition with $\mathbf{A}^3$ matrix of size $S \times S/2$, $\mathbf{A}^4$ matrix of size $T \times T/2$ and smaller kernel $\mathbf{G}$ of size $D \times D \times S/2 \times T/2$ (not the same as compressing the number of weights in half). In the case of the first CONV layer, as $S = 3$, we reduce it to $R_3 = 1$ (integer part of $3/2$). We can see the compression rates and percentages of those two models in table 5.3.

Table 5.4 shows the loss and accuracies values, from which we see that if we abuse with the compression in such a small architecture (fourth model), even using Xavier initialization, the model either gets stuck at a local minimum or does not

| MODEL | LAYER | $C_r$ | $C_{layer}$ |
|---|---|---|---|
| Third compressed model | 1st CONV layer | 2.62 | 61.87% |
| | 2nd CONV layer | 3.45 | 71% |
| | 3rd CONV layer | 3.33 | 70% |
| | $C_{model}$ | **58%** | |
| Fourth compressed model | 1st CONV layer | 40 | 97.5% |
| | 2nd CONV layer | 287.64 | 99.65% |
| | 3rd CONV layer | 423.14 | 99.77% |
| | $C_{model}$ | **82%** | |

TABLE 5.3: Table of the compression ratios and percentages with compression in all the CONV layers of the *cifar quick* model.

| MODEL | $C_{model}$ | Training loss | Training accuracy | Training top-5 accuracy |
|---|---|---|---|---|
| Non-compressed model | 0% | 0.853 | 0.7 | 0.95 |
| Third compressed model | 58% | 1.175 | 0.675 | 0.975 |
| Fourth compressed model | 82% | 2.302 | 0.075 | 0.35 |
| **MODEL** | $C_{model}$ | **Testing loss** | **Testing accuracy** | **Testing top-5 accuracy** |
| Non-compressed model | 0% | 1.128 | 0.825 | 0.975 |
| Third compressed model | 58% | 1.95 | 0.7 | 1.0 |
| Fourth compressed model | 82% | 4.604 | 0.05 | 0.525 |

TABLE 5.4: Table of the loss, top-1 and top-5 accuracies for train and test without compression and with compression in all the CONV layers, after 100 epochs.

converge properly due to the lack of parameters. The third compressed model (with 0.675 of training accuracy and 0.7 of testing accuracy) gives even a better performance than the first (with 0.65 of training accuracy and 0.675 of testing accuracy) and the second (with 0.525 of training accuracy and 0.65 of testing accuracy) compressed models, with a higher compression of the network (58% with respect to 3% and 53%, respectively). The reason for that is because the compression is distributed among the different CONV layers and not centered in a single CONV. The third compressed model can be understood as a compression related to the redundancy between the filters in the same CONV layer (we *"share"* the parameters between the filers that are dependent on each other).

We can conclude that compressing each layer partially is better than completely compressing some layers is the best option. For different models, we could study the redundancy or the correlation of the filters in order to select the appropriate tucker decomposition for each layer of the network.

### 5.1.2 AlexNet model

As applying too high compression in such a small architecture as *cifar quick* leads to a bad performance when compressing it too much (as the fourth compressed model in 5.4), the higher order tucker decomposition will be applied on a larger architecture, an adaptation of the *AlexNet* architecture for *CIFAR-10* dataset (the one in table 4.2).

First of all, we train the full model to see which results achieves with *CIFAR-10*, as *AlexNet* is an architecture usually used with *ImageNet* dataset. After that, we verify the supposition in 5.1.1 about how compressing in different layers affects the outcome. Finally, we apply the tucker decomposition on all the CONV layers to see how much we can compress the network.

The first compressed model implements the full tucker-2 decomposition on the first CONV layer (as in the first compressed model of *cifar quick* architecture, 5.1), the second compressed model implements it to the last CONV layer (as in the second compressed model of *cifar quick* architecture, 5.1) and the third compressed model implements it to the three CONV layers in the middle (the 2nd, 3rd and 4th CONV layers). The compression ratios and percentages are in table 5.5 whereas the losses and accuracies are in table 5.8.

