UNIVERSITY OF PERUGIA
DEPARTMENT OF MATHEMATICS
AND COMPUTER SCIENCE

Capsule Networks Knowledge Extraction through Network Visualization

BACHELOR'S THESIS IN COMPUTER SCIENCE

Candidate
Antonio Strippoli

Advisors
Prof. Valentina Poggioni
Ph.D. student Alina Elena Baia

Academic Year 2019/2020

"If you want to understand a really complicated device,
like a brain, you should build one."

Geoffrey E. Hinton

Contents

Introduction

Artificial Intelligence (AI) is a very popular field which is augmenting human capacities and disrupting eons-old human activities. Most AI examples that we hear about today, from chess-playing computers to self-driving cars, rely heavily on **Artificial Neural Networks (ANNs)**, particular architectures that have been found to achieve **state-of-the-art (the best)** performance on many problems.

Specifically, **Convolutional Neural Networks (CNNs)** have been explored as a type of ANN which can reduce translational variance in the input signal. One of the oldest example is **AlexNet** [1], a CNN capable of performing well on the ImageNet data set. However, this architecture comes with its own set of drawbacks, which led to further research over the years.

At the end of 2017, a new architecture named **Capsule Network (CapsNet)** was proposed [2], addressing some of the CNNs' drawbacks. The authors replace classical notions of scalar neural computation with a **vector-based approach**, allowing CapsNet to describe input images not only by the presence of constituent features but also by the pose and transformation of detected features, thus imparting **view-point** and **pose invariance**. However, most of the recent works employ Capsule Network models as a **black box**, which is particularly troubling when the application concerns human well-being.

With this work, we want to examine how much more valuable information about Capsule Networks we can get with **network visualization techniques** to both improve the network's model and the training process. Through network visualization, we can *render and visualize what a network has learned*, understand its functioning, localize bad behaviours and intervene quicker to fix them. Specifically, we propose an analysis of well known visualization techniques, as well as **two novel techniques** that could be employed on CapsNet models. Moreover, in the case of "*Routing Path Visualization*", our studies also led to an improved version capable of rendering more interpretable images compared to the original ones.

Finally, we propose **Capsule Network Knowledge Extractor**, a software to easily **produce network visualizations** of a Capsule Network created with **Keras under TensorFlow 2.0**. By providing an easy to use software for CapsNet visualization, we hope to encourage the use of visualization techniques.

Thesis Organization

*Chapter **1*** will provide the background to fully understand the thesis: some Machine Learning concepts will be explained, moving then to the world of Deep Learning, analyzing the general structure of Deep Neural Networks with a final zoom to Convolutional Neural Networks and Capsule Networks.

*Chapter **2*** provides a detailed explanation of the visualization techniques employed and the software created.

*Chapter **3*** analyze the outputs of our software on some Capsule Networks.

Lastly, in *chapter **4*** the conclusion are drawn and cues for further investigations are discussed.

1 Background

This chapter shall provide an high-level overview of the theoretical background.

1.1 Machine Learning Concepts

Machine Learning (ML) is a sub-domain of Artificial Intelligence (AI) that allows us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings. A Machine Learning algorithm is *capable of learning* how to process a task based on the data it analyzes. What has been learned by the algorithm is formally called **model**, consisting in rules, numbers, data structures and procedures that will be used to process the task.

ML is so pervasive today that we probably use it dozens of times a day without knowing it. However, it is surprising to know that the fundamental ML algorithms that base today most used applications have been around *for ages*. [3]

In this chapter, we will define formally a learning algorithm, describing its features, most common behaviours and how it could theoretically be implemented.

1.1.1 Learning Algorithms

Tom Mitchell [4] defines *learning* as follows:

> *"Formally, a machine is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P** if its performance at tasks in **T**, as measured by **P**, improves with experience **E**."*

For example, a computer program that learns to play *chess* might improve its performance as measured by its *ability to win* (**P**), at the class of tasks involving *playing chess games* (**T**) through experience obtained by *playing games* (**E**).

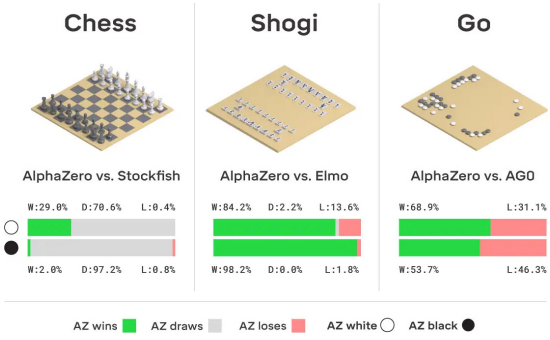


Figure 1.1: *Great performances obtained by AlphaZero, an algorithm based on Machine Learning capable of playing several games, including chess. [5]*

The inputs of a model are formally called **examples**. Examples are represented as a collection of **features** quantitatively or qualitatively measured from an entity or event. We generally represent an example as a vector

$$x = (x_1, \ldots, x_n)$$

where each entry $x_i \in D_i$ and D_i is the domain where the feature values can vary. A feature could be expressed as a number, a string or any other type wanted (e.g. the features of an image are usually the values of the pixels). Examples are usually collected in what are called **data sets**.

1.1.2 Training and Evaluation

In a ML algorithm we can generally identify two phases:

- **Training**: the algorithm improves from experiences, refining the model based on the inputs it receives;
- **Evaluation**: the performance of the algorithm is measured and it is determined whether it is good enough or it is preferable to another one.

It is common to split all the examples available in two different data sets: the **training set** and the **test set**. We can use the training set to define a model, calculating some error measure on the outputs (which we will call **training error**) to iteratively improve it, increasing the accuracy on the training set. On the other side, we reserve the test set as a meter of evaluation, since we are interested in how well the algorithm performs on data that it has never seen before. In the same way as for the training set, we can calculate a **test error** on the test set, also called **generalization error**.

The training is a crucial phase and it has a huge impact on the final performances of the model. During the years, three main types were distinguished:

- **Supervised Learning**: the algorithm is provided with examples, and every example has a **target**, which is the answer expected. So, every input is in the form $(x, y) \in X \times Y$, where X are the examples and Y the labels. With supervised learning, we want the algorithm to learn a function that best approximates the relationship between input-output pairs;
- **Unsupervised Learning**: the algorithm is provided with examples, but this time no target is provided. So, every input is in the form $x \in X$. With unsupervised learning, we want to infer the natural structure present in the data. Depending on the task, we could learn the entire probability distribution that generated a data set;
- **Reinforcement Learning**: the algorithm interacts with an environment and do not simply experience a fixed dataset, so there is a feedback loop between the learning algorithm and its experiences. With reinforcement learning, we want the algorithms to be goal-oriented, training them on how to attain a complex objective.

To control and alter the behaviour of the training process, a ML algorithm usually provides a particular type of parameters as part of its configuration, called **hyperparameters**. Hyperparameters are usually **tuned by humans** through a sequence of trial and errors, since there is *no analytical formula* available to calculate an appropriate value for them. We could however design a nested learning procedure where one learning algorithm learn the best hyperparameters for another learning algorithm.

We could provide tons of hyperparameters, and they strictly depend on the algorithm we want to use. For example, about Figure 1.2 we could provide an hyperparameter to set the degree of the polynomial function used for the red curve.

Based on the task, we can specify different performance measures to be used during the evaluation phase. For supervised learning, the **accuracy** of the model is measured and analyzed, which is the proportion of examples that have been *well classified*. As the opposite, we can define the **error rate** as the proportion of examples *misclassified*.

1.1.3 Classification

There are many kinds of tasks where ML algorithms find application. One of the most common task is named **classification**, where the computer program is asked to predict which of given *k classes* an input belongs to. More technically, the algorithm is asked to identify (or at least approximate) a mapping function f defined as:

$$f : D_1 \times \ldots \times D_n \rightarrow \{l_1, \ldots, l_K\}$$

where $\{l_1, \ldots, l_K\}$ is a finite set of *labels* identifying the given k classes. In other variants of the classification task, the output is returned as a *probability distribution* over the classes, e.g. in *Artificial Neural Networks*, from which we can infer a single class as the most probable one.

The learning process is done with the *supervised learning* approach and there are two types of learners:

- **Lazy Learners**: the training data are stored and the classification is performed by categorizing the incoming data in the same way as the most related data previously stored. An example of lazy learner is the *k-nearest neighbor algorithm*;
- **Eager Learners**: an effective classification model is determined through a training process. Compared to lazy learners, they take more time in the training process, but less for a prediction. Examples of eager learner are the *Artificial Neural Networks*.

A well-known classification task is to classify image of *handwritten digits* to their respective **label**, which we will analyze in details in the following chapters.

1.1.4 Overfitting, Underfitting, Generalization

We could think of letting a Machine Learning algorithm train until it gets a perfect accuracy. While this could sometime be a reasonable strategy, it could lead to some difficulties when there is some noise in the data or we did not provide enough examples to the algorithm. On the other hand, we do not want to train the algorithm poorly, or it will get a high error rate.

In both cases we will obtain poor performances. That's why those two situations are the central issues of Machine Learning:

- The first one is called **overfitting**: the model has assimilated concepts that do not apply well to new unseen data, negatively impacting its ability to generalize. A model that is overfitting has a very low training error, but an higher and unbearable test error;
- The second one is called **underfitting**: the model is not able to obtain a sufficiently low error value on the training set, nor on the test set: it has not learned enough. Underfitting is not so much discussed, as it is easy to detect given a good performance metric. Even so, the remedy is not always that easy: we could even need to move on a different ML algorithm.

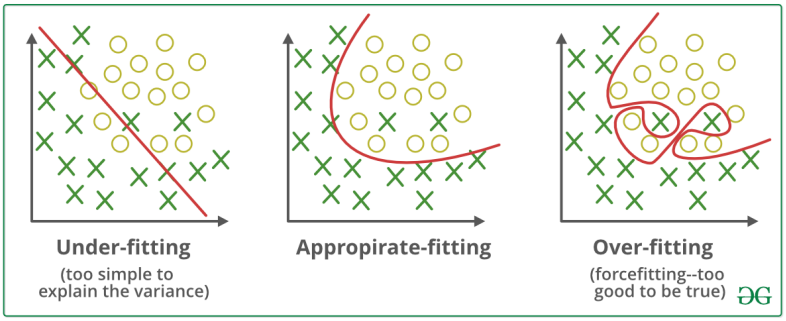


Figure 1.2: *Visualization of an example of a classification task, considering the red curve as the function to be learned. An appropriate fitting is needed to make the function generalize well for future examples provided. [6]*

We can say that the goal of a good Machine Learning algorithm is to perform well on new, previously unseen inputs, not just those on which our model was trained on. This ability is called **generalization**.

1.1.5 Gradient Descent

Optimization is a big part of Machine Learning. Almost every learning algorithm has an optimization algorithm at its core. With optimization, we mean either minimizing or maximizing some function $J(\theta)$ by altering θ. In ML, the function $J(\theta)$ is called **objective function** and tells us "how good" our model is at making predictions for a given set of parameters. However, we will also refer to $J(\theta)$ as **cost function**, **loss function** or **error function** if we are *minimizing* it (from now on, we will always refer to objective function as loss function).

The **loss** is another very important measure for learning algorithms.

Now, how can we find a $\hat{\theta}$ that minimizes J? That is where **gradient descent** comes in. Gradient descent is an optimization algorithm used to find the values that minimizes or maximizes a function J.

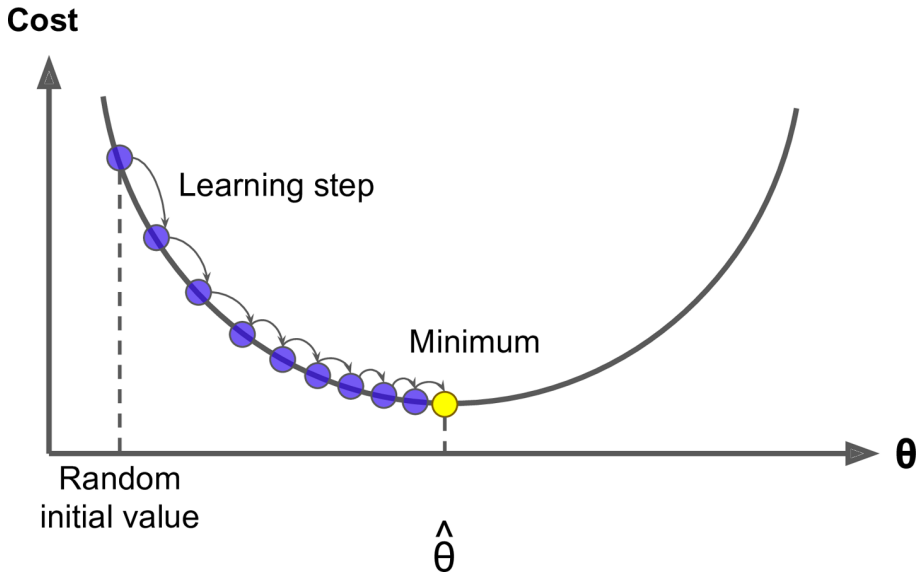


Figure 1.3: *Visualization of the finding of a simple function's minimum using gradient descent. Initially, θ is initialized with a random value; then, we begin to go towards the minimum using gradient descent, until we find the minimum (or a value very close). [7]*

Our loss function J will have its own curve and its own gradients. The *slope* of this curve tells us **how to update** our parameters θ to make the model more accurate. Gradient descent technique calculates the partial derivatives w.r.t. each parameter and stores the results in a **gradient** ∇_θ. It then **updates the parameters** in the *opposite direction* of the gradient: $\theta = \theta - \eta \nabla_\theta J(\theta)$ where η is called **learning rate** and determines the size of the steps we take to reach a (local) minimum. Iterating over this procedure for enough steps, will give us a good approximation of the minimum of J.

