UNIVERSITAT DE BARCELONA

Bachelor's Thesis

Double bachelor's degree in Mathematics and Computer Science

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Learning contextual information via Deep Learning

Author: Sara Bardají Serra

Directors:	**Dr. Santi Seguí**
	Pere Gilabert
Developed in:	**Departament de Matemàtiques**
	i Informàtica

Barcelona, **January 22, 2022**

Abstract

During the last few years, deep learning has become one of the most attractive fields of artificial intelligence, with the use of artificial neural networks at its core. In this project we propose several neural networks architectures for the context learning methodology. The main goal of this project is to verify if these methodologies might work on medical images by first testing them on simpler datasets.

We propose two different approaches, one consisting of a convolutional architecture and the other being a recurrent neural network. Whilst the first approach provided grate results with the first datasets we used, it proved to be insufficient as the complexity of the dataset increased. The recurrent architecture provided successful results when working with more complex datasets.

This thesis provides a general overview of neural networks and explains the different steps taken to reach the proposed models.

Resumen

Durante los últimos años, el *deep learning* se ha convertido en uno de los campos más atractivos de la inteligencia artificial, con el uso de redes neuronales artificiales como foco de atención. En este proyecto proponemos varias arquitecturas de redes neuronales para la metodología de *context learning*. El objetivo principal de este proyecto es comprobar si estas metodologías podrían funcionar en imágenes médicas probándolas primero en conjuntos de datos más simples.

Proponemos dos enfoques diferentes: el primero usa un arquitectura convolucional, mientras que el segundo se basa en una arquitetcura recurrente. Así como, el primer enfoque proporciona buenos resultados con los primeros conjuntos de datos que utilizamos, resulta ser insuficiente a medida que aumenta la complejidad del conjunto de datos. La arquitectura recurrente proporciona resultados satisfactorios al trabajar con conjuntos de datos más complejos.

En esta tesis se ofrece una visión general de las redes neuronales y se explican los distintos pasos dados para llegar a los modelos propuestos.

Resum

Durant els darrers anys, el *deep learning* s'ha convertit en un dels camps més atractius de la intel·ligència artificial, amb l'ús de xarxes neuronals artificials com a focus d'atenció. En aquest projecte proposem diverses arquitectures de xarxes neuronals per a la metodologia de *context learning*. L'objectiu principal d'aquest projecte, és comprovar si aquestes metodologies podrien funcionar en imatges mèdiques testejant-les primer en conjunts de dades més simples.

Proposem dos enfocaments diferents: el primer fa servir una arquitectura convolucional, mentre que el segon es basa en una arquitecura recurrent. Així com, el primer enfocament proporciona bons resultats amb els primers conjunts de dades que utilitzem, els resultats que proporciona són insuficients en augmentar la dificultat del conjunt de dades utilitzat. En canvi, l'arquitectura recurrent proporciona resultats satisfactoris en treballar amb conjunts de dades més complexes.

En aquesta tesi s'ofereix una visió general de les xarxes neuronals i s'expliquen els diferents passos fets per arribar als models proposats.

2021 Mathematics Subject Classification. 68T07

Acknowledgments

First of all, I would like to express my gratitute to my tutors Dr. Santi Seguí and Pere Gilabert for their guidance and advice through this project, but mostly for the support they have given me.

I would also like to thank my family for their constant support, but mostly to my sister Paula for her never-ceasing encouragement.

Lastly, I would like to thank my friends for walking down this road with me for the last six years, helping me whenever I needed it.

Contents

Introduction

Motivation and objectives

Deep learning is a branch of artificial intelligence that has seen an exponential growth in recent years. Today, deep learning models are used in a variety of real-world applications, from image recognition [5] to medical image analysis [14] and self-driving cars [3]. They excel at identifying complex patterns in large datasets, and provide an improved performance over traditional methods. This, joined with the advancement of computing power with graphics processing units and the technological improvements that allow for large data acquisition, has caused deep learning models to be applied more and more to healthcare problems such as computer-aided diagnosis.

One of these technological advancements in the medical field is the Wireless Capsule Endoscopy (WCE). The WCE is a medical procedure used to perform an endoscopy that uses a small pill-shaped capsule instead of the traditional long, flexible tube equipped with a video camera at the end. This equipment provides the patient's medical team with a video, that can be hours long, of his intestines and can be used for polyp detection. A deep learning model can be trained so that given a video obtained through this procedure it outputs the time-stamps of the frames in which a polyp has been detected. These frames can then be shown to a medical team so that they can confirm the results. However, the output might contain more than one frame per polyp, which will cause the medical team having to employ more time to confirm the results.

The purpose of this project is to approach this problem, so that a medical team only has to confirm one frame per polyp that has been detected. Therefore, during this project we aim to create a model that, given a series of images, each belonging to a class, the output of the model determines whether the last image in the series belongs to a class that is already contained in the other images. For its development we will not be working with medical images, but with simpler datasets. This will allow us to test if the methodologies that we propose for context learning work, so that in the future they can be tried on medical images.

Planning of the project

This project took about seven months to complete. I first contacted my tutors in July and decided that before choosing a topic on which to develop my project, it was best I developed some background knowledge on neural networks. For this reason , I completed the Deep Learning Specialization that can be found in Coursera, over the months of July and August, through which I worked on a few models that helped on learning how to create a neural network.

Starting September we decided upon the topic of Context Learning and defined the objectives and the approach to the proposed problem. From this point forward I started to progressively work through the plan until the projects completion. The planning of the project can be found in Figure 1.

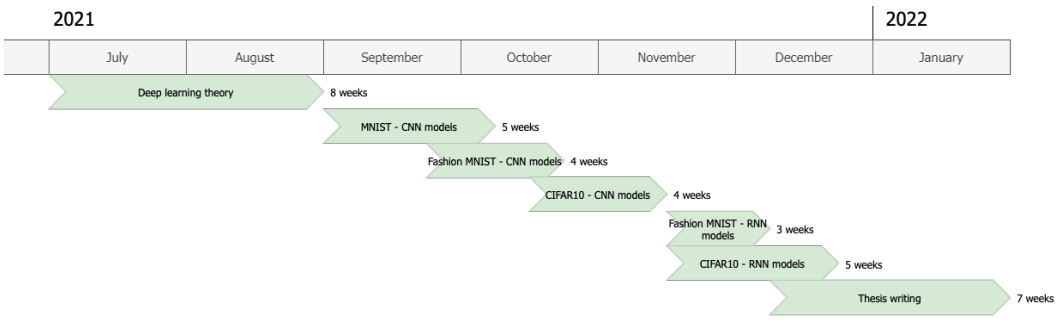


Figure 1: Gantt chart showing the project's timeline.

Structure of the memory

This project follows a structure that goes bottom-up. It starts by setting a solid foundation that includes all the concepts related to artificial neural networks that are necessary to understand the models that have been implemented. It is organized as follows:

1. **Artificial Neural Networks:** Artificial Neural Networks (ANN's) have not been covered in any course of the Computer Science degree at the Universitat de Barcelona. Therefore, a previous research had to be done. This chapter attempts to cover the necessary concepts required to develop this project. It aims to explain what an ANN is and how it works.
   This chapter covers the basic structure of an ANN, and the different types of networks, as well as some of the most commonly used activation functions and layers. It also explains a network's learning algorithm: loss functions, gradient descent and backpropagation.

2. **Context Learning:** This chapter contains an explanation of the models that have been created for this project and aims to reflect the steps that have been taken whenever a new challenge has arisen. Every architecture is discussed in detail and the reasoning behind every choice, explained.

It presents the two basic structures we have used for the models we have developed and explains the steps we have taken in order to present these models.

3. **Results:** Contains the work environment and the frameworks used during the whole development of this project. It also contains a discussion on the obtained results. The proposed models are tested and a clear visualization of the results is shown.
   This chapter presents the experiments we have performed, explaining the different datasets that have been used and offering visualizations that help understand what models offer better results.

4. **Conclusions and future research:** Brief summary of the project. It comments on whether the objectives have been met or not and attempts to reach some conclusions based on the experiments that have been performed.
   It also contains a discussion on possible improvements that can be added to the models and mentions some ideas to further expand the project in the future.

Chapter 1

Artificial Neural Networks

1.1 What are they?

Definition 1.1. *An **artificial neural network**, also referred to as an ANN, is a computing system that attempts to imitate the connections between neurons in the human brain.*

Artificial neural networks are formed by different mathematical processing layers that attempt to make sense of the input they are given, in order to predict a certain output. Similar to the human brain, where neurons are interconnected to one another, ANN's are made up of artificial neurons, also called nodes, that are interconnected and distributed in various layers. Each of these nodes receives an input from several other nodes and multiplies them by assigned weights. A transfer function is then used to collect all the products, which usually consists of the summation of these products. Sometimes, before passing the result to another node, an activation function might be applied to the output (Figure 1.1). These weights are then tuned by a learning algorithm (which will be further discussed in the following sections).

Definition 1.2. *An **artificial neuron** is a connection point in an artificial neural networks. It receives as input a vector $x^T = (x_0, ..., x_n)$, with $x_0 = -1$, and contains a weights vector $w^T = (w_0, w_1, ..., w_n)$, where w_0 is actually denoted as b and referred to as the bias. Then, given an activation function φ, its output is defined as*

$$a = \varphi(w^T x) = \varphi(\sum_{i=1}^{n} w_{ij} x_i - b)\,. \tag{1.1}$$

The process an input x undergoes when it enters a neuron is graphically shown in Figure 1.1.

As it was previously mentioned, neurons that constitute an artificial neural network are distributed in layers. In an ANN we can find three different type of layers.

Definition 1.3. *A **neural network layer** is a collection of neurons that share an input vector and work at a specific depth within an artificial neural network.*

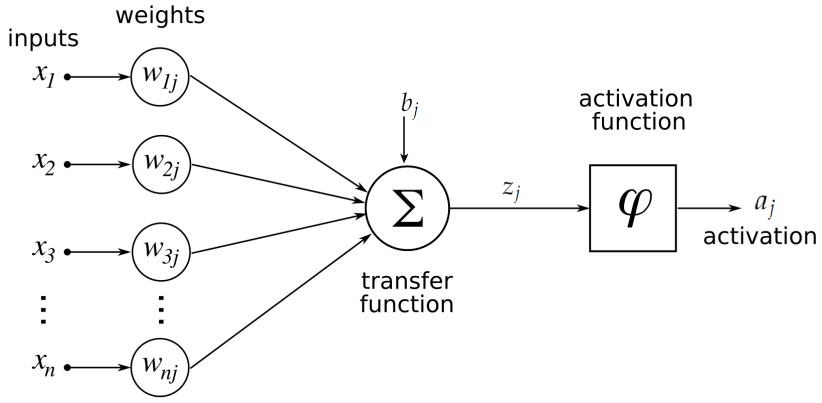


Figure 1.1: Structure of an artificial neuron.

Definition 1.4. *The **input layer** in an artificial neural network is the layer that receives the external data.*

Definition 1.5. *The **output layer** in an artificial neural network is the layer that produces the predicted results.*

Definition 1.6. *We say that a layer in an artificial neural network is a **hidden layer** if its input is given by another layer in the network and its output is received by another layer in the network.*

We say an artificial neural networks is *deep* when it has two or more hidden layers. Also, depending on how layers are connected, we can differentiate between several types of neural network architectures. In Figure 1.2 we can see a diagram of a neural network with two hidden layers. It is therefore, a deep neural network. Also, notice that all the nodes in a layer are connected to all the nodes in the following layer, as we will see in the following section, this means that it is a fully connected neural network.

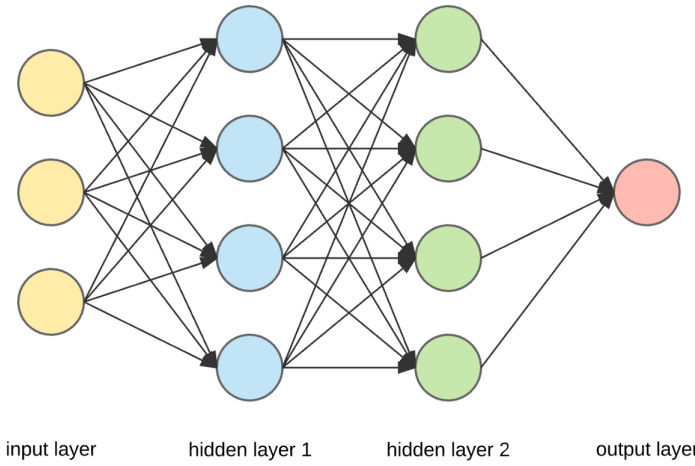


Figure 1.2: Diagram of a neural network with two hidden layers.

It is through this structure and components that ANN's are able to approximate non-linear functions that help solve complex problems like classification, object detection and natural language generation.

1.2 Types of Artificial Neural Networks

As it was previously mentioned, there are several kinds of artificial neural networks and these are mainly distinguished by how different layers of the ANN are connected. In this section we will discuss some of the simpler ANN types and a few more which will appear later in the memory.