| MODEL | LAYER | $C_r$ | $C_{layer}$ | $C_{model}$ |
|---|---|---|---|---|
| First compressed model | 1st CONV layer | 158.4 | 99.37% | 1% |
| Second compressed model | 5th CONV layer | 1363.23 | 99.93% | 13% |
| Third compressed model | 2nd CONV layer | 1629.71 | 99.94% | |
| | 3rd CONV layer | 1330.43 | 99.93% | |
| | 4th CONV layer | 1673.52 | 99.94% | |
| | | | | 41% |

TABLE 5.5: Compression ratios and percentages of tucker-2 compression (with $R_3 = 1$ and $R_4 = 1$) at some CONV layer of the *AlexNet*.

| MODEL | LAYER | $\mathbf{C_r}$ | $\mathbf{C_{layer}}$ |
|---|---|---|---|
| Fourth compressed model | 1st CONV layer | 158.4 | 99.37% |
| | 2nd CONV layer | 1629.71 | 99.94% |
| | 3rd CONV layer | 1330.43 | 99.93% |
| | 4th CONV layer | 1673.52 | 99.94% |
| | 5th CONV layer | 1363.23 | 99.93% |
| | | $\mathbf{C_{model}} =$ | 54% |
| Fifth compressed model | 1st CONV layer | 157.68 | 99.65% |
| | 2nd CONV layer | 1330.43 | 99.93% |
| | 3rd CONV layer | 1673.52 | 99.94% |
| | 4th CONV layer | 1707.98 | 99.94% |
| | 5th CONV layer | 1363.23 | 99.92% |
| | | $\mathbf{C_{model}} =$ | 54% |
| Sixth compressed model | 1st CONV layer | 3.34 | 70.1% |
| | 2nd CONV layer | 3.22 | 68.9% |
| | 3rd CONV layer | 2.7 | 63% |
| | 4th CONV layer | 2.77 | 63.88% |
| | 5th CONV layer | 2.7 | 62.96% |
| | | $\mathbf{C_{model}} =$ | 34.29% |

TABLE 5.6: Table of the compression ratios and percentages with compression in all the CONV layers of the AlexNet model.

| LAYER | PARAMETERS | $\mathbf{C_r}$ | $\mathbf{C_{layer}}$ |
|---|---|---|---|
| 1st CONV layer | kernel of 1x11x1x48, $A^1$ of 11x1, $A^3$ of 3x1, $A^4$ 96x48 | 6.77 | 85.22% |
| 2nd to 5th CONV layer | as the sixth model | | |
| | | $\mathbf{C_{model}} =$ | 34.36% |

TABLE 5.7: Table of the seventh compressed model of the AlexNet model.

By compressing the maximum possible with tucker-2 decomposition, we have the fourth compressed model, which consists on tucker-2 decomposition with $R_3 = 1$ and $R_4 = 1$ in all the layers. And, by compressing as much as possible using the full tucker decomposition, we have the fifth compressed model, which consists on the full tucker decomposition with $(R_1, R_2, R_3, R_4) = (1, 1, 1, 1)$. Both lead to a compression of ∼54% with respect to the total network (note that the total compression is computed using also the FC layers, so the full tucker decomposition does not reduce much number of parameters with respect to the tucker-2, when applying both with maximum compression). The sixth compressed model handles the *half* tucker-2 decomposition mentioned in 5.1.1 to all the layers. Table 5.6 shows the different compression ratios of these 3 models. We also have the seventh compressed model, explained in table 5.7, which differs from the sixth model in the application of partial tucker decomposition in the first layer.

We analyze, as before, the losses and accuracies for training and test. We first compute the models with 6 epochs, to have a relative comparison between them and with the non-compressed model. The non-compressed model reached a 77.5% of accuracy in train and 70% of accuracy in test after those 6 epochs. We figure out in 5.8 which levels of loss and accuracy are reached by the other levels in the same number of epochs of training. Compared to the non-compressed model, the first compressed model obtains similar results to the non-compressed model for training (but 57.5% of testing accuracy), with only a compression of 1%. Meanwhile, the second compressed model, with compression on the last CONV layer and total compression of 13%, reaches the 65% of accuracy in train and 45% of accuracy in test. The third, fourth and fifth compressed models reach non-desirable poor results compared to the other models (with accuracies of ∼20% of less). The sixth model that applies less compression in the different CONV layers, but still reaches the 34% of compression, has even better outcomes than the first 1%-compressed model or the second 13%-compressed model. The seventh compressed model reaches acceptable values too, training accuracy of ∼60% and testing accuracy of 71%. With this, we verify once again our hypotheses in 5.1.1 of how important it is the way we introduce compression in the model (which CONV layers we choose and how much we compress each one), rather than the total compression of the model. We ask ourselves if those models are able to reach better outcomes and how many epochs are needed to reach the non-compressed model.