We will not get into mathematical details on how Gradient Descent works, since it would be outside the scope of this thesis.

1.2 Deep Learning Concepts

There are several strategies that were proposed to implement learning algorithms for Machine Learning. A sub-category of Machine Learning is called **Deep Learning**, where the term "Deep" comes from how the hierarchy of learned concepts is treated: in fact, if we try to visualize how the concepts are built on top of each other as a *graph*, we will obtain a *deep graph* with many layers.

In this section, we will cover the basic elements needed to make a working Artificial Neural Network, analyzing a general architecture and a training method to make it learn.

1.2.1 Artificial Neural Network

An **Artificial Neural Network (ANN)** is a ML algorithm based on a *logical network* mimicking the functioning of the *biological brain*. Just like our brain, Artificial Neural Networks are composed of **neurons**, interconnected to each other and organized in **layers**:

- The first layer is called **input layer** and it receives an *input vector* containing 1 or more examples;
- The last layer is called **output layer** and it returns an *output vector* as the final result of the computation;
- The layers in between the input and output layers are called **hidden layers**.

However, a very simple network could be composed of only an input layer and an output layer. When there is at least one hidden layer, we refer to those networks as **Multilayer Networks**.

The connection between two units is called **link** and propagates the **activation** between two units. All the links that connect a single neuron to multiple neurons of a previous layer are called **receptive fields**. Networks with *directed links* and *no cycles* are called **Feedforward Networks** and are defined as acyclic directed graphs.

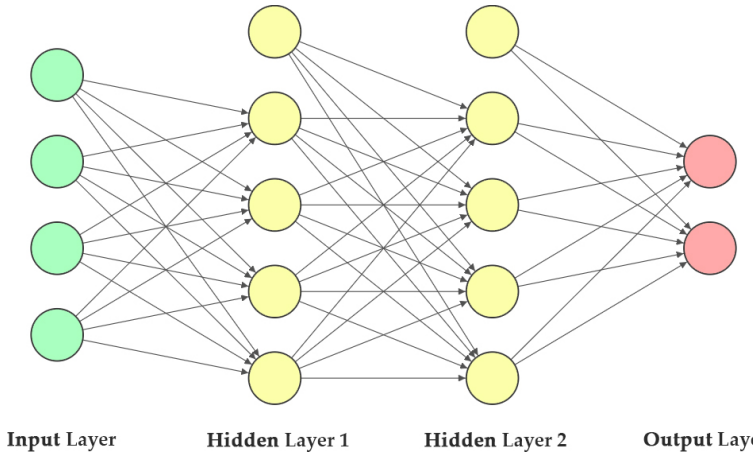


Figure 1.4: *Graph representation of a simple Feedforward Artificial Neural Network with 2 hidden layers.*

In Feedforward Networks, the input layer provides the initial information, that propagates up to the hidden units of each hidden layer, finally reaching the output layer and producing the output. Then, a scalar cost will be produced ($J(\theta)$) and used for the training. This passage of information is called **forward propagation**.

For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the *most effective learning methods* currently known. Such networks have been frequently used over the last decades for several applications, like pattern recognition (image and voice recognition).

1.2.2 Neurons

As anticipated, **neurons** are the building blocks of an ANN, acting as **processing units** conceptually derived from the biological neuron. Each neuron can take *multiple inputs*, but produces a *single output*.

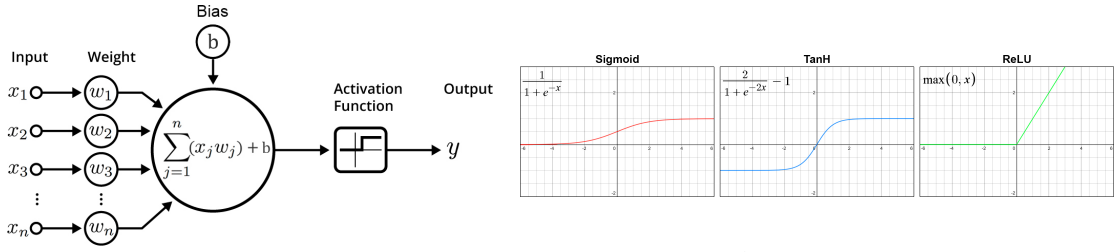
Regarding inputs, they can either be feature values of an example (if the neuron is in the first hidden layer) or outputs of other neurons. Each input has always associated a float value, which we call **weight** and will be updated as the learning phase advances. The weights are used to *increase or decrease the strength* of the incoming inputs and their **initialization** is a really delicate topic, as it could speed up the learning process.

Designing improved initialization strategies is a difficult task because ANN optimization is not yet well understood. Our understanding of how the initial point affects generalization is especially primitive, offering little to no guidance for how to select the initial point [8]. There are several initialization methods, but they all *avoid to initialize all the weights to zero*, or every neurons will learn the same features during training, resulting in low final performances. One of the most popular is the *Glorot Uniform* initializer.

About the output, it can either be transmitted to the next layer or be used as a partial result for a prediction (if the neuron is in the output layer). It is produced in 2 steps:

1. First, a **weighted sum** of the inputs is computed, weighted by the weights associated to the inputs. Generally, this weighted sum is seen as a **matrix multiplication** between the weights (transposed) and the inputs, since this way the process can be algorithmically optimized. Sometimes, neurons may also have an additional number, named **bias**, acting as a threshold for the aggregate signal. In that case, the bias is then added to the weighted sum.

2. Finally, an **activation function** is applied. Activation functions specify the importance of the neuron output. Hidden units can be distinguished from each other by the choice of the form of this activation function. Some activation functions that are very common are the **sigmoid**, **tanh** and **reLU**.



(a) *Neuron* ***(b)*** *Activations*

Figure 1.5: *A visualization of the neuron computation process and the plotting of the 3 most used activation functions.*

1.2.3 Training: Back-propagation

In section 1.1.5, we described an optimization method called **Gradient Descent**, which is used to make a ML algorithm learn. However, the deep structure of ANNs makes *more challenging* to calculate the loss function's gradient, since we need to calculate derivatives of the loss function with respect to each weights and biases of the network. In ANNs, the loss function is computed with neurons' outputs of the output layer: we make use of *supervised learning*, so targets are provided and compared with the outputs of the network, making the derivatives' computation very simple for this layer. However, for the hidden layer this calculation becomes complicated, because they strictly depends on the parameters of previous and following layers.

A solution for the calculation of the gradient in ANNs was provided in 1986 with the **back-propagation algorithm** [9] (often simply called **backprop**), which make use of the **chain rule**, a derivation rule that allows to calculate the derivative of a composed function.

So, after a forward propagation, there will be a backward propagation, where the information will flow backwards through the network and every weight and bias will be updated with the **negative** of the *partial derivative of the loss function* with respect to the parameter, multiplied by the *learning rate*.

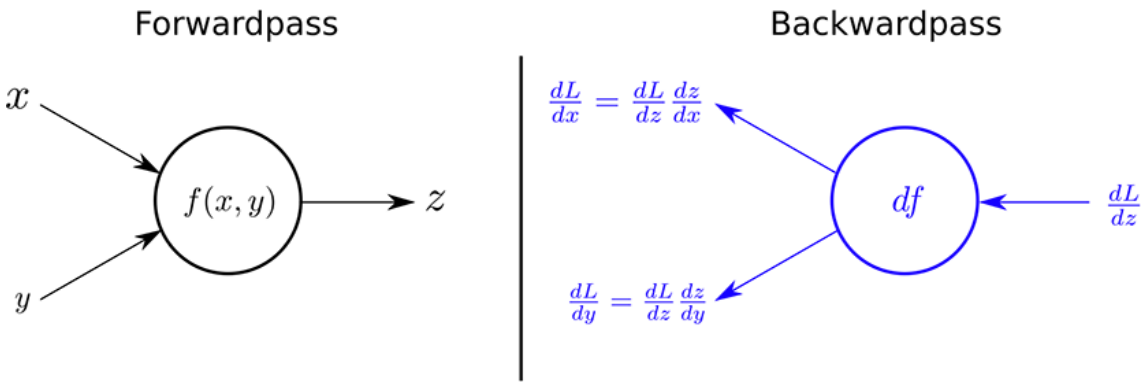


Figure 1.6: *Calculation of the gradients using the chain rule, in the backward pass of back-propagation. [10]*

During the years, the training procedure for Artificial Neural Networks was better defined and organized. We define an **epoch** as one cycle through the full training dataset and usually more than 1 epoch is needed to fully train a ANN. Also, it was seen that letting an ANN update their weights after just one example does not provide a right balance between physical memory and CPU instructions used during the training. So, **batches** were introduced, which are subsets of the training set. Using batches speeds up the overall training procedure (since we update the parameters less often) without affecting the training results, but requires more memory space. Batch size and epoch number are treated as hyperparameters.

1.3 Convolutional Neural Networks

At the moment of writing this thesis, it does not exist a generic ANN capable of solving any wanted task of any field. For the **computer vision** field[1], a particular type of network became highly popular, called **Convolutional Neural Network (CNN)**. The name indicates that the network employs the **convolution** mathematical operation, a specialized kind of linear operation.

> *"Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. [8]"*

Although it is a network specialized for the task of computer vision, it has also been proven to perform very well in other applications, such as natural language processing [11].

CNNs are inspired from biology, trying to emulate the behavior of the *visual cortex*. They introduce two new types of hidden layers: the **convolutional layer** and the **pooling layer**. A general architecture of CNN is composed of a *first part* of **feature extraction and learning**, consisting of several concatenation of convolutional layer and pooling layer, followed by a final **classification** composed of some fully connected layers.

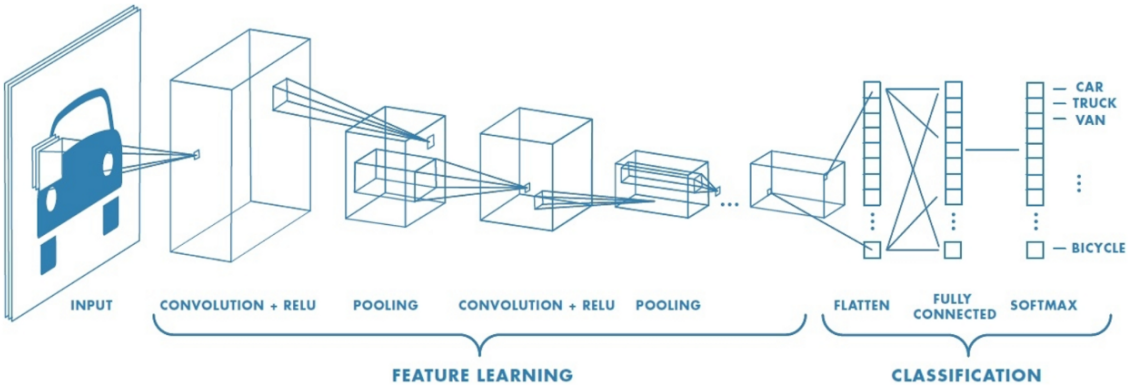


Figure 1.7: *A general architecture of a CNN for recognizing different type of vehicles from an image. [12]*

1.3.1 The Convolutional Layer

The **convolutional layers** are the most important innovation of CNNs as they are able to extract features such as *edges and color gradients*, used to distinguish one image from another (which is what we want to achieve in an image classification task).

A convolutional layer contains a new type of neuron that replace the weighted sum with the convolution. Its weights are grouped in **kernels**, organized in 3 dimensions: *width*, *height* and *channels* (which are all hyper-parameters). The neuron will just **convolve** the input, apply an **activation function** (like ReLU)

[1]Computer vision is a field that deals with making computer get **high-level understanding of images or videos**, in order to automate tasks that our human visual system could do.

and pass its result to the next layer. The convolution operation visually consists in shifting the kernel across the input, computing the element-wise multiplication between the kernel and the input, resulting in a **feature map**, whose size is determined by the kernel size and also by the **stride** hyper-parameter, which determines the number of pixels that the kernel will skip in x and y direction.