1.2.1 Feedforward Neural Netowrks

Feedforward neural networks are one of the simplest forms of ANN's, in which information travels only in one direction, meaning that the data enters through the input nodes, passes through the hidden nodes (if the neural network has any) and exits through the output nodes, where no cycles are formed in between. Therefore, in this type of neural networks, nodes in layer j are connected only to nodes in layer $j+1$.

Fully connected Neural Networks

A fully connected neural network follows the structure of a feedforward network. They consist of a series of fully-connected layers in which every neuron in one layer is connected to every neuron in the following layer. Fully connected neural networks are also known as dense neural networks and can be used for a great variety of applications given that they are 'structure agnostic'. This means that when working with deep neural networks there is no need to make any assumptions about the input, for example, there is no need to consider if the input consists of images or videos. However, while being 'structure agnostic' makes this type of network appropriate for a wide variety of tasks, they also tend to have a weaker performance than other networks that are tuned to the structure of the input.

The main problem we can have when working with this type of networks is that they often need large computational power given the amount of connections or weights that make up the structure of the network.

Another problem is the overfitting that appears when the network has too many weights and therefore has the capacity to learn the necessary weights to get good results with the training set. However, this causes the network to have a poor performance when evaluating the test set. The more layers a network has and the more weights constitute its structure, the more serious this problem becomes.

Convolutional Neural Networks

Convolutional neural networks (CNN's) also follow the structure of a feedforward network. They are a type of neural network where it is often assumed that the input consists of images or speech. Unlike fully connected neural networks, convolutional neural networks are multi-layer, meaning that their structure is built up using more than one type of layer. The different layers that can be used are the convolutional layer and the pooling layer, which we will see more in detail in the following section.

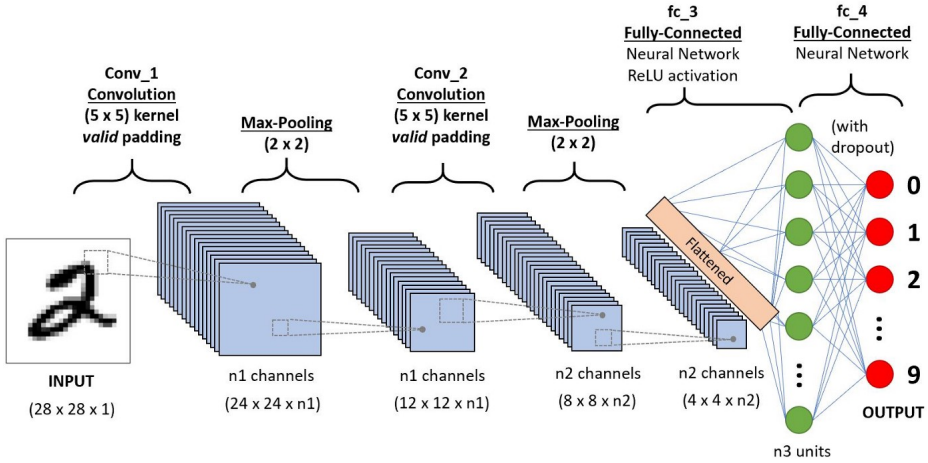


Figure 1.3: A CNN sequence to classify handwritten digits.

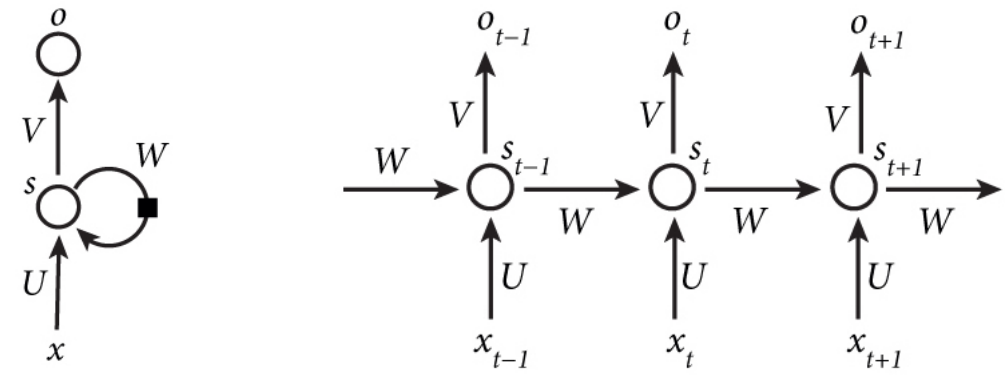
Convolutional neural networks are often used for feature extraction. They allow for the image to be reduced into a form where no critical information is lost and thus can be used in the final learning task. Once the image has been reduced, a few dense layers are added to the network in order to complete the initial prediction using the reduced but representative input.

In Figure 1.3 we can see a CNN sequence that can be used to classify handwritten digits. As we can see, the network consists of two blocks of a convolution layer and a pooling layer, which extract the critical features that are necessary in order to complete the classification task, and these are then fed to two fully connected layer, which actually fulfill the classification task.

1.2.2 Recurrent Neural Networks

Recurrent neural networks (RNN's) are characterized for using sequential data or time series data. They are often used for language translation, natural language processing or speech recognition. A substantial difference with the types of neural networks we have talked about is that in RNN's, the relationships of neurons can form cycles (Figure 1.4). These cycles allow for RNN's to have a certain sense of 'memory' as they take information

from previous outputs and use it to predict the output for the current input.



(a) Rolled RNN (b) Unrolled RNN

Figure 1.4: A typical recurrent neural network.

In Figure 1.4 we can see a typical recurrent neural network structure. To the left it apears rolled, and to the right it appears unrolled. In said figure we can also observe the different parameters of the network:

- x_t is the input in the instant t.
- o_t is the output in the instant t.
- s_t is the 'memory' or state of the network in the instant t.
- W, U and V are the parameters that are shared through all the network.

Notice that o_t is only the output in the instant t. Recurrent neural networks allow for the output given to be either all of the hidden states o_t for $t = 1, ..., m$, where m is the number of timesteps in the input, or for it to be only the last output in the sequence of outputs, o_m. A case in which we would be interested in all of the output hidden states would be when training for temperature forecasting, given that we want the whole sequence of predictions. Whereas if we were training a model that given a sequence of inputs, predicted the following value in the sequence, then we would only be interested in the last output of the output sequence. An example of this second case would be given the input $[2, 4, 6]$, the expected output would be 8.

As we can see from the parameters that a recurrent neural network has, another distinguishing characteristic of RNN's is that they share the same weight parameter (W in Figure 1.4) throughout every layer in the network, whereas feedforward networks have a different weight for every node in a layer.

The main problem we often run into when working with recurrent neural networks is the exploding gradient, which results in an unstable network that can not learn from the training data it is fed.

1.3 Layers

In the previous section we have talked about different types of neural networks and how these are characterized by the layers they are composed of. Many more layers than the ones we are going to comment on here exist, these are the most relevant for this memory.

1.3.1 Fully Connected

A Fully Connected layer, also called Dense layer, is the most general purpose layer as it imposes the least amount of structure to our layers. It is almost always found in a neural network given that it is usually used to control the size and shape of the output layer. This type of layer connects every node in the previous layer to every node this layer itself has. Therefore, it is one of the most time consuming layers to work with.

Given a certain input X, the operation that a fully connected layer performs is the following:

$$Y = \varphi(W \cdot X + B)$$

where Y is the output of the layer, $\cdot$ is the product of matrices, W is the weights matrix, B is the bias and φ is the activation function if any is used. Let $X \in \mathbb{R}^m$, then $W \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^n$ and $Y \in \mathbb{R}^n$, where n is the number of units in the next layer. Therefore, a fully connected layer is a function from $\mathbb{R}^m$ to $\mathbb{R}^n$.

1.3.2 Convolution

A convolution layer is a biologically inspired layer [16] that performs, as its name implies, a convolution (Figure 1.5) over an image. A convolution consists on applying a filter to an input, which results in an activation map.

Definition 1.7. *A **filter kernel**, or filter, is a set of coefficients $w(s,t) \in \mathbb{R}$. If the filter has size $(2a+1, 2b+a)$, then (s,t) runs over a neighbourhood $\mathcal{N} = \{(s,t) \colon -a \leq s \leq a, -b \leq t \leq b\}$.*

Definition 1.8. *The **convolution** of an image f and a filter w of size $(2a+1, 2b+1)$ is defined as*

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t)\,.$$

The convolution operation of f and w is denoted by $w * f(x,y)$.

When working with a convolution layer, the filters are often of size $p \times p$ with p an odd number and there is no need to choose the filter by hand, the network learns the best parameters for its input and the prediction problem in question.

The use of a convolution layer over a fully connected layer when working with images has some advantages, such that, a convolution layer preserves the spatial shape of the image and is capable of finding textures, boundaries and objects in the image. This is achieved with the use of one or more filters in every convolution. Usually, the filters in

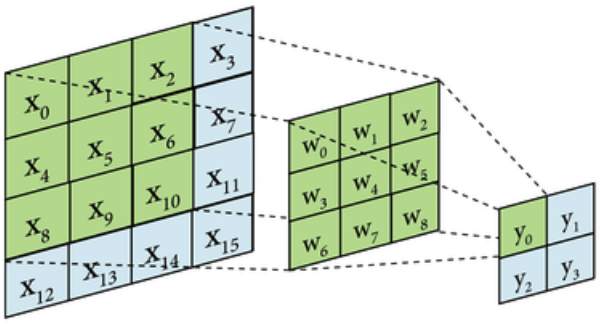


Figure 1.5: Convolution of a 3×3 kernel over an image.

the first layers of a convolutional neural network, detect simple features such as lines or edges. The further into the network we get the more complex features the filters will be able to detect.

This type of layer allows for the following hyperparameters to be tuned:

1. **Number of filters:** determines the capacity of the layer, how many patterns it will learn. The shallower the layer, the fewer filters are used, as there is less information to be gathered.
2. **Filter size:** the size of the filter to be convolved.
3. **Stride:** determines the interval at which the filter is applied, it can be used to skip steps if set to a value other than 1 which will result in a smaller activation map.
4. **Padding:** it can be used so that pixels on the edge of the image act as if they were in the middle of it. It is done by adding 0 padding to the original image.

1.3.3 Pooling

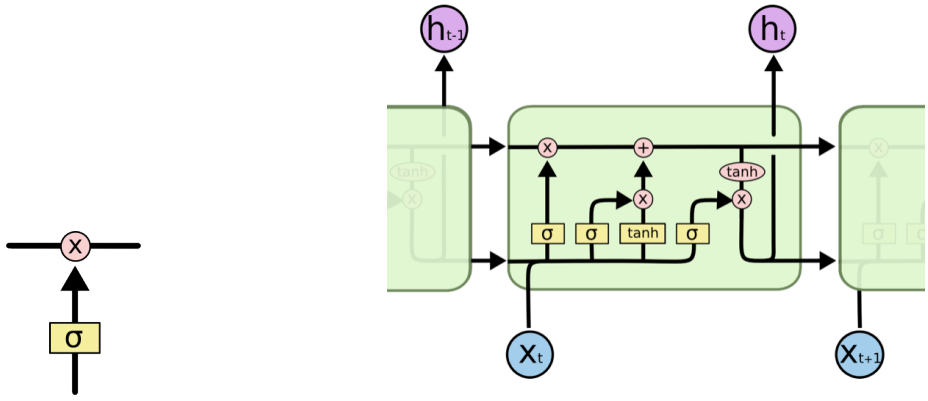
The pooling layer is often used when working with a convolutional neural network, as it performs a non-linear downsampling of the original image, thus reducing the number of parameters that must be computed in the following layer. This layer operates over different regions of the image that fully cover the image. It then outputs a single output for every region resulting in a reduced version of the original image. The operations it can perform over a region are:

1. **Maximum:** over the region, output the maximum value.
2. **Minimum:** over the region, output the minimum value.
3. **Average:** output the average values of the region.
4. **Stochastic:** output a random value of the region.

Just like when working with a convolution layer, we can also tune the filter size and stride to determine the regions the image will be divided into. Usually if a $k \times k$ filter size is used, then a stride of equal size, k in this case, is determined. This way the different regions do not overlap.

1.3.4 Long Short-Term Memory

The long short-term memory layer, often referred to as LSTM [8] is a layer used in recurrent neural networks and it addresses the problem of learning long term dependencies. As we saw in Section 1.2.2, recurrent neural networks contain a hidden state that allows them to have some sort of 'memory'. However they are not capable of handling long term dependencies. LSTM's are designed so that they can learn long-term dependencies in a sequence of data. The way it does this is by having two hidden states, instead of one, known as the hidden state h and the cell state c. Having two states allows the LSTM unit to remember things both long and short-term.



(a) An LSTM gate structure.

(b) Long Short-Term Memory unit

Figure 1.6: LSTM internal workings. Images from [18]

Internally the LSTM has three gates (Figure 1.6b) that allows it to control the information: the forget gate (f) determines what information should be lost, the input gate (i) determines what information to remember and the output gate (o) determines what to predict. Therefore, gates (Figure 1.6a) are a way the unit has to optionally let information through.

The first step that the LSTM unit takes is to determine what information should be kept, and it does this using the following equation:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)\,,$$

where h_{t-1} is the output of the previous state and x_t is the input at instant t.