With the non-compressed model we can reach 0.97 of training accuracy and 0.83 of testing accuracy after 34 epochs. We train during more epochs the other models and find out that the 3rd, 4th and 5th compressed models continue immobile, with the same losses and accuracies. Then, they are stuck in a local minimum or the network is not able to support this level of compression. The sixth compressed model reaches the training accuracy of 0.96 and the test accuracy of 0.73 after 21 epochs. The seventh model reaches the 0.96 accuracy for train and the 0.7 accuracy for test after 15 epochs. Note that those two last models almost reach the non-compressed model with less epochs, as the number of parameters to train is smaller.

Therefore, even compressing the AlexNet network to 34% we are able to reach those satisfactory outcomes if we train during more iterations. Thus, the best compression to apply in other networks will depend on the application or purpose (trade-off performance/compression), but all have in common that we can partially compress each CONV layer (up to ∼80% per layer).

| MODEL | $C_{model}$ | Training loss | Training accuracy | Training top-5 accuracy |
|---|---|---|---|---|
| Non-compressed model | 0% | 0.575 | 0.775 | 1.0 |
| First compressed model | 1% | 0.979 | 0.725 | 1.0 |
| Second compressed model | 13% | 0.978 | 0.65 | 0.975 |
| Third compressed model | 41% | 2.294 | 0.15 | 0.45 |
| Fourth compressed model | 54% | 2.307 | 0.025 | 0.375 |
| Fifth compressed model | 54% | 2.307 | 0.05 | 0.45 |
| Sixth compressed model | 34.29% | 0.764 | 0.75 | 1.0 |
| Seventh compressed model | 34.36% | 1.096 | 0.59 | 0.99 |
| **MODEL** | $C_{model}$ | **Testing loss** | **Testing accuracy** | **Testing top-5 accuracy** |
| Non-compressed model | 0% | 1.016 | 0.7 | 0.95 |
| First compressed model | 1% | 2.428 | 0.575 | 0.95 |
| Second compressed model | 13% | 2.017 | 0.45 | 0.975 |
| Third compressed model | 41% | 4.612 | 0.1 | 0.625 |
| Fourth compressed model | 54% | 4.6109 | 0.1 | 0.475 |
| Fifth compressed model | 54% | 4.607 | 0.175 | 0.5 |
| Sixth compressed model | 34.29% | 1.776 | 0.675 | 0.95 |
| Seventh compressed model | 34.36% | 1.673 | 0.71 | 0.97 |

TABLE 5.8: Table of the loss, top-1 and top-5 accuracies for train and test for all the tested models of AlexNet, after 6 epochs.

## 5.2 Filters visualization

As explained in 3.3.2, we want to analyze the filters of the CONV layers in order to try to understand what is going on inside the network when we apply the compression. We will analyze the distribution of the filter values for different compressions.

We will study, as an example, the values of the weights in the first convolution of the *cifar quick* architecture with the compressions in 5.9, obtained from the models in 5.1.1. In order to see their behaviour, we picture the shape of the probability density functions (PDFs) of the filters for each slice in the input depth (Red-Green-Blue, RGB).

Figure 5.1 shows the shape of the PDFs of the first CONV layer weights in each depth slice (R, G and B slices) for the non-compressed, *half* tucker-2 compressed and full tucker-2 compressed *cifar quick* models. In the three different cases, the peak of the PDF tends to be taller (more zeros in the filters) and narrower (the cues around the peak tend to compress or decrease) as we apply more and more compression to the filters.

As we know from entropic theory, the narrower and taller a PDF is, the more we can add entropic compression to the values when using an encoding method to store them. According to Shannon's source coding theorem, the optimal code length for a symbol is $-\log_b P$, where $b$ is the number of symbols used to make output codes and $P$ is the probability of the input symbol, related to the PDF of the input symbol. Consequently, we can apply one type of entropy encoding methods, such as the Huffman encoding (as Han, Mao, and Dally, 2015 applies for compression) or arithmetic encoding, in order to reduce even more the space occupied by the weights (in terms of bits). Then, not only we are able to reduce the number of parameters, but also the number of bits needed to store them in memory.

| **MODEL** | **WEIGHTS SIZE** | $C_{layer}$ |
|---|---|---|
| Non-compressed model | [5x5x3x32]−kernel | 0% |
| *Half* tucker-2 compressed model | [3x1]$-A^3$, [5x5x1x16]−kernel and [32x16]$-A^4$ | ∼62% |
| Full tucker-2 compressed model | [3x1]$-A^3$, [5x5x1x1]−kernel and [32x1]$-A^4$ | 97.5% |

TABLE 5.9: Table of the different cifar quick compressions on the 1st CONV layer used in the weights visualization.