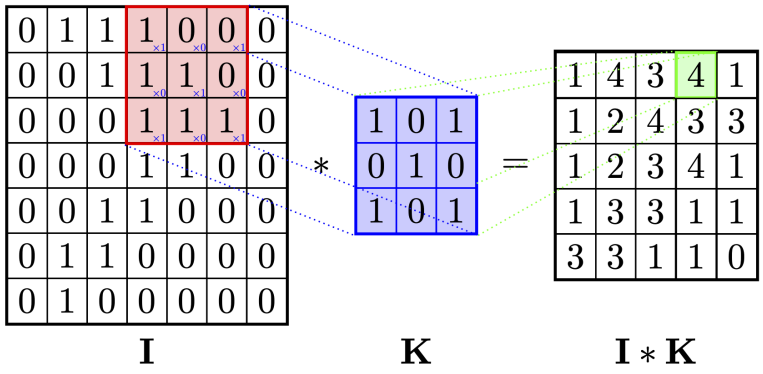


Figure 1.8: *A visual representation of the convolution operation between an input (7x7x1) and a kernel (3x3x1) using stride=1, resulting in a feature map (5x5x1). [13]*

There are several advantages provided by the convolution layer instead of the fully connected layer, including a reduced number of network's weights and number of operations (due to **parameter sharing**).

1.3.2 The Pooling Layer

Pooling layers are layers (usually put after a convolutional layer) whose role is to progressively reduce the spatial size of the representation, decreasing the amount of parameters and computation in the network. A pooling layer operates independently on every depth slice of the input and resizes it applying an **aggregate function** like *max* or *average*. Considering how it works, it does not need any weights and so it does not need to be trained.

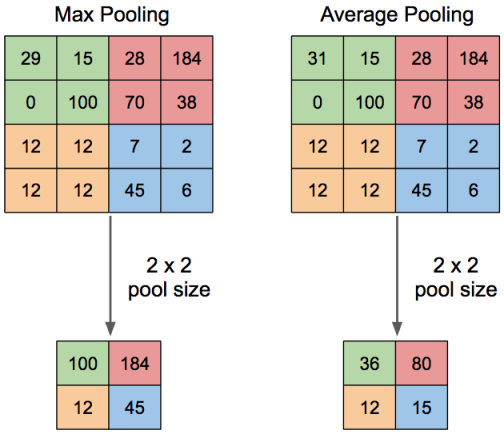


Figure 1.9: *A visual representation of the pooling operation. On the left,* ***max function*** *was applied, on the right* ***average function*** *was applied. [14]*

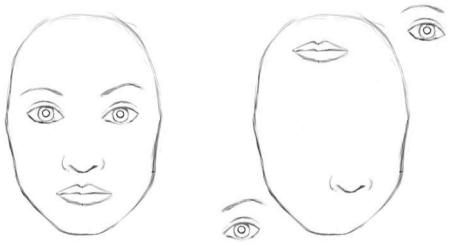
The pooling operation gives to the model **invariance** to small translations of the input, since it computes a local summary of the input, so small perturbations will not affect the output. Without a pooling layer, CNNs would only fit data which are very close to the training set.

1.3.3 Drawbacks

CNNs are great and widely used, but NN studies should not stop with them, since they are far from perfectly emulating our brain. They are capable of recognizing objects, but they have limits and fundamental drawbacks. We can identify two major problems in CNNs:

1. They are incapable of representing an object's **pose** in the space, failing in achieving **viewpoint invariance**. The *pose* is simply the rotation and and translation data of an object in the space. For example, look at Figure 1.10a. You should not have too much issues in recognizing that all the pictures are different instances of an identical statue at different orientations and conditions. That is because you can intuitively and effortlessly **manipulate the viewpoint** of one image to approximate the view of the other. For a CNN, that turns out to be very difficult without explicit training examples (which takes a lot of time and seems counter intuitive);
2. Also due to the previous point, their internal data representation does not take into account important **spatial hierarchies between simple and complex objects** inside the data. As an example, have a look at Figure 1.10b. The picture on the left is a *human face*. But the picture on the right has something wrong: there are definitely *two eyes*, a *nose* and a *mouth*, but you can easily tell that they are in the **wrong place** and this is not what a human face is supposed to look like. However, a well trained CNN do not understand this concept.

(a) *The* ***Statue of Liberty*** *viewed at different angles and conditions. You can easily recognize all the images as containing the same object. [15]*



(b) *The well known* ***Picasso problem****. You can easily tell that something is wrong with the picture on the right. [16]*

Figure 1.10: *Visualization of the 2 main issues of the CNNs.*

1.4 Capsule Networks

All the drawbacks of CNNs are mostly due to their pooling layers. We said that pooling do not only *decreases the size* of the layers (saving memory), but provides some *translation invariance* that allows the network to recognize a subject in an image regardless of where it is positioned. This may be more of a bug than a feature, as pooling operation is the cause that make CNN loose all the valuable information that we discussed before. As Hinton said [17]:

> *"The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster."*

Instead of achieving invariance, we better aim for **equivariance**. In general, an Operator is **equivariant** with respect to a Transformation when the effect of the Transformation is detectable in the Operator Output (when it is not detectable, the operator is invariant). In an ideal network, we want that changes in viewpoint to lead to corresponding changes in neural activities by achieving viewpoint equivariance, with feature property as part of the information extracted.

A new proposed ANN called **Capsule Network** [2] aims to solve all the discussed drawbacks of the CNNs.

1.4.1 Capsules

Capsules are the new building block for Capsule Networks. As the official paper [2] says about capsules:

> *"A **capsule** is a group of neurons whose activity vector represents the instantiation parameters of a specific type of entity such as an object or an object part. We use the length of the activity vector to represent the probability that the entity exists and its orientation to represent the instantiation parameters."*

The instantiation parameters that may be learned by a capsule may include the pose, texture and deformations of the entity. When working properly, the probability that the entity is present is *locally invariant*, since it is not affected by the exact position within the limited domain covered by the capsule. The instantiation parameters, however, are **equivariant**. This means that as the viewing conditions of the entity change, the instantiation parameters change by a corresponding amount since they represent these viewing conditions. [18]

To ensure that capsules' length can encode the probability of detection of an entity, we must assure that it is a scalar between 0 and 1. For that purpose, a new non-linear activation function called **squash** was produced, shrinking the short vectors to almost 0 and the long vectors to almost 1:

$$v_j = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|}$$

where v_j is the vector output of a capsule j and s_j is its total input.

1.4.2 Dynamic Routing-by-Agreement

Capsule Networks can only work with at least two layers of capsules. Transferring information from one layer to the next is a very delicate operation which required years of research. In the original paper, an algorithm called **Dynamic Routing-by-Agreement** was proposed for this task and it is meant to be the official substitute of the *pooling* operation.

The algorithm routes the outputs of lower level capsules to the **relevant capsules** in a higher level. For example, in a well trained Capsule Network, a lower-level capsule detecting an eye will be routed to the higher level capsule responsible for detecting faces, not to the one detecting legs.

The output u_i of a capsule i in layer l is forwarded to each of the capsules in the next layer. A capsule u_j in the layer $l+1$ will take u_i and multiply it with a corresponding weight W_{ij} (which will be trained with back-propagation), producing what is called **prediction** or **vote**: $\hat{u}_{j|i} = W_{ij}u_i$. It can be interpreted as the capsule i prediction of the entity's encoding of capsule j.

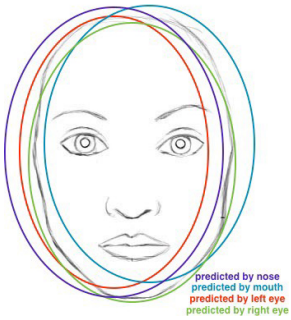


Figure 1.11: *Low level capsules prediction for an higher level capsule, where capsules' encode was simplified to represent just the location, width, height and shape. All the lower level capsules generally agree that a face is present in the input. [16]*

We then perform a **weighted sum** of all the prediction vector, using some weights named **coupling coefficient** c_{ij} determined by the routing-by-agreement algorithm, and apply the squash function. This produces an **activity vector** v_j.

At this point, we can evaluate how much capsule i and capsule j agrees by having prediction $\hat{u}_{j|i}$ and activity vector v_j: if the activity vector has close similarity with the prediction vector, we conclude that both capsules are highly related. Such similarity is measured using the scalar product between the prediction and the activity vector.

If the agreement is large, the two capsules are relevant to each other and the coupling coefficient is increased. Otherwise the capsules are not relevant to each other and the coupling coefficient is decreased. By **iteratively refining** the coupling coefficients during training the relevant capsules are linked to each other. Experiments have demonstrated that usually 3 iterations are enough. The full original algorithm is provided below [2]:

Procedure 1 Routing algorithm.

procedure ROUTING($\hat{\boldsymbol{u}}_{j|i}, r, l$)
  for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
  for r iterations **do**
    for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$ ▷ `softmax` computes Eq. 3
    for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
    for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$ ▷ `squash` computes Eq. 1
    for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
  return $\mathbf{v}_j$

1.4.3 Architecture

We already said that capsules are the fundamental building block of Capsule Networks. However, other already existing layers like the convolutional layer or fully connected layer are also employed in those networks. The general Capsule Network architecture proposed is composed of two parts: an **encoder** and a **decoder**.

In the **encoder**, basic features are extracted using a convolutional layer; then, we move to capsules' domain with a layer called **PrimaryCapsule** layer, whose task is to produce *combinations of the basic features* it receives as inputs and to encapsulate them. Primary capsules are followed by one or more capsule layers, which *route the information* with the Routing-by-Agreement algorithm. The last layer of capsules is usually called **ClassCapsule**, which *encodes the final entities* of the classification (1 capsule for each class), producing the final results.

The **decoder** will take as input the ClassCapsules and tries to *reconstruct the input image*. It is extremely useful, since it will encourage the capsules to encode the parameters for the input image, and it is done by **masking** (zeroing) all but the output vector of the correct class capsule and sending it through some **dense layers** (generally 3) of various size.

Lastly, the **loss function** for the architecture is given by the sum of two different loss: a **margin loss** and a **reconstruction loss**:

- For the encoder, we calculate the **margin loss** by summing up every partial loss L_c for each category c of class capsules:

$$L_c = T_c max(0, m^+ - ||v_c||^2) + \lambda(1 - T_c) max(0, ||v_c|| - m^-)^2$$

- For the decoder, we calculate the **reconstruction loss** by sum up the squared differences between the outputs of the logistic units and the pixel intensities of the input. Usually this loss is scaled down by a coefficient named **lambda recognition**, so that it does not dominate the margin loss during training.

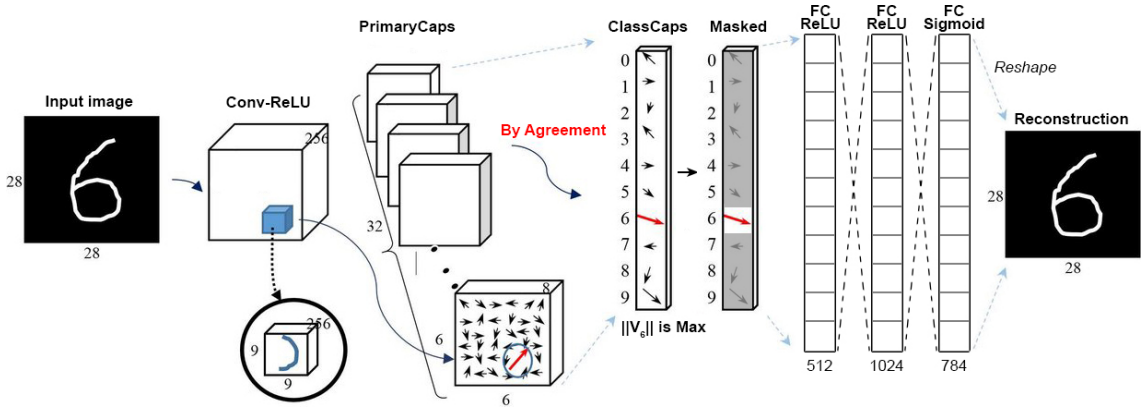


Figure 1.12: *Minimal architecture of a CapsNet for digit detection. Each arrow in the PrimaryCaps layer indicates activity vector of a capsule. The red arrow shows an activated capsule with higher magnitude for the example kernel introduced to find right curve fragment of digit 6. [2][19]*

2 Capsule Networks Knowledge Extractor

Capsule Networks could really be the next step in the ANNs field by overcoming the drawbacks of CNNs. However, the research is not mature and needs further investigation, as argued in the work "Examining the Benefits of Capsule Neural Networks" [20]. In general, measurements of performance and accuracy should not enough to evaluate Capsule Networks.

With this thesis, we want to investigate how much more valuable information we could get with **network visualization techniques** to both improve the network's model and the training process. Through network visualization, we can *render and visualize what a network has learned*, understand its functioning, localize bad behaviours and intervene quicker to fix them. Furthermore, we could *visualize how capsules process a single image* and study how they react to perturbations. That could be a starting point to make up new strategies focused on making a model more robust.