The next step is to determine what information should be kept by the unit, and it does so

using these equations:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)\,,$$
$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)\,,$$

where i_t determines the information that will be updated and $\tilde{C}_t$ are possible candidates that could be added to the cell state. With the previous computed steps, it then updates the cell state (C_t):

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t\,.$$

Finally, the last step is to decide what the output is:

$$i_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)\,,$$
$$h_t = o_t \cdot \tanh(C_t)\,.$$

1.3.5 Gated Recurrent Unit

The gated recurrent unit [4] is a variation of the LSTM layer that attempts to solve the vanishing gradient problem when working with standard recurrent neural networks. The way it does this is by combining the forget and input gate into one update gate, so the GRU has an update gate and a reset gate. With these two gates GRU can keep information from long ago without it gradually fading away. In Figure 1.7 we can see the internal structure of a gated recurrent unit.

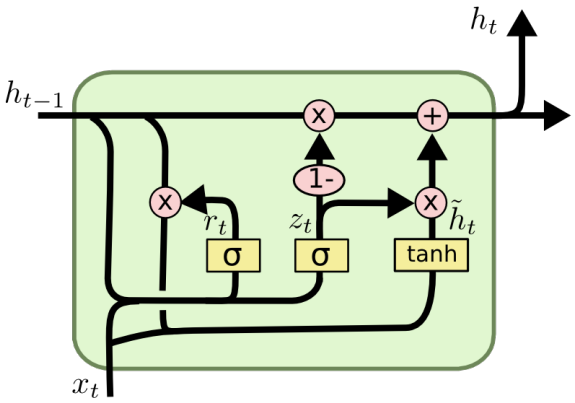


Figure 1.7: Gated Recurrent unit.

1.4 Activation Functions

Definition 1.9. *An **activation function** is a function $\varphi\colon \mathbb{R}^n \longrightarrow A \subset \mathbb{R}^n$ that delivers an output based in its input.*

Activation functions are a critical part of neural networks given that the activation function that is chosen will affect the capability and performance of the neural network. Activation functions are desired to have the following properties:

- **Non-linearity:** Although it is not an essential condition this property is often desired so that the outputs are not a linear combination of the inputs. Also, it prevents a multi-layered network from behaving as if it had only one.
- **Continuously differentiable:** This property is needed in order to enable the use of gradient-based optimization methods. We will see an example of a function that is not continuously differentiable, but that can still be used as an activation function. Therefore, this property is not completely necessary.
- **Range:** If the range of the activation function is finite, it has been shown that gradient-based training methods tend to be more stable. If the range of the activation function is infinite, it has been shown that training is usually more efficient.

In this section we will talk about some of the most well-known and most commonly used activation functions and those that have relevance in this memory.

1.4.1 Linear

The linear function (Figure 1.8a) is the most basic function. It is defined by the identity function and so its domain and range are both $\mathbb{R}$. Obviously, this function is continuously differentiable. However, we can see that it does not perform any non-linearity so the complete network will act as a model with only one layer.

Its mathematical formula is:

$$\varphi(x) = x\,. \tag{1.2}$$

1.4.2 Rectified Linear Unit

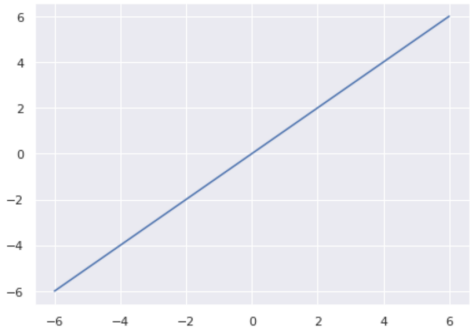
The Rectified Linear Units (ReLU) (Figure 1.8b) are an improvement of the linear function. They have shown to improve gradient stability in the learning algorithm by being less susceptible to vanishing gradients. This allows networks to be deeper and learn better features.

Its mathematical formula is:

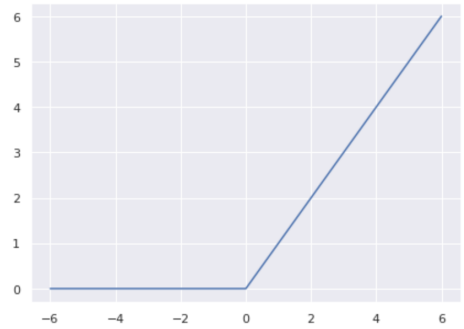
$$\varphi(x) = \begin{cases} x\,, & \text{if } x > 0\,, \\ 0\,, & \text{if } x \leq 0\,. \end{cases} \tag{1.3}$$

This function has domain $\mathbb{R}$ and range $[0, \infty)$. It is also continuous and so it is of class $\mathcal{C}^0$.

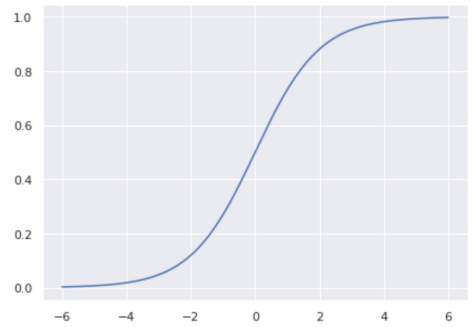
Notice that this function is not differentiable at $x = 0$ and so it is not continuously differentiable. However, it can still be used as an activation function, as training algorithms do not often reach local minima of the loss function. Therefore, it is acceptable for these minima to correspond to points with an undefined gradient. Nevertheless, if a local minimum with an undefined gradient is reached, the training algorithm returns one of the one-sided derivatives instead of raising an error.



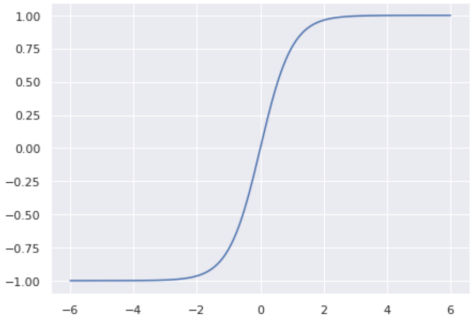
(a) Linear



(b) Rectified Linear Unit



(c) Sigmoid



(d) Hyperbolic Tangent

Figure 1.8: Activation functions.

1.4.3 Sigmoid

The sigmoid function (Figure 1.8c) is defined in $\mathbb{R}$ and maps any input value into the range $[0, 1]$.

The activation is given by the following mathematical function:

$$\varphi(x) = \frac{1}{1 + e^{-x}}\,. \tag{1.4}$$

This activation function has been widely used due to its easy and fast to compute derivative.

1.4.4 Hyperbolic Tangent

The hyperbolic tangent (Figure 1.8d) is also defined in $\mathbb{R}$, however, it maps an input value into the range $[-1, 1]$. Its main difference with the sigmoid function is that it has a steeper slope and wider range.

The activation is given by:

$$\varphi(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}\,. \tag{1.5}$$

1.5 Learning algorithm

Until now we have talked about weights and how different layers compute an output using these weights and biases. However, weights and biases are randomly initialized and therefore their best value must be learnt by the neural network in order to make accurate predictions. The way a neural network learns the optimal values for its weights and biases is through and error gradient. When fixing the values for the weights and biases we want to know whether they are too large or too small with respect to their optimal value and how much they deviate from said value. Therefore, the gradient that must be analyzed is that of error with respect to weights and biases:

$$\frac{\partial E}{\partial W}\,, \quad \frac{\partial E}{\partial b}$$

where E is an error, W is the weight and b is the bias.

1.5.1 Loss function

As we have seen, neural networks need a way to measure the error in its prediction. The way they do this is by using a loss function (L) that measures the error of the output of a neural network with respect to the desired output and during the training phase of the neural network, the learning algorithm aims to minimize it. Since the error is given by the loss function, then $E = L_y(\hat{y})$, where E is the error we mentioned before, and $L_y(\hat{y})$ denotes the loss function applied to the predicted output.

These are some examples of loss functions:

- **Mean Squared Error (MSE):** It is usually used in regression predictive modeling problems in which a real-valued quantity is predicted.

$$L_y(\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2\,, \tag{1.6}$$

  where $\hat{y}$ is the predicted output vector by the neural network and y is the desired output vector.

- **Binary Cross-Entropy:** This loss function is used in binary classification problems, meaning classification into two classes, for which target values are in the set $\{0, 1\}$.

$$L_y(\hat{y}) = -y \cdot \log(\hat{y}) - (1 - y) \cdot \log(1 - \hat{y})\,. \tag{1.7}$$

  Given this formula, if $\hat{y}_i \approx y_i$, then $L_{y_i}(\hat{y}_i)$ is very close to 0. Otherwise, $L_{y_i}(\hat{y}_i)$ becomes a very large number.

- **Categorical Cross-Entropy:** This loss function is used in multi-class classification problems.

$$L_y(\hat{y}) = -\sum_{i=1}^{C} y_i \cdot \log(\hat{y}_i)\,. \tag{1.8}$$

Usually, the output of a multi-class classification model is passed through a softmax activation function that normalizes the input into C probabilities proportional to the exponential of the input. Thus, $\sum_{i=1}^{C} \hat{y}_i = 1$.

1.5.2 Backpropagation

Backpropagation is a method used to calculate the error gradient. The idea with backpropagation is for every node in the network to assume a share of the blame for the total error. In order to do this, the chain rule is used to compute the partial derivatives of the loss function L with respect to the individual weights.

Let us define the following notation:

- w_{ij} denotes the weight between node i from the previous layer and node j form the current layer.
- a_j is the output of the neuron.
- z_j is the weighted sum of the outputs a_k of the previous neurons. If the neuron is in the first hidden layer, then a_k is just the input x_k.
- g denotes the activation function.

Then,

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}\,. \tag{1.9}$$

Observe however, that $\frac{\partial z_j}{\partial w_{ij}}$ is the same as the output of the previous layer, a_i, and so the error is propagated form the last layer to the first one.

The second factor in (1.9) is easily calculated as it is simply the partial derivative of the activation function:

$$\frac{\partial a_j}{\partial z_j} = \frac{\partial g(z_j)}{\partial z_j}\,. \tag{1.10}$$

When the node j is in the output layer, the computations of the first factor in (1.9) are pretty straightforward as $a_j = y$ and

$$\frac{\partial L}{\partial a_j} = \frac{\partial L}{\partial y}\,. \tag{1.11}$$

However, in case we want to know the gradient error of a layer in the middle of the neural network, then derivative of L with respect to a_j is not as simple to calculate. It is given by the following formula:

$$\frac{\partial L(a_j)}{\partial a_j} = \sum_{l \in L} \frac{\partial L}{\partial z_l} \cdot \frac{\partial z_l}{\partial a_j} = \sum_{l \in L} \frac{\partial L}{\partial a_l} \cdot \frac{\partial a_l}{\partial z_l} \cdot w_{jl}\,, \tag{1.12}$$

where L is the set of nodes receiving input from node j.

Once these computations have been completed, the weights can be updated using an optimization algorithm or optimizer.

1.5.3 Optimizer

An optimizer is an algorithm used to update the weights and biases of a neural network in order to reduce the loss. Many different optimizers are used today. Some of the most well-known are: Momentum [19], AdaGrad [6], Adam [11] and RMSProp. However, they all run based on the same idea, the gradient descent optimization algorithm.

Gradient descent

The gradient descent is an optimization algorithm that iterates in order to minimize the output of the loss function and find a local minimum. Using this method we update the weights of the neural network.

As previously mentioned, weights are randomly initialized in a neural network. Then the network evaluates its input using these weights and computes the cost. Once the cost is known then its derivative is calculated as it indicates the slope at that point of the function and thus, it indicates the direction in which to vary the weights in order to obtain a lower cost in the next iteration. The operation is as follows:

$$(w_{ij})_{n+1} = (w_{ij})_n - \alpha \frac{\partial L}{\partial (w_{ij})_n}\,, \tag{1.13}$$

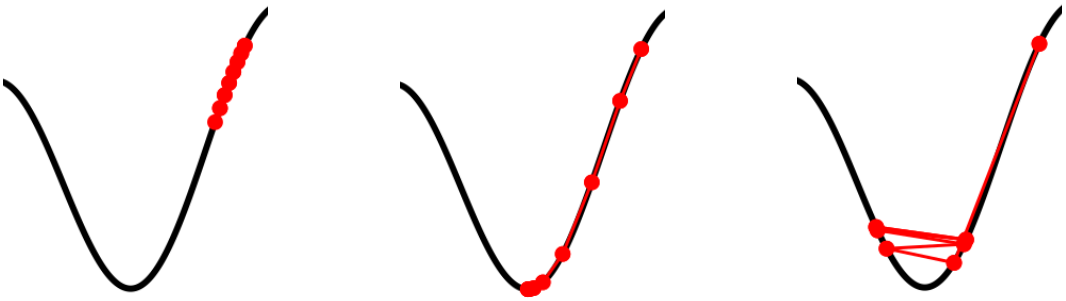
where $(w_{ij})_n$ is the value of the weight w_{ij} at step n, L refers to the loss function and $\alpha > 0$ is a hyperparameter called the *learning rate*. The partial derivative indicates in which direction the weights should vary, and the learning rate indicates the velocity at which to move. This value is tuned by the user and should be chosen so that the function converges fast but does not take steps that are too big, otherwise the local minimum might be missed.