## 5.3 Summary

In this chapter we apply the tucker decomposition to obtain different models from the *cifar quick* and *AlexNet* models of the previous chapter. We see the importance of where we apply the compression: the results are better when compressing the last layers rather than the first layers and compressing a part of each layer gives us

better outcomes rather than compressing as much as possible a few layers or only one layer. The best option is to compress partially each layer, around ∼60 or ∼70% in our case, which gives us an important amount of compression (58% for *cifar quick* and 34% for *AlexNet*), and, in both cases, we can reach acceptable results.

Finally, we see how the filters of the compressed layers tend to have more zeros as we compress more and more (5.2). After picturing some PDFs of the weights, we arrive to a conclusion that we may not only be able to compress the number of parameters, but also the number of bits used to store those parameters if we use an entropy encoding technique.

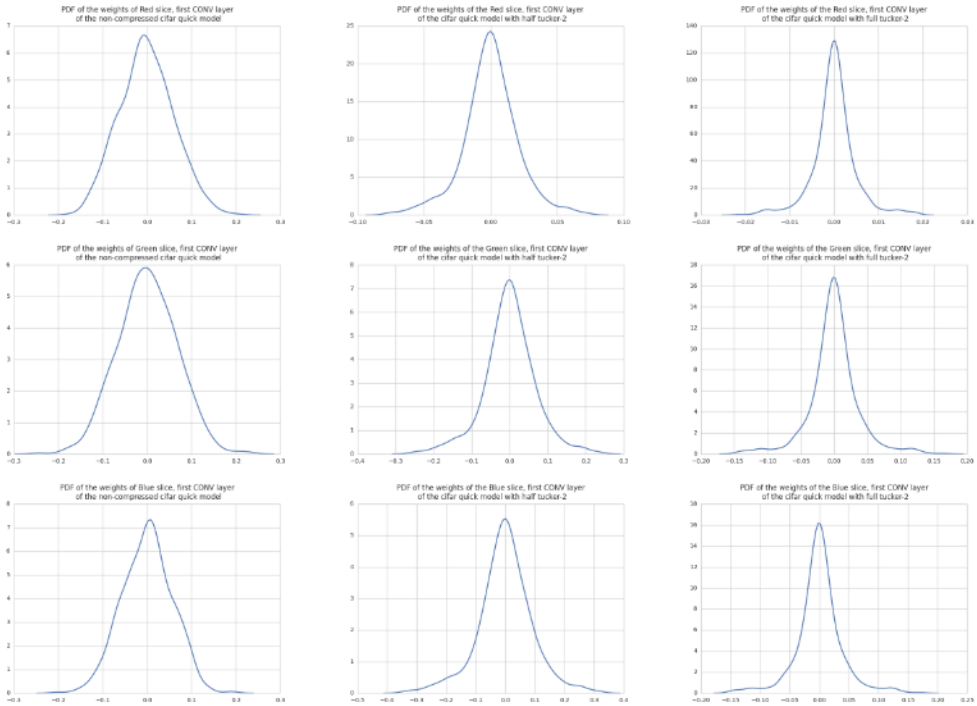


FIGURE 5.1: PDFs of the weights in the RGB slices of the first CONV layer. From left to right the PDFs are: the non-compressed model, *half* tucker-2 compression and the full tucker-2 compression.

# Chapter 6

# Conclusions and Future Work

The purpose of this chapter is to go through all the conclusions that can be derived from this project and to give some future lines to see the opened possibilities from this point on.

We have seen how we can decompose the kernel or weights of each CONV layer in a CNN by using the tucker decomposition and, as a repercussion, the reduction of the number of parameters and operations in each CONV layer. We have supposed different tucker decompositions (with fixed rank size, $R_1$, $R_2$, $R_3$, $R_4$) of the kernel in different CONV layers and trained the models with random initializations of the smaller kernel and matrices resultants from the decomposition. We have seen how the models perform when training and testing, by evaluating their loss, top-*1* and top-*5* accuracies. And we have visualized some of the weights to study the differences between the compressed and not compressed filters.