For the visualization and interpretation of the Capsule Networks, we introduce "**Capsule Networks Knowledge Extractor**". This tool has been designed with two goals in mind:

- Have a **fast, intuitive and user-friendly way** to obtain the visualizations;
- Be **scalable**, allowing to easily implement new visualization techniques.

All the code is available on **GitHub** under the *MIT License*, at the following link: https://github.com/CoffeeStraw/CapsNet-Knowledge-Extractor

2.1 Related Works

The definition of **interpretability** in Deep Learning has not yet been fully formalized. For sure, interpreting a model can make us more confident about its future results, since we should not only rely on its predictive capabilities over a fixed data set.

In Lipton's work *The Mythos of Model Interpretability* [21], there is an attempt to refine the discourse on interpretability by analyzing related literature, discovering several motivations for why interpretability is important and occasionally discordant. Lipton defines a set of properties for interpretability, but the one that concerns our work is **decomposability**, for which *each part of the model admits an intuitive explanation*. Another important point in this publication is the difference between **transparency** and **post-hoc interpretation**: if a model is transparent we can grasp *how it works*, while a post-hoc interpretation can describe the causes that led to an output *without elucidating the mechanisms* by which the model works.

While research into interpretation itself is relatively new, its impact has already been seen in applied Deep Learning contexts. A number of papers and projects that make use of Deep Learning models, **include a section on interpretation** to qualitatively evaluate and support the model's predictions and the work's claims overall. As being the heart of many ANN, Convolutional and Fully Connected layers already have a number of visualization techniques that have been employed to interpret them.

Generally, visualizing the **internal learned representation** is one approach to understand the relationship between the inputs and the outputs. About this, one of the oldest and discussed interpretation technique is **Activation Maximization** [22], which dates back to 2009. Its objective is to **synthesize an input** (e.g. an image) **that highly activates a specific neuron**. So, the expected result should represent *what a specific neuron has learned to recognize* in the input. However, it is not always an effective technique, since it is not unfrequent to obtain high-frequency images, which are uninterpretable by humans. So, **regularization methods** have been introduced to instill natural images prior, guiding the process of input synthesis. Some of these hand-designed priors that have been shown to experimentally improve quality are *Gaussian blur, total variation or jitter* [23].

Another approach is to identify **input specific regions that highly contributed** to the final results. In this regard, a recently publicized technique by the name of **Grad-CAM** has become particularly popular. Grad-CAM uses the gradients of any target concept, flowing into the final convolutional layer to produce a coarse localization map **highlighting important regions in the image** for predicting the concept [24]. The popularity of Grad-CAM made so that other researchers produced an improved version of this technique, called **Grad-CAM++** [25], which provides a better localization of objects as well as explaining occurrences of multiple objects of a class in a single image.

It did not take too long until some dedicated tools appeared to facilitate and entice users to take advantage of these techniques. A remarkable example is **tf-keras-vis** [26], whose aims are to allow **visualization and debugging** of a *Tensorflow 2.0+ Keras Neural Network model* by easily providing several techniques, including Grad-CAM and Activation Maximization. Note that this package is in constant development, which is surely a plus and makes it one of the few options up-to-date.

Moving to Capsule Networks, visualization techniques have only recently started to become subject of research. One of the oldest work dates back to 2018 and it is a tool called *CapsNet-Visualization* [27]. This tool is able to visualize the layers' activations and interactively show how manipulations on the Digit Caps values affect the decoder's output of a **fixed, pre-trained** Capsule Network.

During the course of the 2020, other two interesting works came out:

1. "**Examining the Benefits of Capsule Neural Networks**" [20] analyzes the pros and cons of Capsule Networks, also through a deep visualization analysis. Among other techniques, in this paper *Activation Maximization* was employed in an unconventional way: instead of maximizing the activation of a specific neuron, the authors proposed to **minimize the difference** between the capsule vector associated to a test image and the input that is being synthesized. This way, one should obtain synthesized images that should resemble closer to the original input image;

2. *Routing Path Visualization* [28] has been introduced to **visualize attention maps for Capsule Networks**.

At the time of writing this thesis, we miss an **in-depth discussion on visualization techniques** to interpret Capsule Networks' layers. So, we wanted to lay the foundations for future works on Capsule Networks' interpretability, focusing on post-hoc interpretations and creating a tool to easily produce visualizations for TensorFlow 2.0 Keras Capsule Network models.

2.2 Tools

Since our main goal is to produce a software which could be largely used in the future, we have chosen **Python 3.8** as the *programming language* for the project.

For the same reason, we decided to use **Tensorflow 2.2** (latest version at the moment of writing) along with its high-level API, **Keras**, as the *main Deep Learning frameworks*. For some *mathematical operations*, we had to go a bit further using **Numpy**, which however is a direct dependency of Tensorflow.

For the development of *Capsule Networks Knowledge Extractor*, we used **Flask** (along with some of its extensions) as the *WSGI web application framework*. The *rendering procedures* were written using **Pillow** and **Matplotlib**.

2.3 Interpreting the features

As previously discussed, Capsule Networks' interpretation is a *newborn* field of research. With our work, we want to contribute by *implementing and testing* **new visualization techniques**, as well as **already existing ones**. In the following sections, we will discuss in details the visualization techniques implemented in our *Capsule Networks Knowledge Extractor*.

Since we thought that a *fast execution* was a **key feature** for our software, we only employed techniques that could compute their outputs in milliseconds. However, one of the following explained technique, *activation maximization*, required several seconds to produce an interesting output. Furthermore, this technique needs further investigation to prove its usefulness for Capsule Networks, as we will explain in its dedicated section. So, we decided to not include it in our software, but to leave it in a separated experimental file.

2.3.1 Convolutional Layer

Convolutional layers' interpretation has been extensively studied since they have been invented [29], producing several visualization techniques. Representations learned by these layers are *highly amenable to visualization*, considering they're representations of visual concepts.

In a standard Capsule Network, only *one* convolutional layer is employed in the network as the *first layer*. This implies that it will have a strong expressive power, since it will extract basic features easily interpretable. To get a view of the learned features, we **display the corresponding feature maps** by visualizing the values returned by the activation function. This is a really straightforward technique, which we think is really effective to interpret the convolutional layer in a Capsule Network.

In Figure 2.1, we displayed some really interpretable feature maps produced by our software. We can see that **edge detection** was performed on the entity in the input: from the left, 2nd and 4rth feature maps were activated on *right and left* edges, while 1st and 3rd feature maps detected *top and bottom* edges.

Figure 2.1: *Some feature maps produced by the convolutional layer as they're visualized in the software.*

2.3.2 Primary Capsules

Primary Capsules *combine features* extracted by the convolutional layer, ideally *encoding entities* that compose a larger one in the image. Each capsule in a Primary Capsule **encodes the pose** of the entity and its **probability of being present** in a localized part of the input.

Unlike the convolutional layer, it is **not possible to directly display the activation of a Primary Capsule**, since it results in a set of multidimensional vectors capturing *unknown properties* of the entity, which cannot be summarized in a single image. Instead, by computing capsules' length we will get a set of scalars.

So, to try and interpret a Primary Capsule's entity, we can display a **matrix based on each capsule's length**, describing the localized parts where the entity was found. Since displaying the matrix alone is is not very expressive for small Primary Capsules, we treated it as a **heatmap** and then we *rescaled and superimposed* it to the original input. This technique is *less immediate* than the one utilized for Convolutional layers, but it can still be used to interpret entities found by the network.

In Figure 2.2, we sampled 4 visualizations derived from different Primary Capsules: as we can see, Primary Capsules were able to detect **localized parts** of the number two (with *different degrees of certainty*). The fact that the network has learned to recognize localized part in the number two is a good result, since it was one of the prerogative of the Capsule Networks.

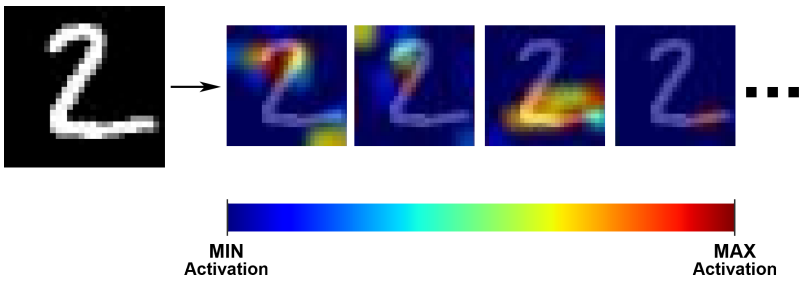


Figure 2.2: *Locations of 4 entities detected by Primary Capsules.*

2.3.3 Dense Capsules

Dense Capsules **encode entities obtained by a hierarchical composition of the outputs of Primary Capsules**. For the same reason given for Primary Capsules, it is not possible to directly visualize them: that is why, in the original paper [2], a decoder network was added and employed to better understand the expressive power of each capsule.

But what about Capsule Networks without a decoder? We thought that it would be interesting to extract and visualize some information from Dense Capsules without being tied to its presence.

Therefore, we employed an already existing technique to better understand and analyze those capsules: **Routing Path Visualization**.

Routing Path Visualization

Routing Path Visualization is a visualization technique recently introduced by Aman Bhullar with the intention of *interpreting the entity* that a given Dense Capsule detects [28]. Actually, we can note that it **displays relevant regions** of the input that **contributed to the final prediction** of a *single Dense Capsule*.

It is based on the ideas that *each Primary Capsule* **preserves the entity's relative position** in the input through the usage of its capsules and that the *Dynamic Routing algorithm* can **distinguish between active capsules containing relevant and irrelevant information** by altering the coupling coefficients.

Assuming we have a capsule i in the Primary Capsule layer and a capsule j in the following Dense Capsule layer, we can measure **how much of capsule i is routed to capsule j** through the path $z = (i, j)$ that relates them with:

$$m_z = l_i \cdot (c_{ij} \cdot l_j)^2 \tag{2.1}$$

where l_i and l_j are respectively the computed length of capsule i and j, while c_{ij} is the coupling coefficient derived by the Routing algorithm. Since the coupling coefficients and the capsules' length are always between 0 and 1, so will be m_z.

Now, assuming that each Primary Capsule uses a grid of $H \times H$ capsules where cells are enumerated as $1 \dots H^2$, we can get the **contribution of input's location** h to obtain the output of capsule j with:

$$p_h^{(j)} = \frac{\sum_{k=1}^{|Z|} m_{z_k}}{|Z|} \tag{2.2}$$

where Z is the set of paths connecting capsules i that shares the same h index to capsule j. At the end, $p_h^{(j)}$ will represent the **pixel value** in position h for the Routing Path Visualization of capsule j.

Differently from the original work that directly displayed the Routing Path Visualization, we treated it as a **heatmap** and then we rescaled and superimposed it to the original input. This way, we have found out that Routing Path Visualization can be **better interpreted** for Capsule Networks employing Primary Capsules with a small $H \times H$ grid.

Also, by running some experiments, we have seen that there is no point in the **square elevation** in equation 2.1, since it will alter visualization by making it more confused and less interpretable, so **we decided to remove it**.

Finally, note that Routing Path Visualization is enough *flexible* to be employed between **more than two consecutive capsule layers**, by extending the Z set and consequently the equation 2.1. However, in our work we provided a **simpler and optimized implementation** limited to Capsule Networks architectures containing only two consecutive capsule layers, since our experiments were restricted to those.

In Figure 2.3, we visualized Routing Path Visualizations derived from Class Capsules employed in a Capsule Network for digit recognition. As we can see, every visualization apart from the one recognizing the number five is not active, since **no regions of the image helped to recognize a number different from the five**. Moreover, from Capsule five's visualization, we can derive that the network only needed to detect a **small number of white pixels on the top-right of the image to conclude that there is a five in the input image**, since most likely only the number five present them in this arrangement.

In Figure 2.4, we provide a better insight of our choice to remove the square from Equation 2.1, by applying the original and modified method to the same model and input. Since the square in the original formula has the effect of flattening the final values, the top visualization results more confused and less interpretable: the bottom-left corner is not so emphasized, as well as the top-right edge of the zero. Instead, in the bottom visualization the difference between the focused area and the background is way more evident, helping us in the interpretation of the interesting areas.