This method requires for the whole dataset to be processed by the neural network and backpropagated at every iteration. Therefore, when working with large datasets it might take a long time. For this reason, gradient descent is no longer used and it has been mostly replaced by the stochastic gradient descent method, which replaces the gradient calculated from the entire dataset for an estimate of it, computed from a subset of the dataset.

Leanring rate

Definition 1.10. *The **learning rate** is a hyperparameter, denoted by α, used in an optimization algorithm that determines the step size taken at each iteration towards the minimum of a chosen loss function.*

When the learning rate is set, there is a trade-off between the rate of convergence and 'overshooting'. 'Overshooting' happens when the learning rate is set to a value that is too high and therefore the optimization method ends up diverging from the minimum. On the other hand, if the learning rate value is too low, then the training process will make progress very slowly, as only very small updates are being made to the weights.



(a) Too low (b) Right value (c) Too high

Figure 1.9: Diagrams representing the problems with picking a value too high or too low for α, and what happens when an ok value is picked.

A technique called *learning rate annealing* can be used instead of setting a fixed learning rate. This technique consists on starting with a higher learning rate and decreasing it as the training progresses. With this decay in the learning rate value, we can quickly reach a range with 'good' parameter values, and once these value are reached, a slower exploration of the loss function starts.

Chapter 2

Context Learning

2.1 Problem explanation

As it was briefly mentioned in the introduction, in this project we are aiming to create a model that given a series of images, each belonging to a class, it determines whether the last image belongs to a class that is already represented by the other images.

In order to solve this problem, we will be working with strips of images. The notation we will use is as follows: given a strip of n images, we are going to refer to the first $n-1$ images as the context images and the last image will be the target image, as it is the image for which we want to see if its class is already represented in the context images. We will also refer to the strip-size as the number of context images in the strip. An example of this notation can be seen in Figure 2.1.

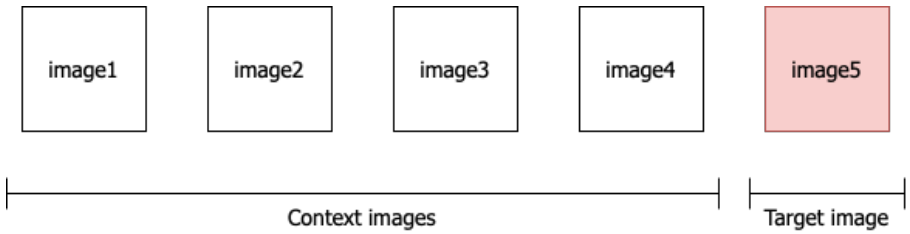


Figure 2.1: Input example with four context images and a target image, therefore it has a strip-size of 4.

Given that we want the model to detect whether the target image represents a repeated class in the context images, this is a binary classification problem. We will expect the model to output a 0 if the class represented by the target image is not contained in the set of the classes represented by the context images. To the contrary, we will expect the model to output a 1, if the class represented by the target image is already contained in the set of the classes that are represented by the context images. In Figure 2.2 we can see an example for each possible outcome. Given the input seen in Figure 2.2a, the model is

expected to classify it as a 0, given that there are no 0's i the context images. On the other hand, given the input shown in Figure 2.2b, the model is supposed to classify it as a 1, since both the first and fourth images of the context images also represent a 5.

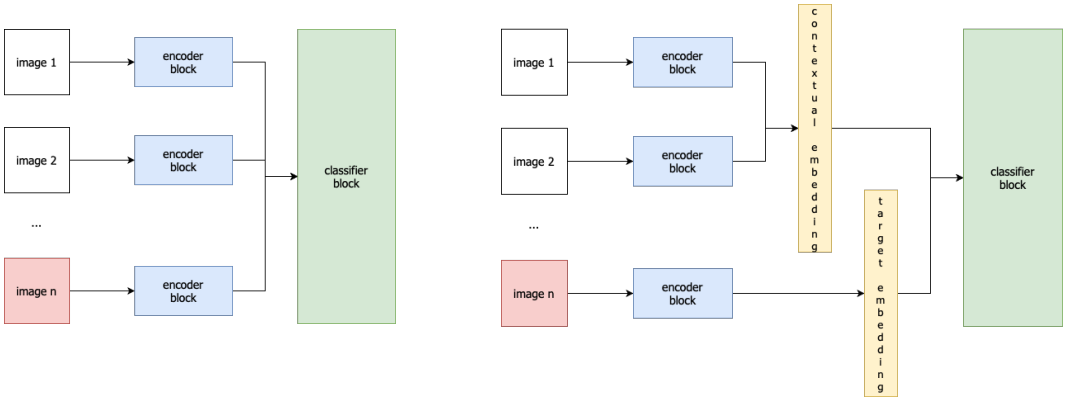
(a) Input example with expected output 0.

(b) Input example with expected output 1.

Figure 2.2: Examples of expected outputs given strips of images created with the MNIST datset.

2.2 Model structure

All models developed for this project follow one of two structures. A diagram representing each structure is shown in Figure 2.3. As it can be seen, both structures have two common blocks: the encoder block and the classifier block.



(a) Structure 1

(b) Structure 2

Figure 2.3: Model structures.

Before seeing in detail these structures, notice that the images we work with can be seen as

elements of $M_{w\times h}([0,1])$, where $M_{w\times h}([0,1])$ denotes the set of matrices of size $w \times h$ with coefficients in $[0,1]$. Then, the common blocks in our models can be defined as follows:

- **Encoder block:** This block expects an input consisting of a single image, not a strip of images, and outputs an encoded version, an embedding, of that image with reduced dimension. When an image of size $w \times h$ is processed by this block the output is a vector of size $latent\_dim$. This vector is supposed to be a translation of the original image into a low-dimensional space, but still containing all the important information about its features.

  This block can then be defined as a function

  $$e : M_{w\times h}([0,1]) \longrightarrow \mathbb{R}^{latent\_dim}$$

  that given an image $I \in M_{w\times h}([0,1])$, computes its embedding $e(I) = i \in \mathbb{R}^{latent\_dim}$. We have denoted the function as e so that it can be easily identified as the function performed by the encoder block.

- **Classifier block:** This block is in charge of performing the actual classification. Meaning, given a certain number of embeddings, it must classify its input.

  Again, this block can be defined as a function

  $$c : \mathbb{R}^{latent\_dim\times m} \longrightarrow [0,1]$$

  where m denotes the number embeddings the blocks receives as input. The notation c has been chosen for the function so that it can be easily related to the classifier block.

However, the encoder block is applied individually to every image in the strip, whereas the classifier block is applied to some combination of the embeddings obtained by the encoder block. Consequently, the whole model can be defined as a function, but we must first define some auxiliary functions.

Let $E : M_{w\times h}([0,1])^n \longrightarrow \mathbb{R}^n$ be defined as $E(I_1, I_2, ..., I_n) = (e(I_1), e(I_2), ..., e(I_n))$, where n denotes the number of images in a strip and e refers to the function we previously defined for the encoder block, and let $f : \mathbb{R}^{latent\_dim\times n} \longrightarrow \mathbb{R}^{latent\_dim\times m}$ denote the function used to combine the image embeddings.

Then, the whole model can be defined as:

$$\begin{aligned} M\colon M_{w\times h}([0,1])^n &\longrightarrow [0,1] \\ (I_1, I_2, ..., I_n) &\longmapsto (c \circ f \circ E)(I_1, I_2, ..., I_n) \end{aligned}$$

As we can see in Figure 2.3a, the first structure uses the embeddings obtained from all the images in the strip, including the target image in order to perform the classification,

and so, in the functions we have defined above, $n = m$. On the other hand, the structure displayed in Figure 2.3b shows how the classifier block receives two embeddings as image: the contextual embedding and the target embedding, hence now $m = 2$. The contextual embedding is calculated using the embeddings obtained from the context images (we will explore this further in Section 2.4), whereas the target embedding is simply the embedding obtained from the target image.

2.3 Encoder block

In this section we will see the models that have been tried as the encoder block in the model structure seen in Section 2.2.

This section will be divided into two subsections: the first, called CNN, contains the information of the encoder models that are not well-known model architectures and the second, called EfficientNet, as its name indicates, contains an explanation of the EfficientNet architecture, which has also been used during this project for the encoder block.

2.3.1 CNN

Throughout this project we have used several different convolutional architectures in the encoder block of the model structure. This is because we have worked with several datasets (which we will see later in Section 3.2) and so in order to adapt the model to their unique properties and the complexity of their input, each dataset uses a different architecture for the encoder block.

The structure of the three convolutional encoders that have been used can be seen in Tables 2.1, 2.2 and 2.3. All three of them have two columns: the first indicates the type of layer, whilst the second shows the shape of the output of that layer. The final output shape of all three of them is also the same: $(\text{None}, \textit{latent\_dim})$, which means they all output a vector of dimension *latent_dim*, where *latent_dim* is a hyperparameter that is set and indicates the dimension of the output vector.

Notice that the encoder from Table 2.1, has no MaxPooling layers. This is because this encoder is designed for small and simple datatsets. Meaning, it is for datasets that are made up of small images and that do not contain much information. Also, observe that this first encoder is shallower, has less layers, than the other two. Again, this is also because it is thought for it to be used in datasets that do not contain complex patterns, otherwise it would just end up memorizing the expected output for every input, which would result in it overfitting and not generalizing well to other instances of the same data with which it has not been trained.

Conversely, observe that the other two encoders (Tables 2.2 and 2.3) are deeper than the first. That is because they are thought to work with datasets that contain slightly more complex patterns, albeit still consisting of relatively small images.

Layer (type)	Output Shape
Convolutionl2D	(None, 28, 28, 1)
BacthNormalization	(None, 28, 28, 1)
ReLU	(None, 28, 28, 1)
Convolutionl2D	(None, 14, 14, 32)
BatchNormalization	(None, 14, 14, 32)
ReLU	(None, 14, 14, 32)
Convolutionl2D	(None, 14, 14, 64)
BatchNormalization	(None, 14, 14, 64)
ReLU	(None, 14, 14, 64)
Flatten	(None, 12544)
Dense	(None, *latent_dim*)

Table 2.1: Convolutional encoder 1.

Layer (type)	Output Shape
Convolutional2D	(None, 28, 28, 32)
BatchNormalization	(None, 28, 28, 32)
Convolutional2D	(None, 28, 28, 32)
BatchNormalization	(None, 28, 28, 32)
MaxPooling2D	(None, 14, 14, 32)
Convolutional2D	(None, 14, 14, 64)
BatchNormalization	(None, 14, 14, 64)
Convolutional2D	(None, 14, 14, 64)
BatchNormalization	(None, 14, 14, 64)
MaxPooling2D	(None, 7, 7, 64)
Convolutional2D	(None, 7, 7, 128)
BatchNormalization	(None, 7, 7, 128)
Convolutional2D	(None, 7, 7, 128)
BatchNormalization	(None, 7, 7, 128)
Convolutional2D	(None, 7, 7, 256)
BatchNormalization	(None, 7, 7, 256)
Convolutional2D	(None, 7, 7, 256)
BatchNormalization	(None, 7, 7, 256)
Flatten	(None, 12544)
Dense	(None, *latent_dim*)

Table 2.2: Convolutional encoder 2.

Layer (type)	Output Shape
Convolutional2D	(None, n, m, 32)
BatchNormalization	(None, n, m, 32)
Convolutional2D	(None, n, m, 32)
BatchNormalization	(None, n, m, 32)
MaxPooling2D	(None, $\frac{n}{2}$, $\frac{m}{2}$, 32)
Dropout	(None, $\frac{n}{2}$, $\frac{m}{2}$, 32)
Convolutional2D	(None, $\frac{n}{2}$, $\frac{m}{2}$, 64)
BatchNormalization	(None, $\frac{n}{2}$, $\frac{m}{2}$, 64)
Convolutional2D	(None, $\frac{n}{2}$, $\frac{m}{2}$, 64)
BatchNormalization	(None, $\frac{n}{2}$, $\frac{m}{2}$, 64)
MaxPooling2D	(None, $\frac{n}{4}$, $\frac{m}{4}$, 64)
Dropout	(None, $\frac{n}{4}$, $\frac{m}{4}$, 64)
Convolutional2D	(None, $\frac{n}{4}$, $\frac{m}{4}$, 128)
BatchNormalization	(None, $\frac{n}{4}$, $\frac{m}{4}$, 128)
Convolutional2D	(None, $\frac{n}{4}$, $\frac{m}{4}$, 128)
BatchNormalization	(None, $\frac{n}{4}$, $\frac{m}{4}$, 128)
MaxPooling2D	(None, $\frac{n}{8}$, $\frac{m}{8}$, 128)
Dropout	(None, $\frac{n}{8}$, $\frac{m}{8}$, 128)
Flatten	(None, $\frac{n}{8} \cdot \frac{m}{8} \cdot 128$)
Dense	(None, *latent_dim*)

Table 2.3: Convolutional encoder 3.