As a first conclusion, we have seen how the initialization affects the convergence of the BP algorithm when computing the gradients and trying to minimize the loss function. In this case, Xavier initialization improved the convergence of some models with compression. Some models did not converge, probably due to too much compression. But that is a supposition, as we only tried two different initializations (the truncated normal and Xavier initializations) and two different optimizers (the classical Gradient Descent Algorithm and the RMSProp). Therefore, a future work line would be to find the best initialization scheme and the best optimizer of the BP algorithm, when using high compression ratios, such that we can avoid bad training behaviors.

We have analyzed two different architectures, with different compressions in each architecture. From those studies, we can conclude that it is more important how and in which layers we apply the compression than the total compression of the network itself. For example, when compressing the first CONV layer, the outcome was worse than when compressing the last CONV layer. That is because the first CONV layer is the one directly connected to the input image whereas the CONV layers get more and more abstract as we go deeper in the model. Another example is that applying small compression in all the layers gives much better results than more compression in a few layers, and it may even have less parameters as a whole. That is because each CONV layer has a different purpose with different filters that focus on different features and, if we compress too much one of them, we can end up with a great loss added in the end. For those reasons, we can conclude that best way to perform compression is partially layer by layer. In our architectures, the best performance was achieved when applying the named *half*-tucker-2 decomposition

in all the layers.

Moreover, if we apply compression, we will need more training iterations to reach up to the same values, such as how we showed with the sixth compressed AlexNet model. But, in some cases, as, for example, if we compress too much the network, we may not be able to reach superior results even if we increase the number of training iterations (what happens in the fifth compressed AlexNet model).

If we compare tucker-2 decomposition and the whole tucker decomposition, we see that the importance of the whole decomposition is decreased when the spatial size of the filter is small. Even more than that, sometimes, we may get worse outcomes if we compress the spatial size, when the filter spatial size is smaller than $5 \times 5$, than if we compress a bit more the depth (it depends on the network and the database). Furthermore, the tucker-2 decomposition involves the compression of the (input, output)-pair in each CONV layer, which correspond to the higher values in the kernel shape.

In general, if we have a CNN we want to compress using tucker decomposition, we need to try different compressions that make sense with our database and network structure. In order to see which are the adequate compression ratios, we can, for example, analyze the correlations between the filters, and choose the ranks of the matrices in the tucker decomposition according to those correlations.

From the visualization of the filters PDFs in 5.2, we can suggest to apply an entropy encoding method to reduce the space in bits needed per value. Some entropy encoding methods include Huffman encoding, arithmetic encoding or static codes. Another future line would be to study which entropy encoding process can be applied after the model is trained and evaluate the added loss.

A different interesting point would be to exercise the tucker-2 decomposition on 1D-convolution-based architectures (such as Alvarez and Petersson, 2016) in order to reduce the pair (input, output)-sizes of each CONV layer.

Finally, we need to note that, in this project, we may not be applying the best algorithm (in terms of speed or number of operations) to apply the whole tucker decomposition (with $(R_1, R_2, R_3, R_4) > (1, 1, 1, 1)$) in the kernel of a CONV layer. Thus, searching different algorithmic methods to increase the speed of the computation when applying the decomposed convolution, would be another interesting work.

# Bibliography


Abadi, Martín et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: http://tensorflow.org/.

*Accelerating Floyd-Steinberg on the Mali GPU* (2014). https://community.arm.com/graphics/b/blog/posts/when-parallelism-gets-tricky-accelerating-floyd-steinberg-on-the-mali-gpu. [Online; accessed May-2017].

Alvarez, Jose and Lars Petersson (2016). "Decomposeme: Simplifying convnets for end-to-end learning". In: *arXiv preprint arXiv:1606.05426*.

Bishop, Christopher M (1995). *Neural networks for pattern recognition*. Oxford university press.

Carroll, J Douglas and Jih-Jie Chang (1970). "Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition". In: *Psychometrika* 35.3, pp. 283–319.

Chen, Wenlin et al. (2015). "Compressing neural networks with the hashing trick". In: *International Conference on Machine Learning*, pp. 2285–2294.