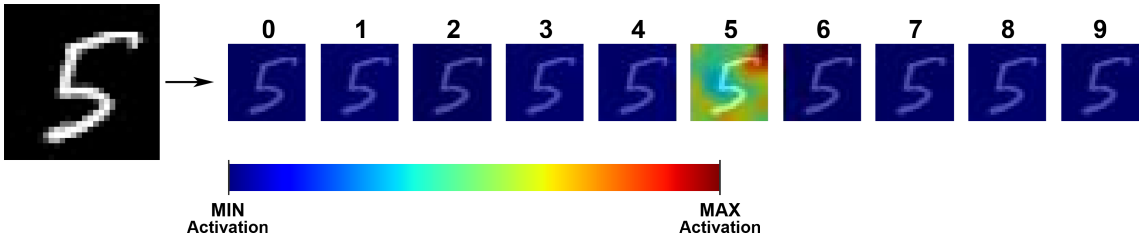


Figure 2.3: *Routing Path Visualization of Dense Caps dedicated to the final recognition of a digit in the input image.*

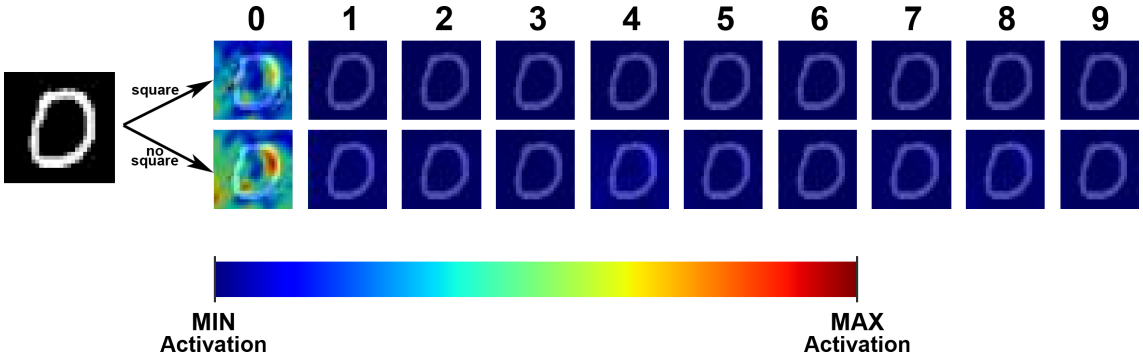


Figure 2.4: *Outputs of the original method and our modified version of Routing Path Visualization.*

2.3.4 Decoder

The Decoder aims to **reconstruct the input image**, given the output of the largest Class Capsule. In the original paper [2], it was analyzed how **perturbed versions** of this capsule **were reconstructed by the decoder**, to try and interpret what the individual dimensions represent. The perturbed versions were produced by **tweaking one of the 16 dimensions at a time**, by intervals of 0.05 in the range [-0.25, 0.25].

In Capsule Network Knowledge Extractor, we repropose this technique, but *we also went further*: we thought that, besides its dimensions, **capsule's length can be manipulated** and analyzed. Since a Class Capsule's length represents the confidence of the network that a given class is present in the image, we expect that by tweaking capsule's length we might get reconstructions that represent the class in a *stronger* or *weaker* manner. We produced manipulated versions of the capsule by **normalizing** the vector at lengths in the range [0.0, 1.0] by intervals of 0.1 with the formula:

$$x = x \cdot \frac{p}{\|x\|}$$

where x is the vector and p is the wanted length. We think that this technique could help us to understand how the network **represents an entity** when it is **not sure about its existence**.

Figures 2.5 and 2.6 show the **outputs of those two techniques**, applied on the same input and trained model.

Figure 2.5 shows the results of our new technique. We can effectively see that the network **has learned how to represent the nine at different degrees of certainty**. We can interpret the left-most reconstruction as how the network represents something which is surely not a nine nor any other class, since the input to the decoder will be composed of almost all zero values: it is the **representation of what the network thinks is completely new and unknown**. Moving gradually to the right, images become less blurred and the nine becomes more recognizable.

In Figure 2.6 we employed the original technique proposed by the paper, sampling the results of 4 dimensions perturbed. We can see that the network has learned how to model **different sizes of the nine's circular section** (dimension 0), as well as **different types of stroke** (dimensions 1, 3) and **localized skew** (dimension 2).

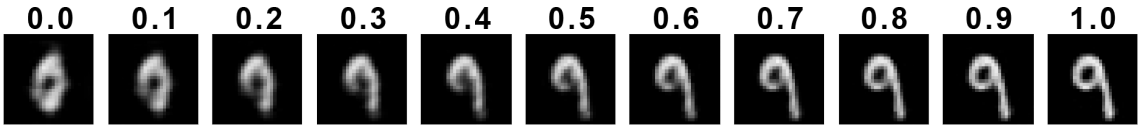


Figure 2.5: *Decoder's outputs of the same input with different length.*

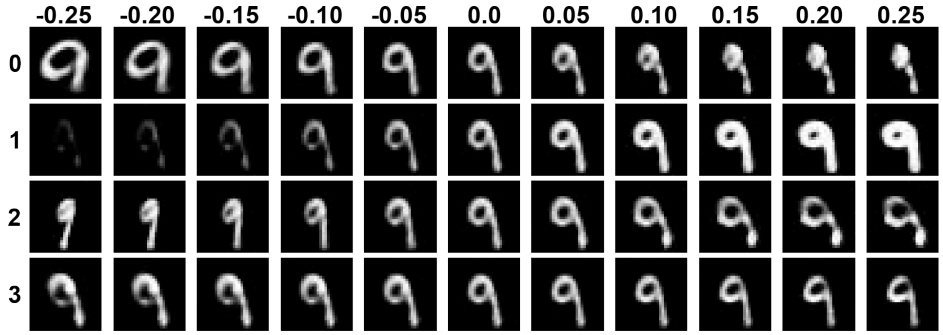


Figure 2.6: *Decoder's outputs of the same input 4 different dimensions perturbed individually.*

2.3.5 Grad-CAM++

Grad-CAM is a recent visualization technique introduced in 2016 [24], which can be used to produce a **heatmap visualization for a given class label**. More intuitively, Grad-CAM can show us *where an ANN is looking at* in the input to determine the existence of a given class instance. It works by examining the gradient information flowing into the penultimate layer (last one before the one producing the final classifications).

Some months later, **Grad-CAM++** was proposed [25]. It is an enhanced version which provides a better localization of objects as well as explaining occurrences of multiple objects of a class in a single image.

One of the results that we obtained in this thesis is to experimentally prove that Grad-CAM++ also **successfully works on Capsule Networks** by employing *tf-keras-vis* [26] library. Moreover, we found out that Grad-CAM++ and Routing Path Visualization produce **comparable heatmaps** when Primary Capsules employ a *sufficiently large grid of capsules*. Routing Path Visualization could then be employed as a **gradient-free, more direct and intuitive alternative** for Capsule Networks attention's interpretation.

Figure 2.7 provides a comparison between the output of Grad-CAM++ and Routing Path Visualization. As we can see, both visualization focus on the **same high activation areas**, that are *near edges pixels*. However, the output of Routing Path Visualization presents those areas in a *more condensed way*, while Grad-CAM++ is more confident about neighbor pixels being important as well: we can conclude that Grad-CAM++ produces results that can be interpreted with less effort, at least on data sets employing small images. It would be interesting to compare these two techniques on bigger data sets.

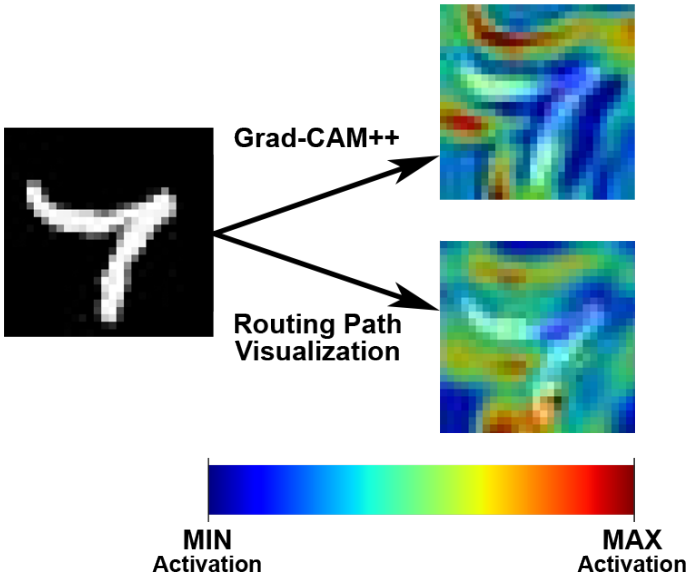


Figure 2.7: *Comparison between Grad-CAM++ output and Routing Path Visualization output.*

2.3.6 Activation Maximization (*How to produce adversarial examples*)

Activation Maximization is an old visualization technique [22] which was proved useful to analyze convolutional and fully connected layers. Its objective is to **synthesize an input** (e.g. an image) **that highly activates a specific neuron**. So, the expected result should represent *what a specific neuron has learned to recognize* in the input. The final visualization is generally obtained by starting from a random image, iteratively refining each pixel via back-propagation to increase the activation of a single neuron.

In order to obtain more defined and noiseless results, different functions acting as *regularizers* have been introduced to adjust the input at each step of the back-propagation. We recommend [30] for an interactive and well explained introduction to these topics.

We've started to experiment this visualization technique on Capsule Networks in a naive approach to try and test if it was easily usable, by employing once more the *tf-keras-vis* library to help us in this task. By "naive approach", we mean that we started from a completely black image and we did not employ any regularizer function, trying to maximize the activation of one of the Class Capsules. This way, we were able to obtain deterministic results at each execution.

As previous studies have shown [31], approaches like ours produces **unrealistic, uninterpretable images**, because the set of all possible images is so vast that it is highly possible to produce *fooling* images that excite a neuron, but do not resemble the natural images that neuron has learned to detect. Of course, on capsules this truth is even greater, considering they employ vectors instead of scalars. So, instead of providing interpretable results, the intention of this section is to **provide a cue for further investigation** on Capsule Networks and possibly get to implement this technique (or maybe, an inspired one) in Capsule Networks Knowledge Extractor in the future.

In Figure 2.8, we **produced a new input image** through our naive approach to activation maximization and then we analyzed the classification provided by a trained network for digit recognition. We **fooled our network** by making it highly believe **there is a seven in the image**, while for us it just appear as a *random high-frequency image*. Similar results can be obtained by maximizing the activation of other capsules.

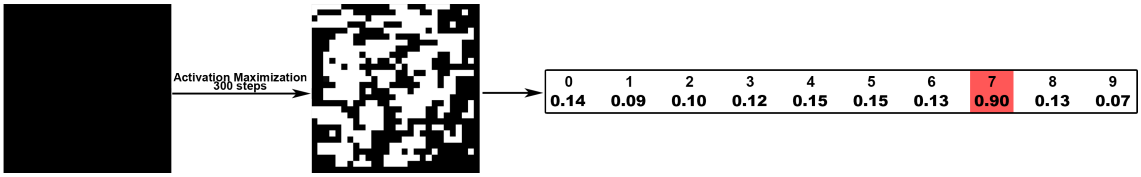


Figure 2.8: *Activation Maximization technique executed with our naive approach on a Capsule Network for digit recognition.*

2.4 Software User Interface

We structured the User Interface (UI) of our software to be minimal, while being easy to use and navigate. To produce an output, it is required to:

1. Select a model;
2. Select a training step;
3. Select an input image.

It is important to specify the model before the other two options: given the model, we have access to its weights saved at each training step, as well as to the data sets on which it was trained. We implemented this constraint in the software by disabling the last two choices until the first is selected, guiding the user. Choices are **remembered until the page is refreshed**, so the user is free to select the first two options and then analyze visualizations of different inputs quickly, without performing re-selections.

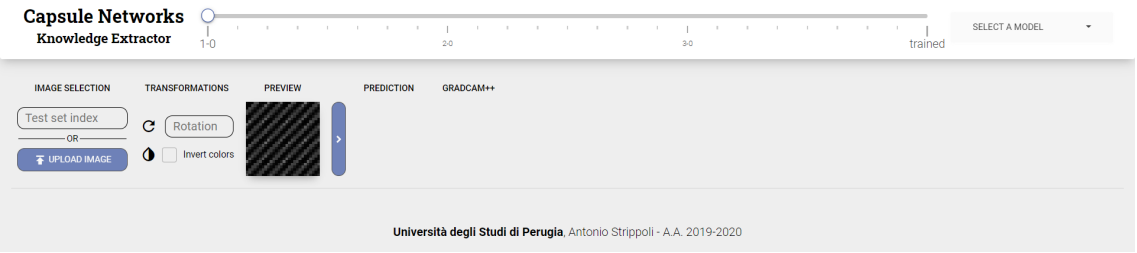


Figure 2.9: *Screen of Capsule Network Knowledge Extractor before any selection is performed.*

We expect the user to create a lot of different models, so we implemented the model's selection as a **drop-down menu** which allows to select one of the models or perform a search among them.

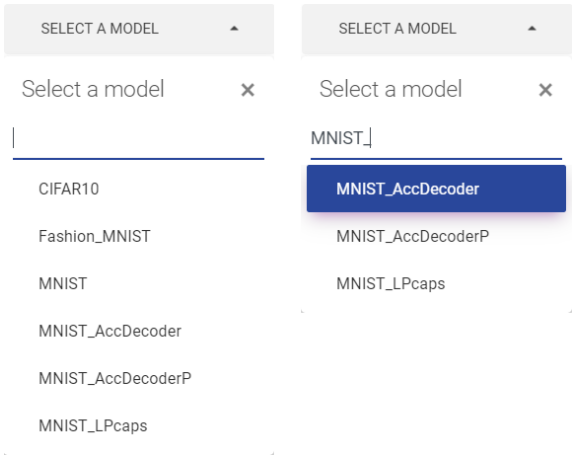


Figure 2.10: *Model selector utilized in Capsule Network Knowledge Extractor.*

We also wanted to make the user well aware of the training step he selected with respect to the others available, so we implemented its selection as a **slider**, constantly displaying an overview of all the steps.