2.3.2 EfficientNet

EfficientNet is a convolutional neural network achitecture that was published in 2019 [24] with the intention of improving not only the accuracy, but also the efficiency of models. When the paper was published, 8 EfficientNet models were published with it, from B0 to B7, and although they are built using the same main block, for this project we have

chosen to work with the EfficientNetB0, given that it is the smallest of them all, with the least parameters.

The basic block structure of the EfficientNetB0 can be seen in Figure 2.4. At the end of the structure shown in the image there is a Convolution 1×1 and a GlobalAveragePooling2D. We have also added a Dense layer with *latent_dim* units in order to control the size of the output vector of the EfficientNet-based encoder. As it is shown, its main building block is the mobile inverted bottleneck MBConv [21, 23], with squeeze-and-excitation optimization [9].

The idea behind the MBConv block is to first do a Convolution 1×1 that increases the number of filters. This is then followed by a DepthWiseConvolution that is either 3×3 or 5×5 in the case of the EfficientNet. Finally another Convolution 1×1 is added in order to return to the original number of filters. Also, behind every convolution there is a BatchNormalization layer and there is a skip connection between the narrower parts of the block. Using the DepthWiseConvolution instead of a Convolution layer, this block saves time and memory.

Finally, the squeeze-and-excitation optimization is also used in order to better map channel dependency, while maintaining access to global information.

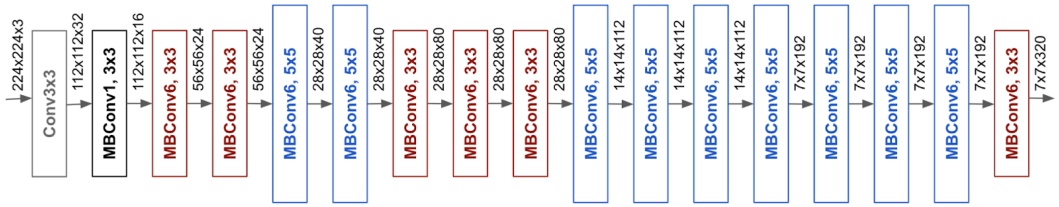


Figure 2.4: A basic block representation of EfficientNetB0.

2.4 Contextual embedding

In Section 2.2 we commented on how the model structure shown in Figure 2.3b feeds two embeddings to the classifier block, one being the contextual embedding and the other being the target embedding. We also defined a function f to denote the operations performed with the embeddings in order to prepare them as input for the classifier block.

For the models that follow the first structure, this function is simply the identity function, which is defined as:

$$f : \mathbb{R}^{latent\_dim \times n} \longrightarrow \mathbb{R}^{latent\_dim \times n}$$
$$(i_1, i_2, \ldots, i_n) \longmapsto (i_1, i_2, \ldots, i_n)$$

On the other hand, in the models that follow the second structure, the target embedding is obtained from the embedding of the target image, where as the contextual embedding

is calculated with an aggregation function that receives the first $n-1$ embeddings. In this case, the f function can be defined as:

$$f\colon \mathbb{R}^{latent\_dim \times n} \longrightarrow \mathbb{R}^{latent\_dim \times 2}$$
$$(i_1, i_2, \ldots, i_n) \longmapsto (g(i_1, \ldots, i_{n-1}), i_n)$$

where $g\colon \mathbb{R}^{latent\_dim \times m} \longrightarrow \mathbb{R}^{latent\_dim}$ denotes the aggregation function.

In this section we are going to discuss the different operations that have been tested as the aggregation function in order to obtain the contextual embedding.

The notation we will use is the following:

- Given a vector $i_m \in \mathbb{R}^{latent\_dim}$, we will denote by $(i_m)_j$ the element in position j in the vector i_m.

2.4.1 Combining contextual embeddings

The operations that we have tried for combining the embeddings from the context images into only one embedding are:

Maximum operation

The first operation we have used to combine the embeddings is the maximum operation. Given n context embeddings then the maximum operation builds the resulting contextual embedding by taking the maximum value for every position in all n embeddings. This aggregation function is defined below and an example of an expected out, given 5 embeddings is shown in Figure 2.5a.

$$g\colon \mathbb{R}^{latent\_dim \times m} \longrightarrow \mathbb{R}^{latent\_dim}$$
$$(i_1, i_2, ..., i_m) \longmapsto \left(\max_{j=1,...,m} (i_j)_1, ..., \max_{j=1,...,m} (i_j)_{latent\_dim} \right)$$

Given that in an ideal case, the embedding produced by the encoder would represent the class to which its input image belongs to, then the contextual embedding calculated using the maximum operation would represent all the classes to which every context image in the strip belongs to.

In the example shown in Figure 2.5a, the first embedding belongs to class one, the second to class three, the third to class two and the last to belong to class nine. Then, the contextual embedding conveys that the context images belong to one of four classes: one, two, three or nine.

Addition operation

The second operation we have tried for combining the context embeddings is the addition operation. Then, given n embeddings, the resulting contextual is calculated by adding

the value in every embedding for every embedding. An example of the expected output when using this operation is shown in Figure 2.5b. As we can see, the initial embeddings belong to classes: one, two, three and nine, with two embedding representing class nine. Therefore, the resulting vector has a 1 in positions one, two and three, and a 2 in position nine.

This aggregation function is defined as follows:

$$\begin{aligned} g\colon \mathbb{R}^{latent\_dim \times m} &\longrightarrow \mathbb{R}^{latent\_dim} \\ (i_1, i_2, \ldots, i_m) &\longmapsto \left(\sum_{j=1}^{m} (i_j)_1, \ldots, \sum_{j=1}^{m} (i_j)_{latent\_dim} \right) \end{aligned}$$

Average operation

The third operation we have used when combining the embeddings produced by the context images is the average operation. Given n context embeddings, then the average operations computes the resulting contextual embedding by taking the average value in every position over the n embeddings. This aggregation function is defined below and in Figure 2.5c, we can see an example of the expected output when working with this operation.

$$\begin{aligned} g\colon \mathbb{R}^{latent\_dim \times m} &\longrightarrow \mathbb{R}^{latent\_dim} \\ (i_1, i_2, \ldots, i_m) &\longmapsto \left(\frac{\sum_{j=1}^{m} (i_j)_1}{m}, \ldots, \frac{\sum_{j=1}^{m} (i_j)_{latent\_dim}}{m} \right) \end{aligned}$$

As we can see from the examples, the output given by the addition and the average are proportional. Therefore, the difference is that the values in the contextual embedding calculated with the average operation are all between 0 and 1, where those calculated with the addition are not in any given range.

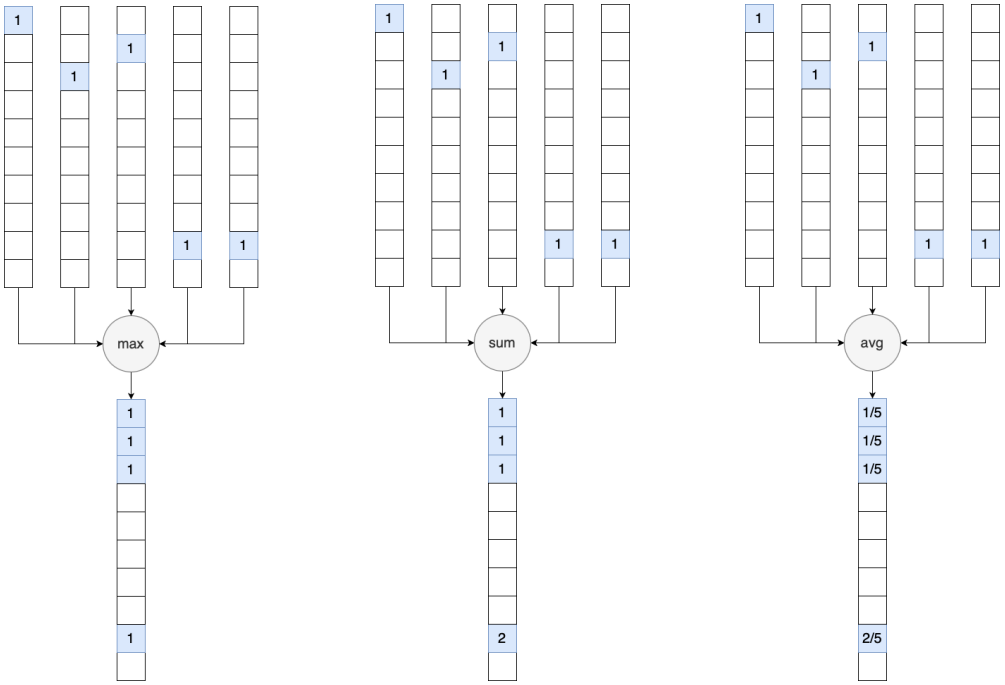
Dense layer

Another method we have used to combine the embeddings into the contextual embedding is by applying a dense layer that received the stack of embeddings as input and gave a vector of size *latent_dim*, the same as the input embeddings, as an output.

Given that a dense layer tends to find the best parameters for an unknown operation, we hoped that in this case it would find the best way, the best possible aggregation function g, to combine all of the embeddings into one.

2.5 Classifier block

Unitl now we have talked about the encoder block, and how the embbedings of different images are combined to obtain the contextual embedding. In this section we are going



(a) Maximum (b) Addition (c) Average

Figure 2.5: Example of the result of combining 5 embeddings using (a) maximum, (b) addition, (c) average operation.

to talk about the final part of the model structure: the classifier block.

The classifier block is the one that receives the embeddings, either all of them or the contextual and target embedding, and does the final classification to detect whether the target image represents a class that is already being represented by the context images.

Throughout the project, we have used different models for the classifier block, each of them adapting the model to the complexities of the datasets we have used.

2.5.1 Dense classifier

One of the models that has been used for the classifier block is a dense model. The model consists of a flatten layer and three dense layers with 128, 64 and 1 unit respectively. Table 2.4 shows the detailed structure of the model.

This particular classifier block has been used with the model structure shown in Figure 2.3b, and so it receives as input both the contextual and the target embedding.

Layer (type)	Output Shape	Param #
Flatten	(None, $latent\_dim \times 2$)	0
Dense	(None, 128)	8320
Dense	(None, 64)	8256
Dense	(None, 1)	65

Table 2.4: Dense classifier.

2.5.2 Aiming for linear independence between contextual and target embedding

On an attempt to try and increase the accuracy of the model, in this case with the structure shown in Figure 2.3b, we also tried a classifier block consisting of a dot product layer. The dot product is an algebraic operation that takes two equal length-like number sequences and returns a single number. The geometric definition of the dot product of two vectors a and b is given by the following equation:

$$a \cdot b = |a|\,|b| \cos\theta\,, \tag{2.1}$$

where θ is the angle between a and b.

This operation is often used to tell whether two vectors are linearly independent or not, given that if the dot product of two vectors is 1, then these vectors are proportional to each other, meaning that one is the other multiplied by a scalar. On the other hand, if the dot product of two vectors is 0, then the vectors are orthogonal.

In our case, we wanted the model to predict a 1, indicating class repetition, or a 0, indicating no class repetition. Therefore, in hopes of provoking the model to encode different classes as linearly independent embeddings, we built the classifier block using only one basic operation: the dot product. Consequently, when the classifier block received the contextual and the target embedding, we would expect the output to be a 1 if these embeddings were linearly independent, and a 0 if they were proportional to one another.

Following this same idea, we also implemented a model whose classifier block consisted of a dot product layer, followed by a dense layer of 1 unit. The idea behind this classifier block was the same as before, but including the dense layer with one unit so that this final layer could make any necessary adjustments to the output given by the dot product layer.

2.5.3 RNN classifier

A different approach for the classifier block was using a recurrent architecture for it. In Section 1.2.2 we mentioned how recurrent neural networks are used with sequential data, in our case, the sequential data consists of the strips of images, each image of the strip being a value of the sequence. Table 2.5 shows the complete structure of the architecture that has been used. Notice that the input layer has (None, None, $latent\_dim$) as output shape. The first value of the shape is always None as it denotes the number of examples

and so it may vary. In this case, the second value denotes the number of images forming the strip, and by being defined as None, it allows for the classifier block to be defined and used independently of the number of images in every strip of images.

This classifier block has been used for the model structure defined in Figure 2.3a, and so it expects as input all of the embeddings obtained by the encoder block.

Layer (type)	Output Shape	Param #
InputLayer	(None, None, latent_dim)	0
GRU	(None, None, 256)	222720
GRU	(None, None, 128)	148224
GRU	(None, 64)	37248
Dense	(None, 128)	8320
Dense	(None, 64)	8256
Dropout	(None, 64)	0
Dense	(None, 64)	4160
Dense	(None, 1)	65

Table 2.5: RNN classifier block.

By changing the classifier block into a recurrent neural network, we intended for the model to use its 'memory' to remember only the important information of every image in the strip.

2.6 Handling coloured images

Some of the images we have worked with are grey-scale images, but we have also worked with RGB datasets. When working with coloured images we have tried giving the three channeled images as input, but we have also tested different techniques for transforming a three channelled image into one of one channel only, and tried the same models with the same images but transformed to one channel using these techniques. This way, we tested if working with RGB images resulted in the models having a better performance or whether they worked better with one-channel images.