*Cifar datasets* (2009). https://www.cs.toronto.edu/~kriz/cifar.html.

*Cifar quick implementation on Caffe* (2014). http://caffe.berkeleyvision.org/gathered/examples/cifar10.html. [Online; accessed May-2017].

Coppi, R and S Bolasco (1989). "RANK, DECOMPOSITION, AND UNIQUENESS FOR 3-WAY AND iV-WAY ARRAYS". In:

Denil, Misha et al. (2013). "Predicting parameters in deep learning". In: *Advances in Neural Information Processing Systems*, pp. 2148–2156.

Denton, Emily L et al. (2014). "Exploiting linear structure within convolutional networks for efficient evaluation". In: *Advances in Neural Information Processing Systems*, pp. 1269–1277.

DiCarlo, James J, Nicolas Pinto, and David Daniel Cox (2008). "Why is Real-World Visual Object Recognition Hard?" In:

*Geoff Hinton Coursera class, lecture 6e.* (2014). http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. [Online; accessed May-2017].

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks." In: *Aistats*. Vol. 9, pp. 249–256.

Gong, Yunchao et al. (2014). "Compressing deep convolutional networks using vector quantization". In: *arXiv preprint arXiv:1412.6115*.

Han, Song, Huizi Mao, and William J Dally (2015). "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149*.

Hasdorff, Lawrence (1976). "Gradient optimization and nonlinear control". In:

Jaderberg, Max, Andrea Vedaldi, and Andrew Zisserman (2014). "Speeding up convolutional neural networks with low rank expansions". In: *arXiv preprint arXiv:1405.3866*.

Jia, Yangqing et al. (2014). "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093*.

Kim, Yong-Deok and Seungjin Choi (2007). "Nonnegative tucker decomposition". In: *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, pp. 1–8.

Kim, Yong-Deok et al. (2015). "Compression of deep convolutional neural networks for fast and low power mobile applications". In: *arXiv preprint arXiv:1511.06530*.

Kolda, Tamara G and Brett W Bader (2009). "Tensor decompositions and applications". In: *SIAM review* 51.3, pp. 455–500.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105.

Lebedev, Vadim et al. (2014). "Speeding-up convolutional neural networks using fine-tuned cp-decomposition". In: *arXiv preprint arXiv:1412.6553*.

LeCun, Yann et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

*LeNet tutorial*. http://deeplearning.net/tutorial/lenet.html. [Online; accessed May-2017].

Lin, Min, Qiang Chen, and Shuicheng Yan (2013). "Network in network". In: *arXiv preprint arXiv:1312.4400*.

*Mining Large Time-evolving Data Using Matrix and Tensor Tools. SIGMOD 2007 tutorial* (2007). http://www.cs.cmu.edu/~christos/TALKS/SIGMOD-07-tutorial/tensor3.pdf. [Online; accessed May-2017].

Nakajima, Shinichi et al. (2012). "Perfect dimensionality recovery by variational Bayesian PCA". In: *Advances in Neural Information Processing Systems*, pp. 971–979.

Puente, Santiago Pascual de la (2016). "Deep learning applied to speech synthesis". MA thesis. Universitat Politècnica de Catalunya.

Shashua, Amnon and Tamir Hazan (2005). "Non-negative tensor factorization with applications to statistics and computer vision". In: *Proceedings of the 22nd international conference on Machine learning*. ACM, pp. 792–799.

Springenberg, Jost Tobias et al. (2014). "Striving for simplicity: The all convolutional net". In: *arXiv preprint arXiv:1412.6806*.

*Standford course on CNNs*. http://cs231n.github.io/. [Online; accessed May-2017].

Tucker, Ledyard R (1966). "Some mathematical notes on three-mode factor analysis". In: *Psychometrika* 31.3, pp. 279–311.

*Tutorial on Multilayer Perceptron* (2010). http://deeplearning.net/tutorial/mlp.html. [Online; accessed May-2017].

*Visualizing what ConvNets learn*. http://cs231n.github.io/understanding-cnn/. [Online; accessed May-2017].

Yu, Fisher and Vladlen Koltun (2015). "Multi-scale context aggregation by dilated convolutions". In: *arXiv preprint arXiv:1511.07122*.

Zhang, Xiangyu et al. (2016). "Accelerating very deep convolutional networks for classification and detection". In: *IEEE transactions on pattern analysis and machine intelligence* 38.10, pp. 1943–1955.