Figure 2.11: *Slider implemented for the training step selection in Capsule Network Knowledge Extractor.*

Finally, we provided two ways to select the input: by **specifying the test set index** or by **uploading an image**. Furthermore, we thought that it would be interesting to analyze visualizations when the image is rotated or the colors are inverted, so we implemented these two selections as well.

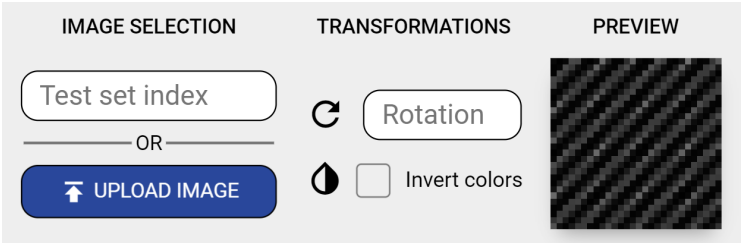


Figure 2.12: *Input selector in Capsule Network Knowledge Extractor.*

The last relevant elements in the UI are the produced outputs, which we organized in two sections:

1. **Layer-independent section**: the outputs are placed right next to the input section. We provided a histogram representing the **probability distribution of the predictions** as well as the **Grad-CAM++** output;
2. **Layer-dependent section**: we provide different **tabs grouping the outputs of a specific layer**. Since a layer could be interpreted using multiple techniques (e.g. the decoder's input can be manipulated both on magnitude and dimensions), we visualize a **title resuming the technique** on the top of each visualization.

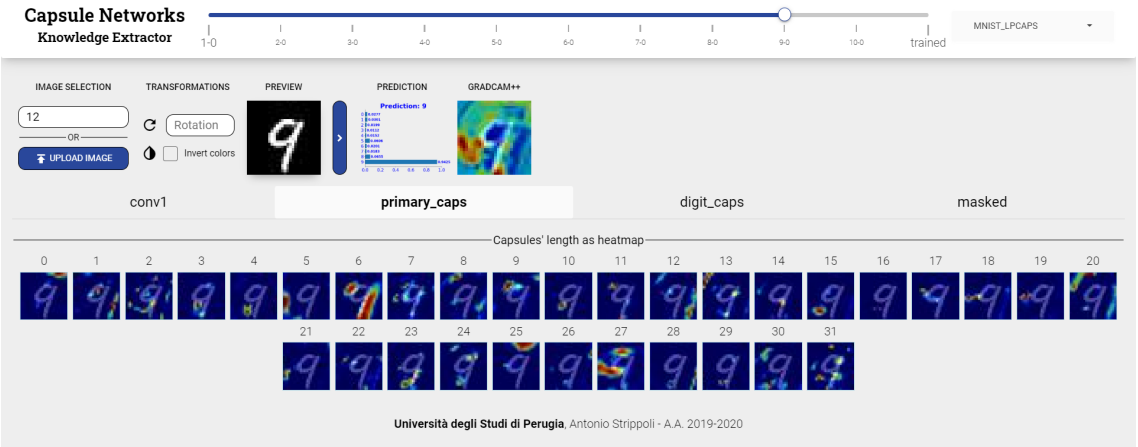


Figure 2.13: *Capsule Network Knowledge Extractor in action: visualizations for primary capsules of the twelfth image of the MNIST test set are visualized.*

2.5 Trainer

To train all of the models that we employed for our analysis, we had to **re-implement the Capsule Network logic** in *Keras with Tensorflow 2* as backend, since no version publicly available was up-to-date and/or sufficiently flexible. However, it is important to note that the original implementation in Keras (with Tensorflow 1 as backend) that greatly inspired our implementation, is *CapsNet-Keras* by *Xifeng Guo* [32].

By implementing it ourselves, we had the opportunity to fully understand every part of the baseline Capsule Network, with the future interest of studying interpretation's techniques more consciously.

Our implementation also goes further than the original one, since we implemented a *weight sharing* technique, that we will explain in details later. We wanted to make our implementation as crystal clear as possible, since it should be easily editable for improvements and/or additions. So, we've organized our implementation in a separated directory called ***_share***, which can be imported easily in other files.

A primary need for the project was to be able to compare different models. In this regard, we dedicated a directory to each model, to include both the architecture and the outputs. The directory's name will be employed as well for the model's name. We decided to utilize the format "dataset_edit" for the names (e.g. "MNIST_AccDecoder" will be the baseline model trained on the MNIST with an edit regarding the decoder). Every directory contains two files:

1. ***capsnet.py***: contains a *Model function* that **describes the architecture** using Keras' functional API. It will always returns two model objects: the first one for the training, the other for the evaluation, since during evaluation mask will be executed differently;

2. ***main.py***: **executes the training procedure** on the model described in *capsnet.py*, with the parameters provided to the function;

A **general output** produced by one of the *main.py* files, consists in:

1. A folder named ***weights*** containing the **weights saved during the training**. Each weights file is saved in the *.h5 format* containing only weights and biases[1]: this is the *lightest option* provided by Keras and also the *safest* in our opinion, since working with custom layers can cause some issues during the reload of the model. The format utilized for the filename of the weights is "epoch-batch" (e.g. a training step named "3-100" represent the model saved at epoch 3 after 100 batches);

2. A pickle file named "*model_params.pkl*", containing the **parameters for the model** defined in *capsnet.py*. This will be used to rebuild the model during the visualization process.

[1] https://www.tensorflow.org/guide/keras/save_and_serialize

2.5.1 Weight Sharing

In a DenseCaps layer of the baseline Capsule Network, we have a **weight for each link** that connects every *individual capsule* of the *previous layer*, to *each capsule* of the *current one*. This results in a **large number of trainable parameters**, but could lead to better performances.

Thinking about a classic **Convolutional layer**, it contains a form of *weight sharing*, where local features of the input are detected by the same filter. This form of weight sharing is not surprising: the same criteria for detecting features in one location of the inputs should be applied to the other locations as well.

The same idea could be applied to a DenseCaps layer: a PrimaryCaps layer contains several Primary Capsules, which employ a group of capsules detecting the same entity in different location of the input. It seems natural to subject all those individual capsules to the same criteria. Hence, we can introduce a form of **weight sharing**, where *capsules of the same type* use a **single weight** for *predicting entities* in the following layer. The original authors of this weight sharing technique are LaLonde and Bagci [33].

This is surely a fascinating idea and results in a dramatic reduction of trainable parameters, but could affect performances. We believe that studying this trade-off could lead to lightweight models with shorter training time, without a huge impact on the performances.

3 Results

To prove the usefulness of our software, we quantitatively produced and analyzed some visualizations of several models trained on **two different data sets**.

To make those results easier to reproduce, we picked the data sets from the Keras' official catalog. This way, we were also able to keep the code compact and consequently more readable, especially in the pre-processing dedicated code.

3.1 MNIST

The **MNIST** data set contains 70,000 28x28 grayscale images depicting *handwritten single digits*. Keras provides those images in a training set of 60,000 images and a test set of 10,000.

The original Capsule Networks paper [2] proposed a **model** capable of performing classification on this data set, obtaining **state-of-the-art performances**. In this work, this model will be considered as the baseline and we will refer to it as "MNIST". Table 3.1 summarizes the hyper-parameters chosen for this model.

Convolutional				
Filters	*Kernel Size*	*Strides*	*Padding*	*Activation*
256	9	1	valid	ReLU

Primary Capsules					
N. Caps	*Dim. Caps*	*Kernel Size*	*Strides*	*Padding*	*Activation*
32	8	9	2	valid	ReLU

Class Capsules			
N. Caps	*Dim. Caps*	*Routing iter.*	*Kernel initializer*
10	16	3	Normal, stddev 0.1

Fully Connected 1		**Fully Connected 2**		**Fully Connected 3**	
Units	*Activation*	*Units*	*Activation*	*Units*	*Activation*
512	ReLU	1024	ReLU	784	sigmoid

Table 3.1: *First Capsule Network Architecture proposed in [2].*

We repropose this model, along with some variations, in order to show how visualizations majorly change under different network's configurations.

In total, we produced 4 variations from the original model:

1. In **"MNIST_LPcaps"** we changed Primary Capsules' kernel-size and strides hyperparameters in order to use **larger grids of capsules**. We expected changes in both Primary and Dense Capsules' visualization, detecting smaller features in the image;
2. In **"MNIST_WSharing"** we employed the **weight sharing** technique in the Dense Capsules. We did not expect major changes in the visualizations, even if the network employs less parameters;
3. In **"MNIST_AccDecoder"** we increased the lambda recognition of the decoder, giving more importance to the decoder. We expected the **decoder's outputs to become more accurate** and much more closer to the input image;
4. In **"MNIST_NoDecoder"** we **removed the decoder** from the network. We expected to see a drop on the performances;

Name	Primary Capsules		Weight Sharing	Lambda Recognition
	Kernel Size	Strides		
MNIST	9	2	No	0.392
MNIST_LPcaps	**3**	**1**	No	0.392
MNIST_WSharing	9	2	**Yes**	0.392
MNIST_AccDecoder	9	2	No	**1.0**
MNIST_NoDecoder	9	2	No	**NO DECODER**

Table 3.2: *Overview of the variations made to the baseline model.*

All the models were trained on **Colab** (employing GPU acceleration) **for 10 epochs**, since it was enough to obtain excellent performances and stable visualizations.

Name	Trainable params.	Average time per epoch
MNIST	8,215,568	4m 10s
MNIST_LPCaps	15,293,456	27m 30s
MNIST_WSharing	6,781,968	6m 30s
MNIST_AccDecoder	8,215,568	4m 10s
MNIST_NoDecoder	6,804,224	3m 20s

Table 3.3: *Overview of the training information of the models we studied.*

As it can be seen from Figures 3.1 and 3.2, we plotted the **loss** and the **validation accuracy** of all the models previously discussed. All the models were able to obtain a final accuracy of over 99%, in both the training set and the validation set.

In the validation accuracy's plot, we can note that two models performed slightly worse: *"NoDecoder"* and *"LPcaps"*. As another work mentioned [20], **the decoder is crucial** to make Capsule Networks generalize and perform well, so it is foreseeable that it would not generalize as well as other models. On the other hand, by making Primary Capsules employ larger grid of capsules, the **learned features are smaller**, which could not be optimal for a data set like MNIST.

Finally, it is worth mentioning that the model *"WSharing"* obtained one of the **highest validation accuracies**, while requiring **less trainable parameters**.