The different techniques we tried are the following:

- Maximum of three channels: A simple way of transforming a three-channelled image into one of one channel is by taking the maximum value of every pixel in the three channels. If a pixel has values (1, 0.5, 0.25), then the resulting image will have a 1 in that pixel's position.
- Flattened image: Consists on passing the image through a Flatten layer and then reshaping the image. This way, an image with shape (n, m, 3) is transformed into one with shape (n, $m \times 3$).

- Convolution layer: By using a convolution layer with one filter of size (1, 1) and stride 1 then a three channelled image is transformed into one of one channel. By incorporating the layer into the model, we would expect it to learn the best weights to make the transformation.
- RGB to black and white function: Several functions for transforming an RGB image into a black and white image already exist. They work like a convolution layer, which has been explained above, but the weights have already been fixed and so it does not need training. In this case, we have used the function rgb_to_grayscale from *Tensorflow*.

Chapter 3

Results

This section discusses the results obtained from the experiments that have been performed with the previously explained architectures and presents the different datasets used to train and test these models.

3.1 Working environment

In this section we will present the software and hardware technologies used throughout the project.

Due to the characteristics of this project, both a substantial amount of computational power and of memory capacity are required. Whilst the former is due to the amount of models that need training, the latter is in view of the fact that every example in our dataset is formed not by one image, but by several. Therefore, a large amount of memory is needed to store the dataset.

Also, the use of a GPU was preferred in order to speed up the training of the models. However, due to the limitations of the GPU's that are often found in personal computers, for this project we have used the Google Colab environment.

3.1.1 Google Colab

Google Colab is a free hosted Jupyter notebook service that requires no previous setup. For this project we have used Google Colab Pro, as it not only provides faster GPU's but also longer run-times and more memory. The specifications are the following:

- CPU: Intel(R) Xeon(R) CPU @ 2.20GHz
- GPU: Tesla P100-PCIE-16GB
- RAM: 27.3GB

3.1.2 Keras and Tensorflow 2

Throughout the project we have also used Keras 2.4.0 and Tensorflow 2.4.1.. Keras is an open source Deep Learning API written in Python that makes it easy to prototype deep learning models. Keras runs on top of Tensorflow, an end-to-end open source platform for machine learning.

3.2 Experiments

This section will present and discuss the experiments that have been performed. Said experiments have been implemented with different datasets and each dataset has been introduced so that the complexity of the problem was increased. The objective was to use what was learnt from a dataset and apply it to the next while adapting to the added difficulties.

The three datasets that we have used are:

- MNIST [13]: A database of handwritten digits. It consists of 60000 training examples and 10000 test examples. Each example is a 28×28 black and white image with a label ranging from 0 to 9 for the digit written in the image.
- Fashion-MNIST [25]: A dataset of Zalando's article images. It consists of 60000 training examples and 10000 test examples. Each examples is a 28×28 greyscale image, associated with a label from 10 classes. Every class is a different clothing piece.
- Cifar10 [12]: A dataset that consists of 50000 training images and 10000 test images. Each examples is a 32×32 colour image, associated with a label from 10 classes.

(a) MNIST example (b) Fashion-MNIST example (c) Cifar10 example

Figure 3.1: Examples of images of every dataset.

The first experiments were performed with the MNIST dataset. Then we started working with the Fashion-MNIST dataset, which shares the same image size and number of classes with the MNIST database, but has classes that are more difficult to distinguish. The final experiments were performed with the Cifar10 dataset, which has changes in size and has coloured images, which means three channels per example. Also, the Cifar-10 dataset has another added complexity when compared to working with the MNIST and Fashion-MNIST datasets. This is that the latter consist of images where the object represented in it is always in the center surrounded by a black background, whereas the Cifar10 images

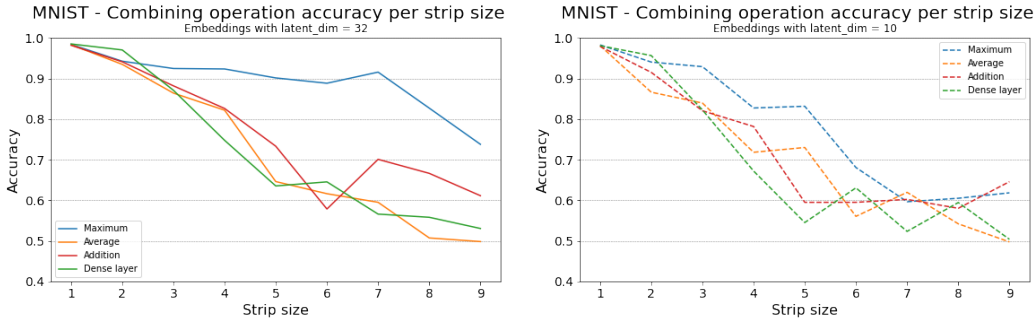
are internally more complex, they have a background which can be completely different for examples with the same label, and the main object is not always found in the center or with the same orientation.

All of the models have been trained with the Adam [11] optimizer, and the loss function used in all of the trainings is the binary crossentropy, given that we are working on a binary classification problem.

3.2.1 MNIST

Testing operations to obtain the contextual embedding

The model used for this experiment was built using the encoder block shown in Table 2.1 and with the classifier block shown in Table 2.4. The aim was to test for the best operation to combine the embeddings obtained from the context images in order to best maintain all the necessary information of these images in the contextual embedding.



(a) *latent_dim* = 32

(b) *latent_dim* = 10

Figure 3.2: Plots of the results obtained with MNIST when testing four different combining operations.

Figure 3.2 shows the results for two different *latent_dim* values, 32 and 10. Recall, *latent_dim* refers to the dimension of the embedding that is obtained from the encoder block. We can see that in both cases the maximum operation is the one with the better performance throughout all of the strip sizes. We can also see that the model built with the maximum operation and *latent_dim* 32 maintains its accuracy mostly above 0.8 for all strip sizes, whereas, the model with this same operation and *latent_dim* 10 drops its performance once it reaches a strip size larger than 5. We can see how this is due to the model not properly identifying the different classes.

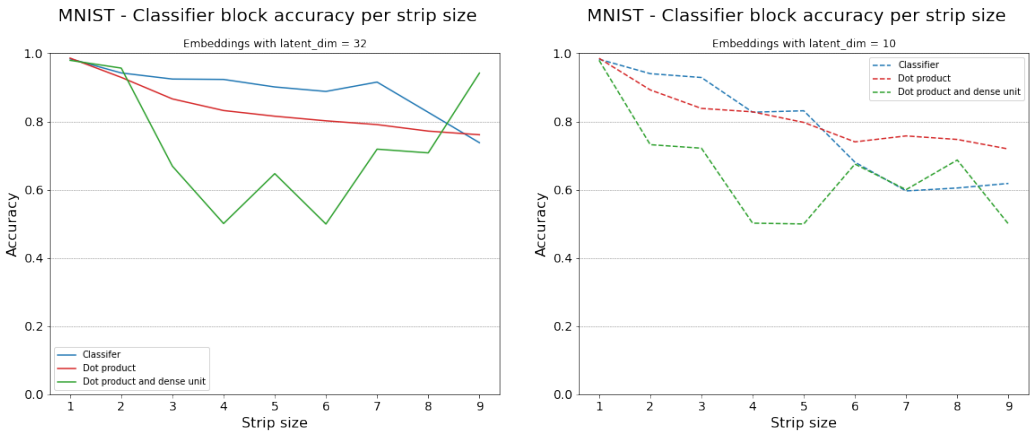
Figures 3.4 and 3.5 show visualizations of the clusters obtained from the UMAP dimension reduction technique [17] using the embeddings of the testing images of the MNIST dataset obtained from the encoder block of the two models built with the maximum operation. The clusters shown in Figure 3.4 are clearly separated for all strip sizes except

for when it is 9, which we have already seen in the accuracy plot is the only strip size for which the accuracy drops below 0.8 when the *latent_dim* value is set to 32. Figure 3.5, shows the same visualizations but created with the model built with the maximum operation and the *latent_dim* set to 10. As we can see, in this case the clusters are separated until the accuracy starts to drop, which is when the strip size reaches 6. From there, as the strip size continues the increase, the clusters keep getting closer together until there is no way to distinguish them and meaning that it is no longer possible to recognize the class of an image from its embedding.

Given these results, in the following experiments, whenever the model follows the structure shown in Figure 2.3b, we will be using the maximum operation to obtain the contextual embedding.

Changing the classifier block

Once the maximum operation was chosen we wanted to try out different classifier blocks. In Figure 3.3 we can see a comparison of the performances of three different models. All three models use the convolutional encoder from Table 2.1 and the maximum operation. However, their classifier blocks differ: one has the dense classifier, another has the dot product and the last has the dot product followed by a dense layer with one unit. These three different approaches for the classifier block were discussed in Section 2.5.

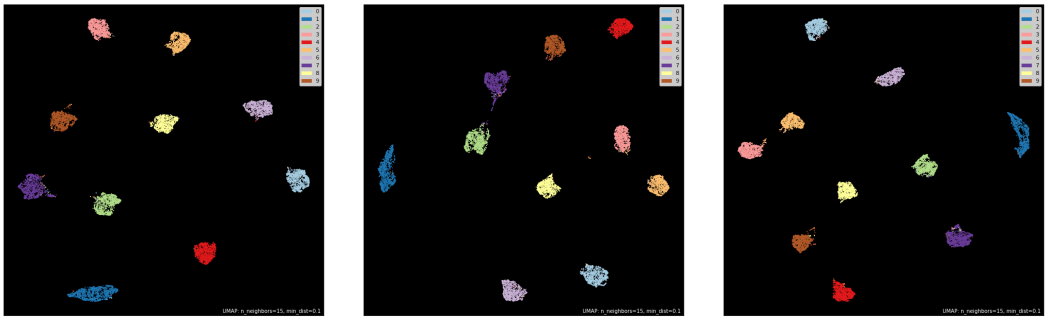


(a) *latent_dim* = 32 (b) *latent_dim* = 10

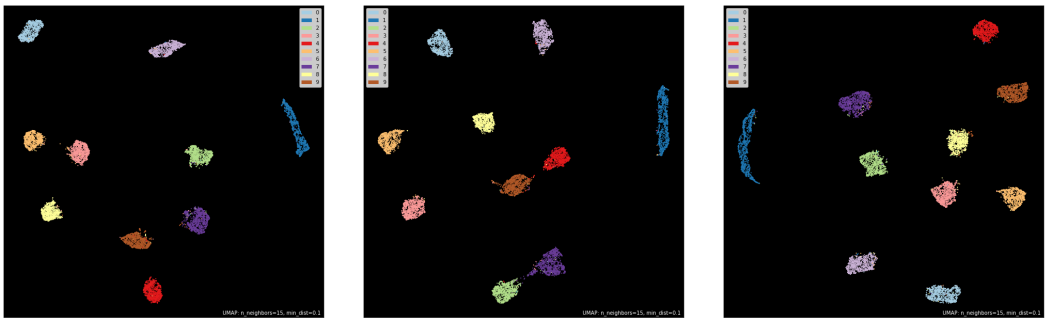
Figure 3.3: Plots of the results obtained with MNIST, when testing different classifier blocks.

In Figure 3.3 we can see two plots of the accuracy of every model for every possible strip size with two different *latent_dim* values: 32 (Figure 3.3a) and 10 (Figure 3.3b). In both cases, the model whose classifier block consisted of a dot product layer followed by a dense layer of one unit had the poorest performance. Nonetheless, we notice that when working with embeddings of size 32, the dense classifier outperforms the dot product

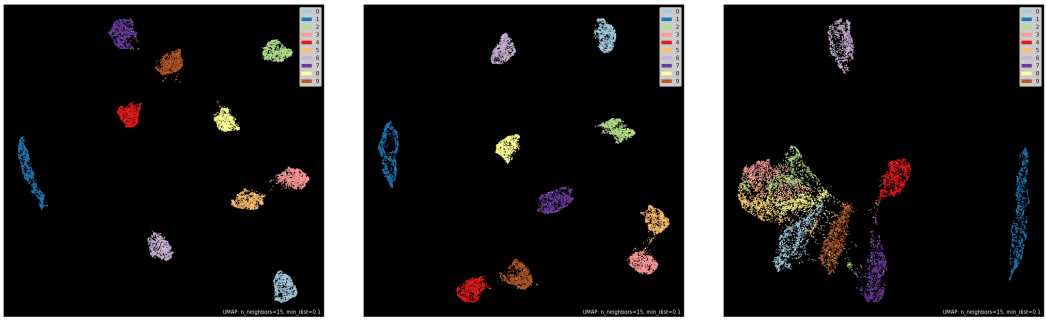
layer, whereas if the embeddings are of size 10, then the model with the dot product classifier block seems to perform better.