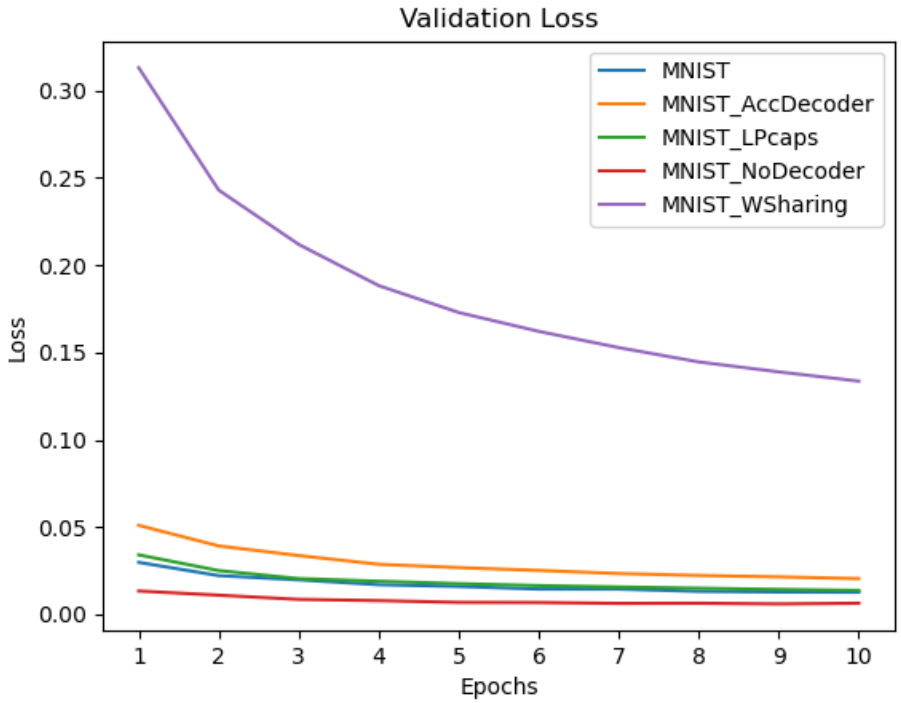


Figure 3.1: *Loss calculated for the models we trained on MNIST.*

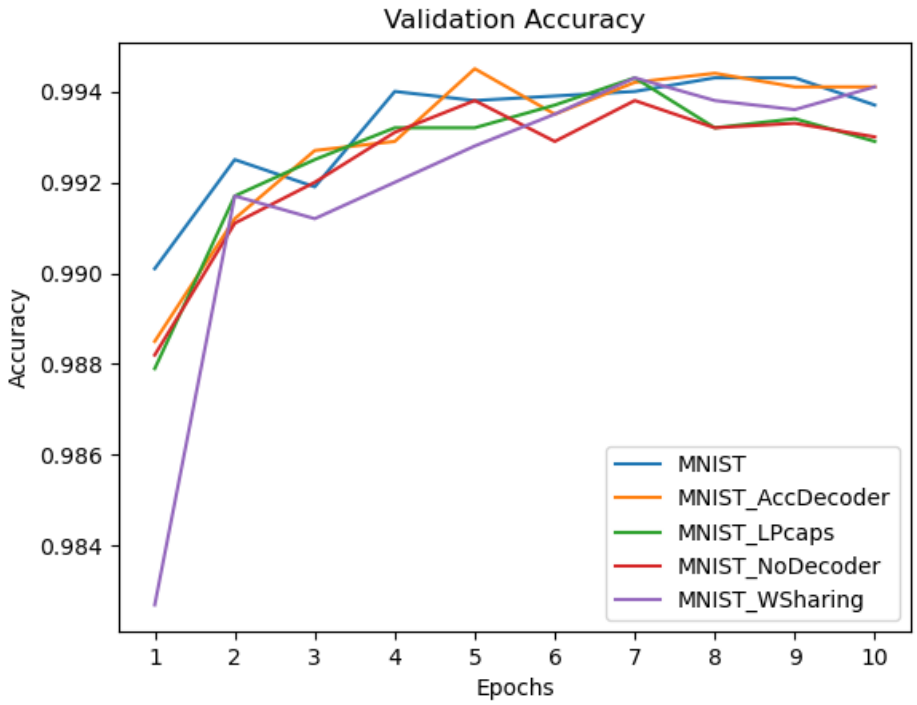


Figure 3.2: *Validation accuracy obtained by the models we trained on MNIST.*

Following, we **sampled visualizations produced for each layer** using the models we defined, comparing them to effectively show the information that can be obtained by *Capsule Network Knowledge Extractor* to draw some hypothesis. All the visualizations were obtained using the **fully trained models**, sending the same following input through the networks.

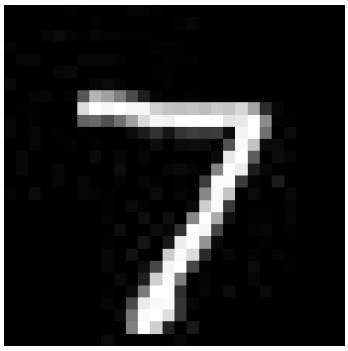


Figure 3.3: *The input we provided to the networks trained on MNIST. It is the first image from the MNIST test set provided by Keras.*

3.1.1 Visualizations: baseline model

First of all, we produced the visualizations using the baseline model. By analyzing them, we were able to identify some cues that allowed us to define the previously discussed variations on the baseline model.

Figure 3.4 shows the outputs of the Convolutional layer. We can observe that almost every activation map focuses on **edge detection**, while very few have been activated by the **white part of the input**. There is also a small percentage of maps which **have not been activated at all**.

Then, Figure 3.5 visualizes the Primary Capsules. As one could expect after analyzing the convolutional outputs, some of the entities that have been identified by the Primary Capsules **reside in the black background**, while a smaller percentage can be found in the white part. However, it is important to note that the majority of the visualizations are **difficult to interpret**: this is due to the fact that Primary Capsules employ a small grid of capsules, which causes the resulting heatmaps to be very small.

Next, Figure 3.6 shows the outputs obtained using Grad-CAM++ and Routing Path Visualization on the Class Capsules. Once more, the Routing Path Visualization results confirmed our precedent analysis: the **black background sections of the input contributed more** than the white ones to the final prediction. Comparing the output of the *Class Capsule n.7* with the output of the *Grad-CAM++* technique, we can see that this time they are completely discordant regarding the relevant parts of the input.

Finally, Figure 3.7 shows the decoder's output, revealing that the dimensions of the Class Capsule n.7 have captured **the horizontal segment of the seven, as well as different types of localized skew and stroke**, with a good variety and without overlaps.

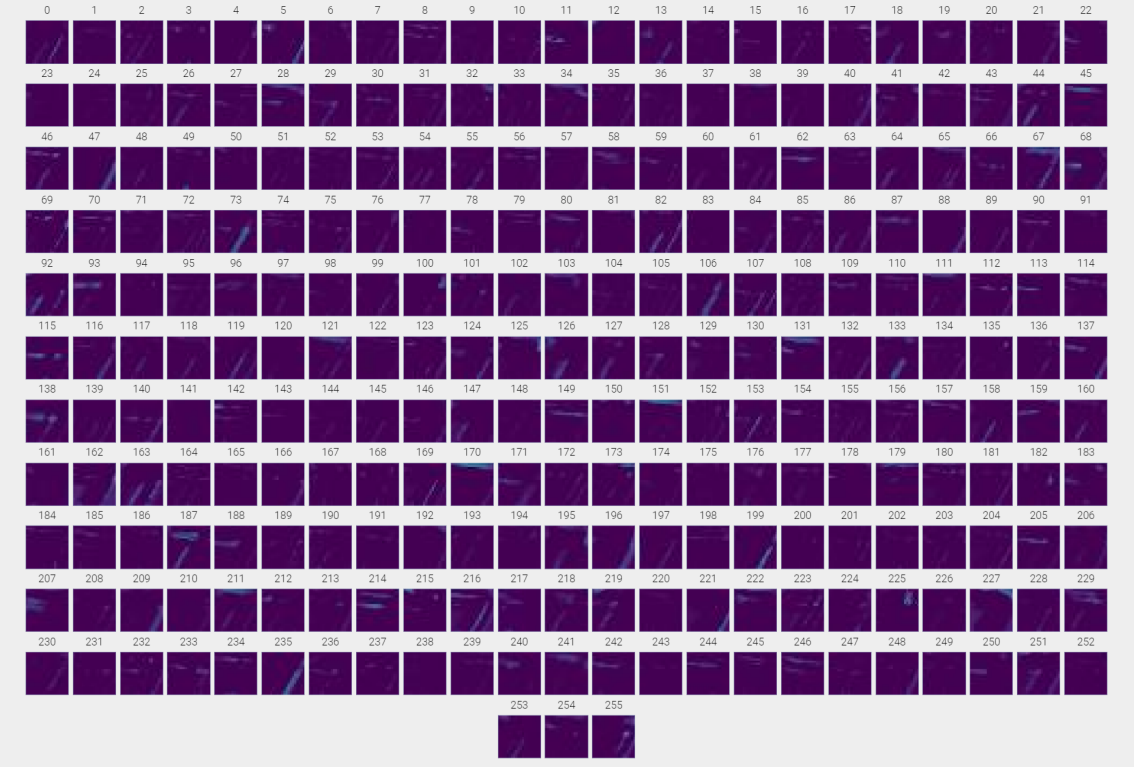


Figure 3.4: *Visualization of the Convolutional layer produced using the baseline model trained on MNIST.*

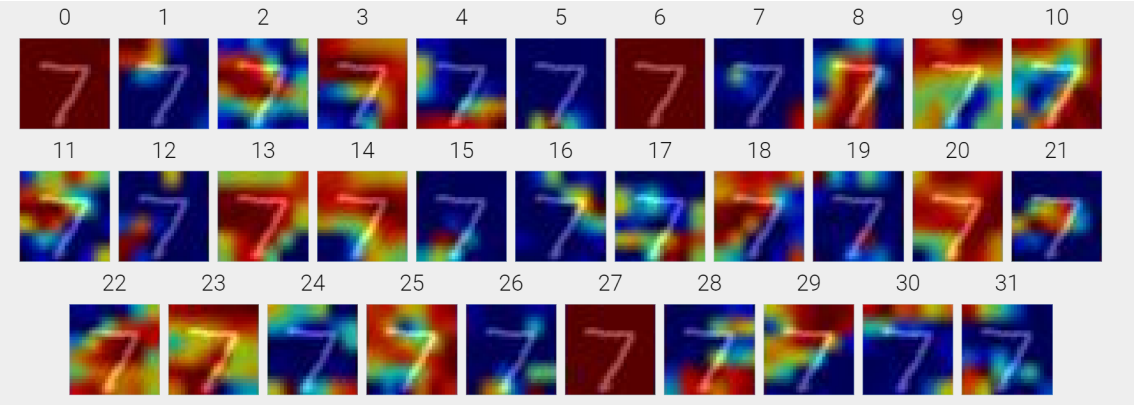


Figure 3.5: *Visualization of the Primary Capsules layer produced using the baseline model trained on MNIST.*

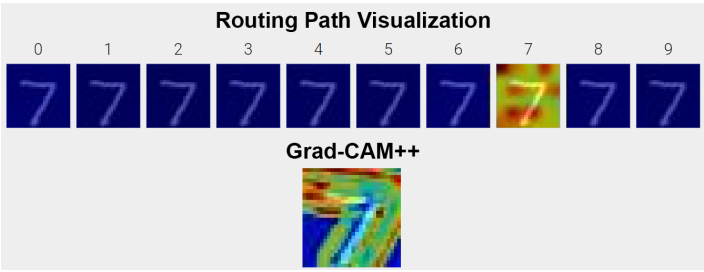


Figure 3.6: *Routing Path Visualization and Grad-CAM++ outputs produced using the baseline model trained on MNIST.*

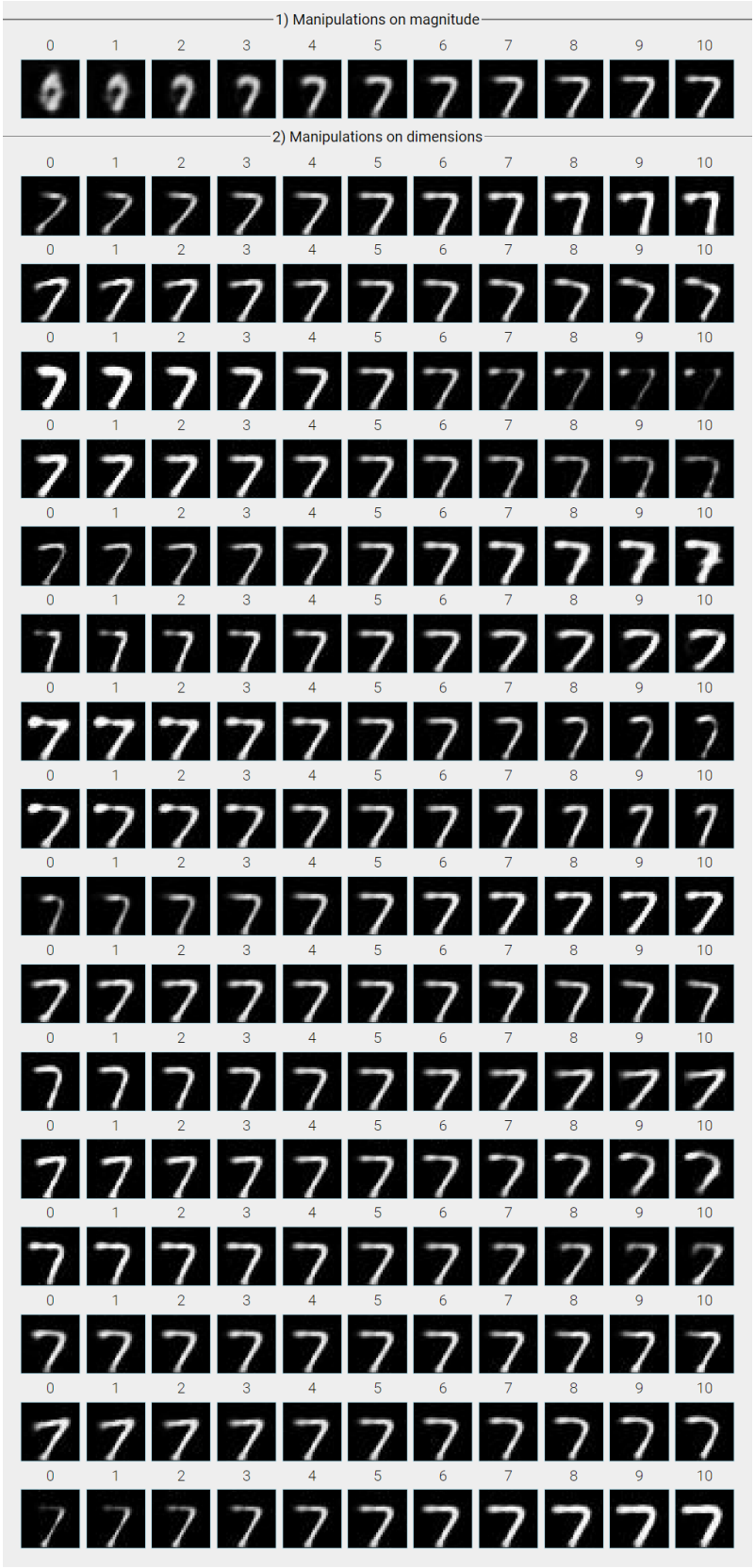


Figure 3.7: *Visualization of the Decoder network produced using the baseline model trained on MNIST.*

3.1.2 Visualizations: larger grid of capsules

We started by analyzing how visualizations could be affected using the model "MNIST_LPCaps", where larger grid of capsules are employed in the Primary Capsules. We found out that the majority of the differences can be found in the visualizations of the Convolutional, Primary Capsules and Dense Capsules layers, while the decoder's outputs have not undergone major changes.