(a) strip size = 1 (b) strip size = 2 (c) strip size = 3

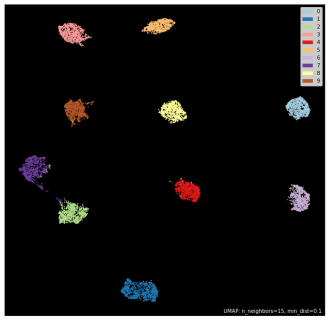


(d) strip size = 4 (e) strip size = 5 (f) strip size = 6

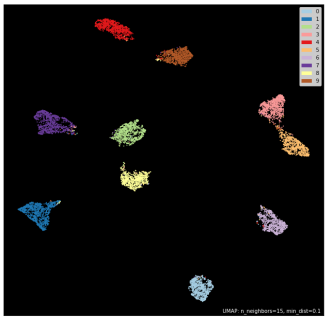


(g) strip size = 7 (h) strip size = 8 (i) strip size = 9

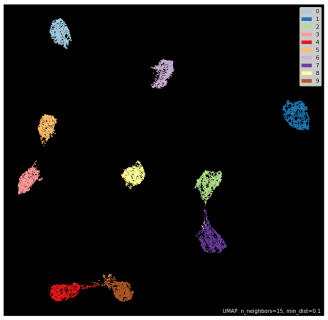
Figure 3.4: Visualizations of the clusters obtained from the encoder block using the UMAP dimension reduction technique. The model was built with the maximum operation for the contextual embedding and the *latent_dim* set to 32.



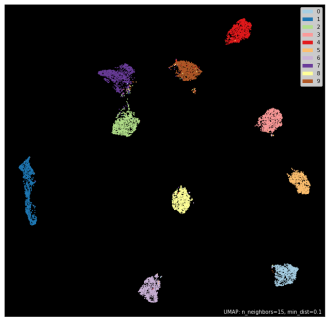
(a) strip size = 1



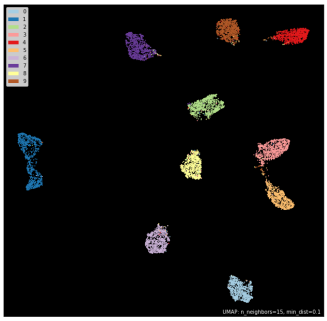
(b) strip size = 2



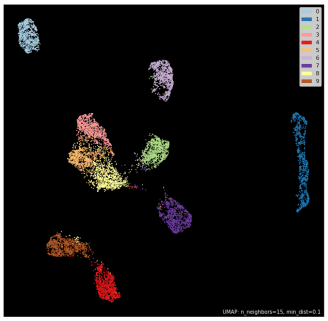
(c) strip size = 3



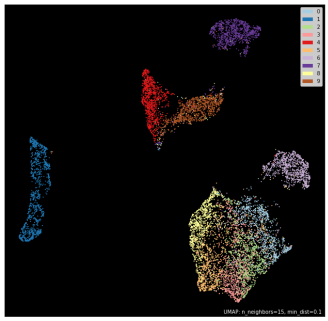
(d) strip size = 4



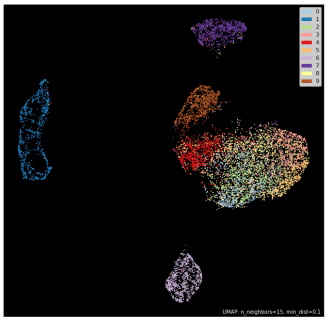
(e) strip size = 5



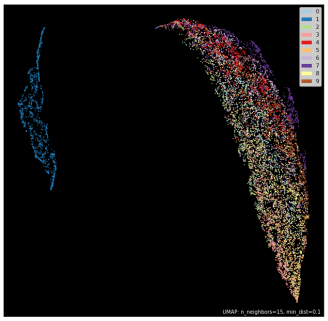
(f) strip size = 6



(g) strip size = 7



(h) strip size = 8



(i) strip size = 9

Figure 3.5: Visualizations of the clusters obtained from the encoder block using the UMAP dimension reduction technique. The model was built with the maximum operation for the contextual embedding and the *latent_dim* set to 10.

3.2.2 Fashion MNIST

As we mentioned before, the Fashion MNIST database is very similar to the MNIST dataset. The main difference, and the one that increases the difficulty when working with this dataset, is that whilst the difference between digits is quite easy to see, the Fashion

MNIST dataset contains classes that are very similar, and so encoding these classes poses a greater challenge.

Testing the classifier block

To see if the results upheld in spite of working with a different dataset, we ran the same exact models again but with the Fashion-MNIST dataset. The results are shown in Figure 3.6.

As we can see, complicating the dataset did indeed affect the performance of the models. The accuracy for all four models decreases faster as the strip size increases.

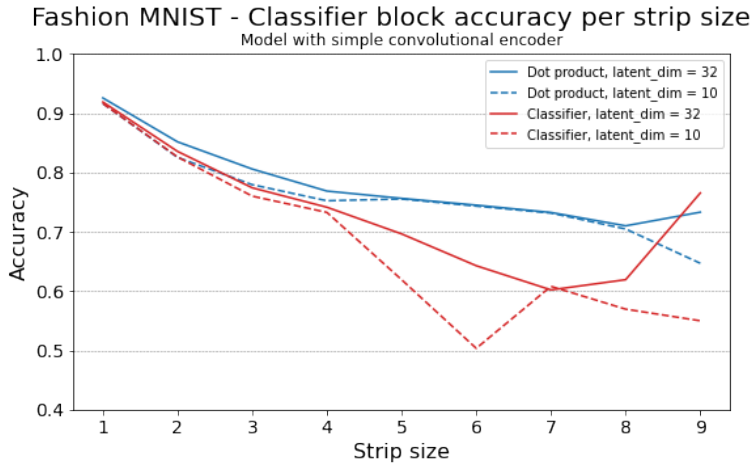


Figure 3.6: Plot of the accuracy per strip size for different classifier blocks with the Fashion-MNIST dataset with the convolutional encoder from Table 2.1.

Seeing how all the models were affected we decided to test the same classifier blocks, but with a different encoder, to see if by doing this we could achieve at least as good a performance as with the MNIST dataset.

Figure 3.7 shows the accuracy per strip size of the four different models that test two different classifier blocks, for two different *latent_dim* values, with the encoder shown in Table 2.2. Notice that the model with the best performance is still the one built using the dense classifier and with *latent_dim* 32. However, its equivalent with *latent_dim* 10 does also have a very high accuracy for all strip sizes and their performances are still very close. Even though the models built with the dot product layer do not perform as well as the other two models, they still have an accuracy of the order of 0.8 for all strip sizes.

3.2.3 CIFAR-10

The experiments performed with the CIFAR10 dataset are the first to consider RGB images. Also, given the added difficulties of this dataset, the first models built for it use

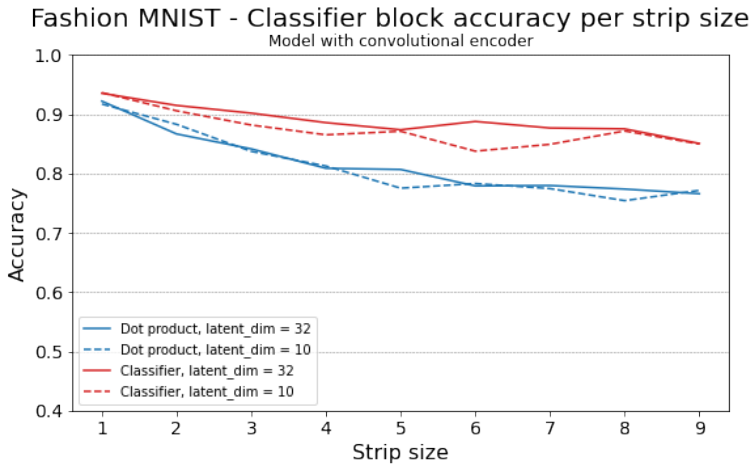
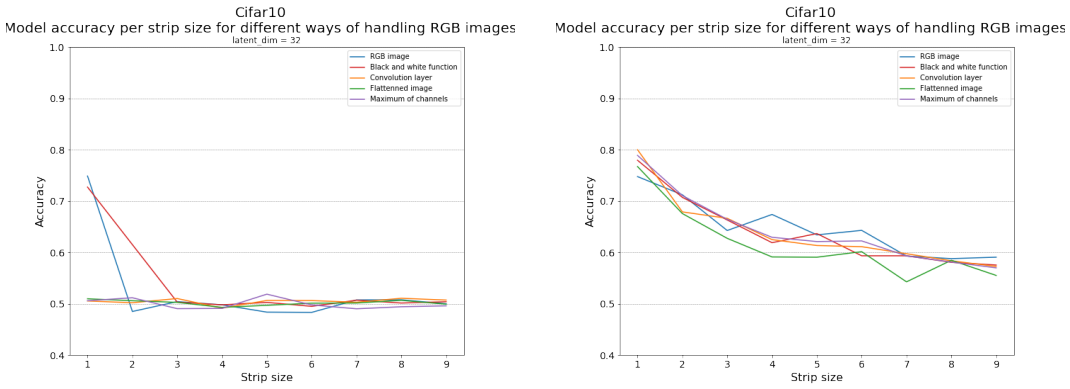


Figure 3.7: Plot of the accuracy per strip size for different classifier blocks with the Fashion-MNIST dataset.

the convolutional encoder from Table 2.3.

Testing the classifier block with different RGB to one channelled images techniques

This section shows the results for the different RGB to grey-scaled images techniques that were explained in Section 2.6. In Figure 3.8 there are two plots, each showing the accuracy per strip size for all of these techniques. The first plot (Figure 3.8a), shows the



(a) Model built with the dense classifier.

(b) Model built with the dot product.

Figure 3.8: Plot of the accuracy per strip size for different RGB to grey-scaled images techniques on models built with the third convolutional encoder.

results of models built using the dense classifier for the classifier block. As we can see, the results obtained are not good at all. The models do not learn, they all have an accuracy of around 0.5 for all strip sizes, which is the same as classifying the examples randomly.

The second plot (Figure 3.8b), shows the results of these same techniques but with a model that uses the dot product layer. Contrary to the models shown in the first plot, these models do learn. However, the results are not as good as would be expected. The accuracy of all the models decreases quickly to 0.6.

Seeing how not only did no technique work, but even when the images were fed to the model without being transformed into images of one channel, the models did not have a good performance, we concluded the Cifar10 dataset needed a different encoder, capable of capturing the complex patterns of its images. Therefore, we completed tests in which the EfficientnetB0 was used as the encoder. In this case, as the EfficientNet already requires that the images it is trained with be of three channels, we did not change them to grey-scaled images.

We can see the accuracy of these models in Figure 3.9. As we can see, the results obtained are still poor. The models built with the dot product layer have a high accuracy when the strip size is 1, but it rapidly decreases towards an accuracy of 0.5 as the strip size gets larger. This plot shows how, once again, the models with the dense classifier do not learn and perform as well as a random classifier would be expected to perform, indicating that for more complex datasets, it is probably better to use the dot product in the classifier block.

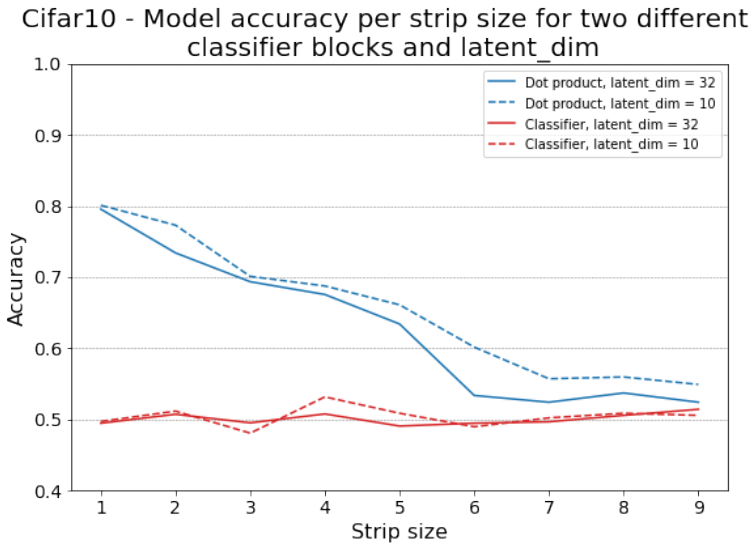


Figure 3.9: Plot of the accuracy per strip size, obtained with Cifar10, of several models with different classifier blocks and the EfficientNet architecture as encoder.

RNN classifier

Changing the approach to the problem, we built a model with a recurrent classifier (Section 2.5.3). In Figure 3.10 we can see how this model has a good performance, as its accuracy does not drop below 0.75 for any strip size. This results are obtained with two different values of *latent_dim*: 32 and 10.

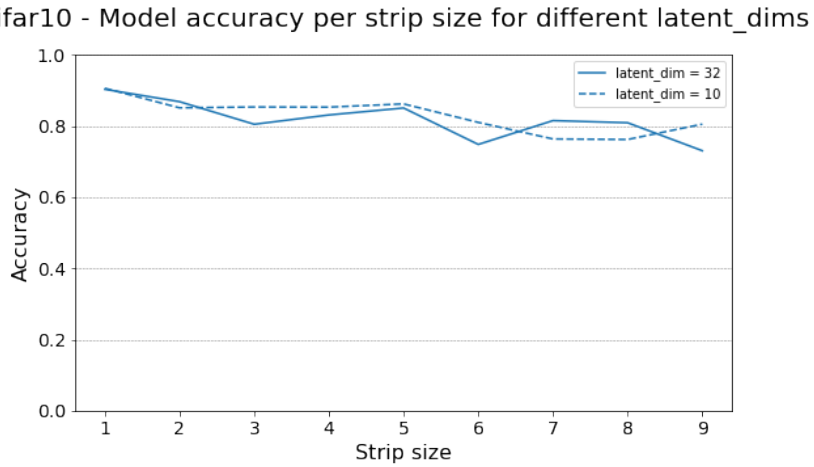
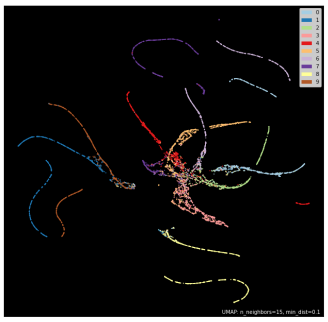
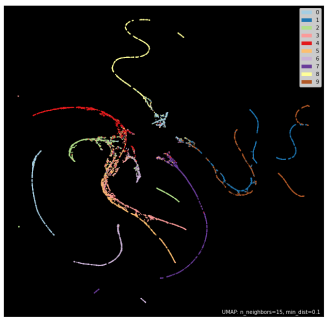


Figure 3.10: Plot of the accuracy per strip size, obtained with Cifar10, where both models are built with the EfficientNeB0 for the encoder and a recurrent network architecture for the classifier.

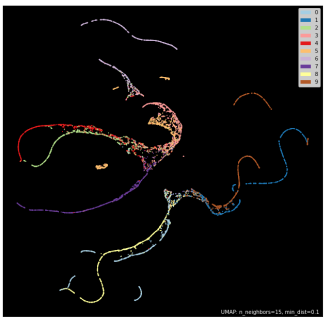
Given the good outcome of these models, we created the same visualizations as we did with the MNIST dataset, using the UMAP technique. The results are shown in Figure 3.11 for the model with *latent_dim* 32, and in Figure 3.12 for the model with *latent_dim* 10. Notice that, whereas for the MNIST dataset we obtained clear and rounded clusters when the models had a high accuracy, with the Cifar10 dataset the clusters we obtain have a thin spirally shape. This is due to the way UMAP tries to create the clusters with the embeddings it is given. Nonetheless, we can see how the curvy lines mostly contain dots of only one color. Meaning that, although the shapes of the clusters are not clear, they have been clearly distinguished.



(a) strip size = 1

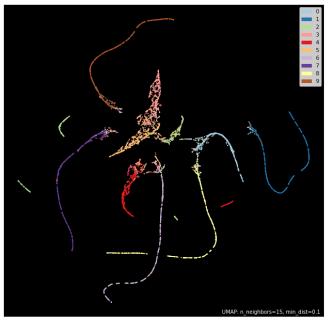


(b) strip size = 2

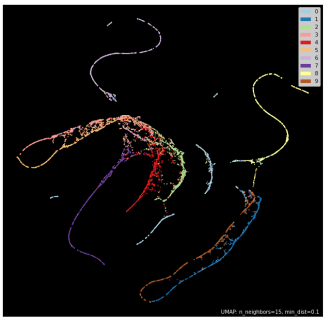


(c) strip size = 3

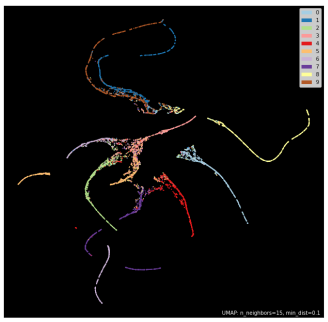
(d) strip size = 4



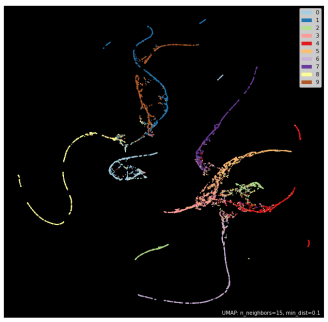
(e) strip size = 5



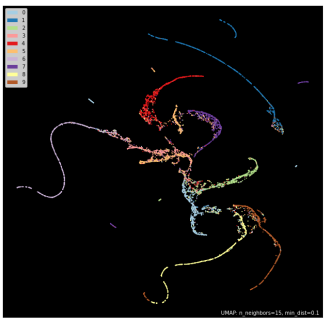
(f) strip size = 6



(g) strip size = 7

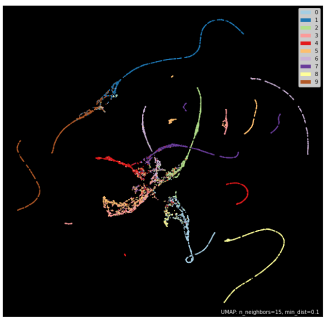


(h) strip size = 8

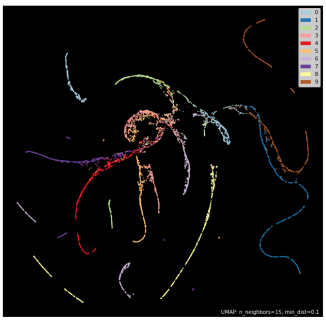


(i) strip size = 9

Figure 3.11: Visualizations of the clusters obtained from the encoder block using the UMAP dimension reduction technique. The model was built with the RNN classifier and the *latent_dim* set to 32.

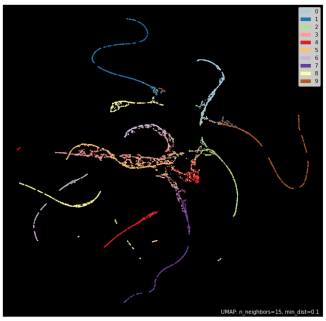


(a) strip size = 1

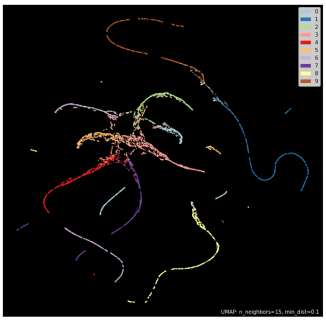


(b) strip size = 2

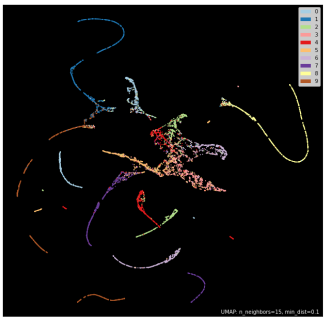
(c) strip size = 3



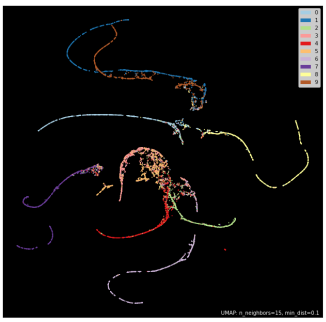
(d) strip size = 4



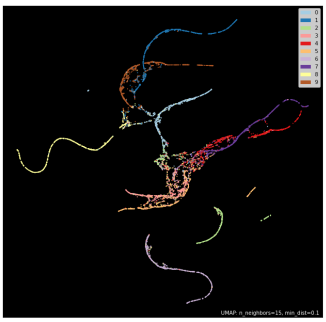
(e) strip size = 5



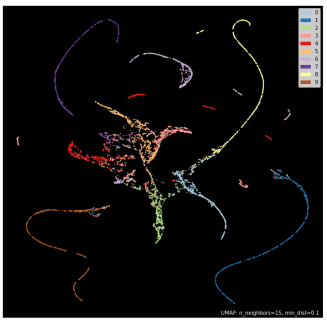
(f) strip size = 6



(g) strip size = 7



(h) strip size = 8



(i) strip size = 9

Figure 3.12: Visualizations of the clusters obtained from the encoder block using the UMAP dimension reduction technique. The model was built with the RNN classifier and the *latent_dim* set to 10.

Conclusions and further work

At the beginning of this thesis we established some goals we were aiming to reach by its completion. These goals included acquiring the theoretical background of deep learning models by working with different model architectures such as convolutional networks and recurrent networks, in order to test different methodologies for context learning. We were expecting to test these models by working with simple datasets, instead of medical images, in order to test if the methodologies we proposed could indeed work with more complex medical datasets.

Throughout the last seven months we have been working towards the completion of this project and the fulfillment of these goals. Firstly, by gaining the foundations of the theoretical knowledge of neural networks and model training, which was previously exposed in Chapter 1. Then, in Chapter 2 we presented the problem we were trying to solve with this thesis and discussed the several approaches with which we have attempted to do it.

Next, we performed several experiments with the proposed models and a variety of datasets. Starting with the MNIST database and gradually increasing the complexity of the dataset the models were trained with by moving onto the Fashion-MNIST dataset and then the Cifar10.

From the models we have proposed, we can distinguish two different architectures. The first, a convolutional network architecture, and the second a recurrent neural network architecture. Given the experiments and results presented in Chapter 3, we can confirm that the convolutional approach was successful with the simpler datasets, providing great results, whilst being deficient with the more complex dataset, Cifar10. However, the results obtained with this dataset were greatly improved when using a recurrent architecture, indicating the superiority of this approach when working with more complex datasets. Furthermore, the way we have structured our models into different blocks, allows us to evaluate not only the precision in its predictions, but whether these are being made by a correct identification of the different classes.

Notwithstanding the above, several improvements can be made in order to improve the models we have proposed. The first would be to perform a more thorough optimization of hyperparameters. Given that many models were tested, and each of them were executed for several strip-sizes, hyperparameter tuning was not our main focus. In addition, we

could also try to include same data augmentation into our training as it would help to avoid overfitting, which sometimes occurs. More improvements should be made on an attempt to increase the stability of the models, since on several occasions they must be trained twice to achieve a good performance.

This thesis attempts to set a basis for further investigation on the context learning problem. It has shown that the proposed methodology works on several datasets that are often used for benchmarking deep learning models. From here on, further experiments can be performed on more datasets with the objective of using this methodology in the medical field.

Bibliography

[1] A. Arora. EfficientNet: Rethinking model scaling for Convolutional Neural Networks, Aug 2020. https://amaarora.github.io/2020/08/13/efficientnet.html.

[2] A. Arora. Squeeze and excitation networks explained with pytorch implementation, Jul 2020. https://amaarora.github.io/2020/07/24/SeNet.html.

[3] M. Bansal, A. Krizhevsky, and A. S. Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. *ArXiv*, **abs/1812.03079**, 2019. https://doi.org/10.15607/RSS.2019.XV.031.

[4] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, Oct. 2014. Association for Computational Linguistics. https://aclanthology.org/D14-1179.

[5] D. Cireşan, U. Meier, J. Masci, and J. Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, 2012. https://doi.org/10.1016/j.neunet.2012.02.023.

[6] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.*, **12**(7):2121–2159, 2011. https://dl.acm.org/doi/10.5555/1953048.2021068.

[7] L. Hardesty. Explained: Neural networks, Apr 2017. https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.

[8] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, **9**(8):1735–1780, 1997. https://doi.org/10.1162/neco.1997.9.8.1735.

[9] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018. https://doi.org/10.1109/CVPR.2018.00745.

[10] I. IBM Cloud Education. What are neural networks?, Aug 2020. https://www.ibm.com/cloud/learn/neural-networks?mhsrc=ibmsearch_a&mhq=neural+networks.

[11] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. http://arxiv.org/abs/1412.6980.

[12] A. Krizhevsky. Learning multiple layers of features from tiny images, 2009. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.222.9220.

[13] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. http://yann.lecun.com/exdb/mnist/.

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

[15] P. Mahajan. Fully connected vs Convolutional Neural Networks, Oct 2020. https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5.

[16] M. Matsugu, K. Mori, Y. Mitari, and Y. Kaneda. Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Netw.*, **16**(5-6):555–559, 2003. https://doi.org/10.1016/S0893-6080(03)00115-1.

[17] L. McInnes, J. Healy, N. Saul, and L. Großberger. UMAP: Uniform Manifold Approximation and Projection. *J. Open Source Softw.*, **3**(29):861, 2018. https://doi.org/10.21105/joss.00861.

[18] C. Olah. Understanding LSTM networks, Aug 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, **323**(6088):533–536, 1986. https://doi.org/10.1038/323533a0.

[20] M. Ryan. How neural networks "learn", Oct 2019. https://towardsdatascience.com/how-neural-network-learn-3b56c175b5ca.

[21] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. https://doi.org/10.1109/CVPR.2018.00474.

[22] K. Sarkar. ReLU: Not a Differentiable Function: Why used in Gradient Based Optimization? and Other Generalizations of ReLU, May 2018. https://medium.com/@kanchansarkar/relu-not-a-differentiable-function-why-used-in-gradient-based-optimization-7fef3a4cecec.

[23] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2815–2823, 2019. https://doi.org/10.1109/CVPR.2019.00293.

[24] M. Tan and Q. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019. https://proceedings.mlr.press/v97/tan19a.html.

[25] H. Xiao, K. Rasul, and R. Vollgraf. Fashion–MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, 2017. https://github.com/zalandoresearch/fashion-mnist.