Outputs of the Convolutional layer can be seen in Figure 3.8, where activation maps are generally **more defined and less blurred**, with a substantial increase of the maps activated on the **white part of the input**. Since we made Primary Capsules recognize smaller entities in the image, we can suspect that the convolutional layer now has more interest in detecting the thin segments of the digit seven, considering it is now possible to make better use of these information.

Next, Figure 3.9 shows the Primary Capsules visualizations, which are now more interpretable: Primary Capsules that have been activated on the background focuses on **smaller regions of the image**, capturing **white part's edges or noise**. Also, a substantial increased number of capsules now focuses on the **white part** of the image, which is consistent with the results observed in the Convolutional layer.

Lastly, Figure 3.10 shows the outputs obtained using Grad-CAM++ and Routing Path Visualization on the Class Capsules. Thus, we can confirm the observation we made in section 2.3.5 about **Grad-CAM++ and Routing Path Visualization being comparable**: in this case, they both mainly focus on the top edge of the seven.

3.1.3 Visualizations: weight sharing

As we hypothesized, there were **not major changes in the visualizations** by employing the weight sharing technique. However, we think this is an **encouraging result** that promotes this technique, since in this case **the expressive power of the internal representations has remained the same**.

3.1.4 Visualizations: increased lambda recognition

One major difference characterizes this model. From the decoder's outputs, we observed how we lost a lot of expressive power in the Class Capsule's dimensions: all the dimensions now seems to **encode similar features**. Additionally, the perturbations did not result in major changes in the decoder's outputs, which make us think that we probably lost a bit of the model's generalization.

3.1.5 Visualizations: absence of decoder

Apart from not being able to study anymore the internal representations of each Class Capsules' dimension, no major changes occurred in the visualizations produced.

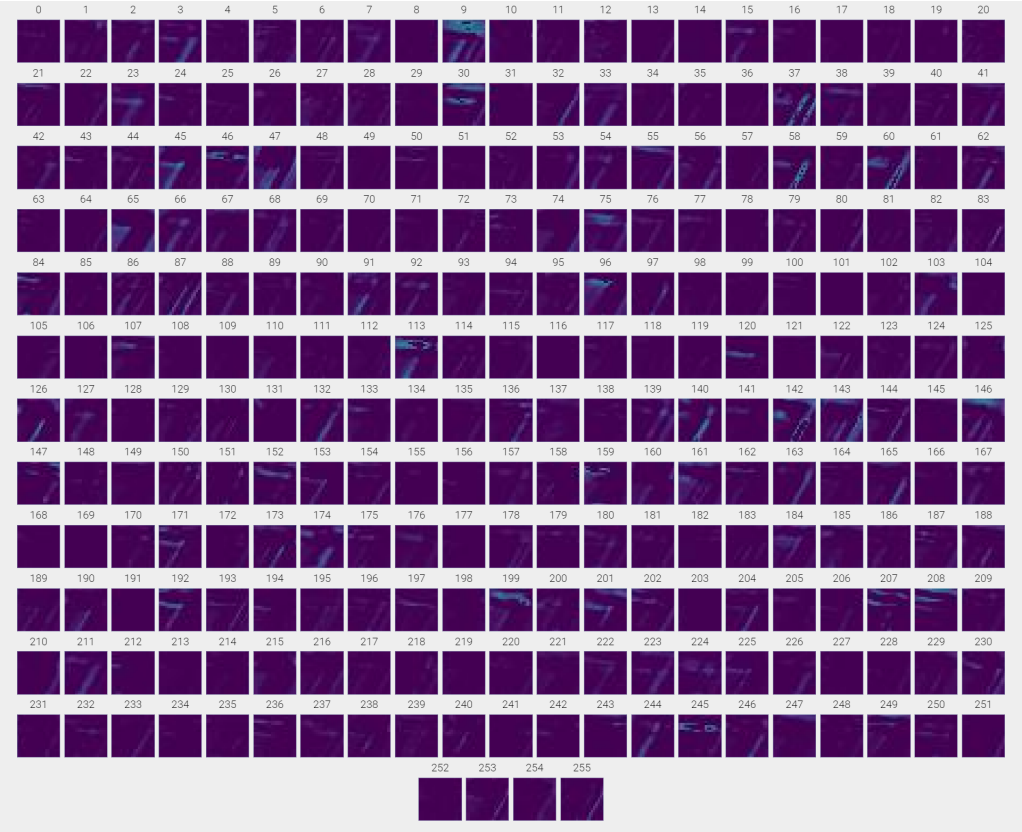


Figure 3.8: *Visualization of the Convolutional layer produced using the MNIST_LPCaps model trained on MNIST.*

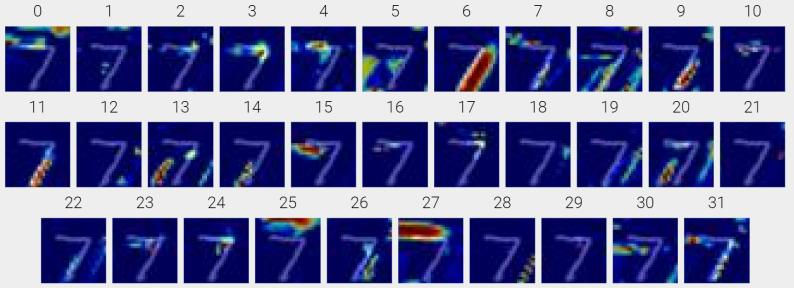


Figure 3.9: *Visualization of the Primary Capsules layer produced using the MNIST_LPCaps model trained on MNIST.*

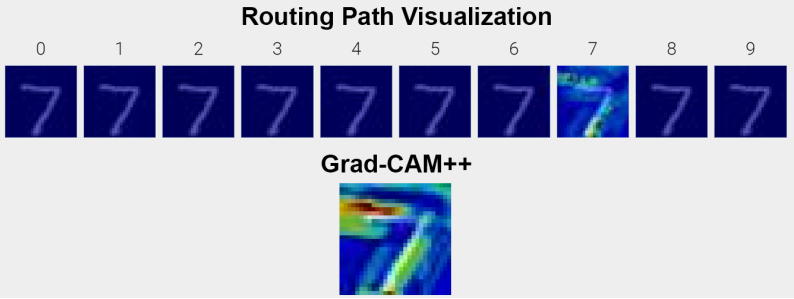


Figure 3.10: *Routing Path Visualization and Grad-CAM++ outputs produced using the MNIST_LPCaps model trained on MNIST.*

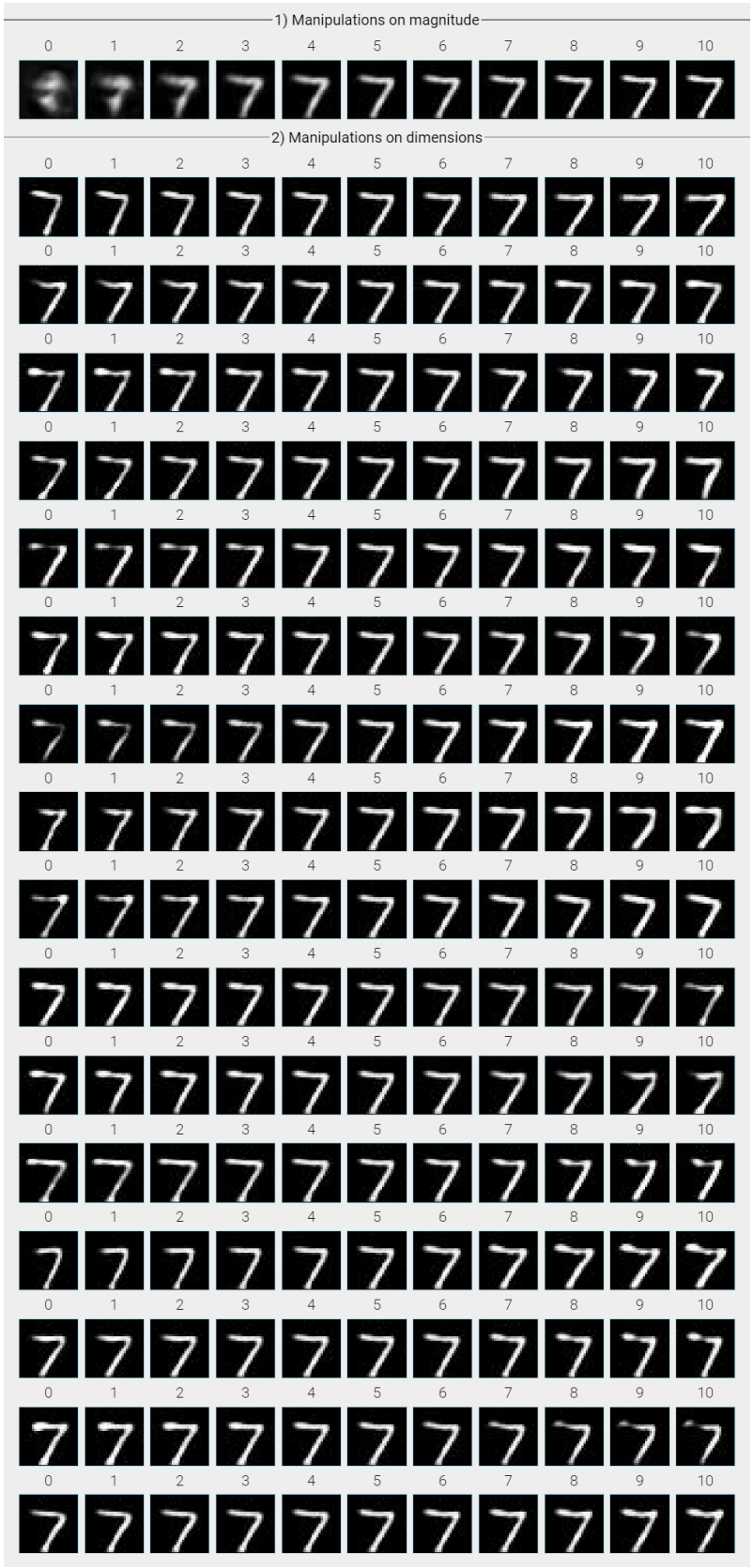


Figure 3.11: *Visualization of the Decoder network produced using the MNIST_AccDecoder model trained on MNIST.*

3.2 CIFAR10

We decided to include CIFAR10 data set in our experiments to show that our software is also capable of producing visualizations of models employing RGB input. Since our interest in this data set was limited, we only produced the visualizations based on a single model.

We used the same baseline model we employed on MNIST, this time trained on CIFAR10. As we did for the MNIST data set, we trained the model for 10 epochs, even if this time we did not obtain good performances, as it can be seen from Figures 3.12 and 3.13. This is to be expected, since CIFAR10 is a more complex dataset than MNIST and requires a more complex model.

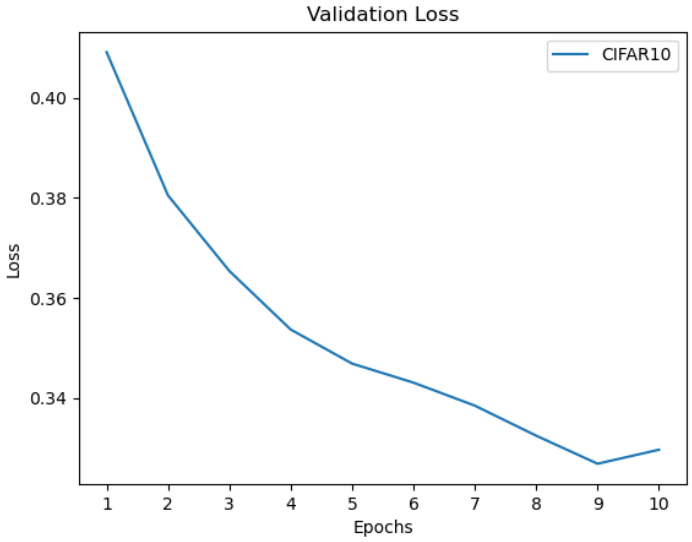


Figure 3.12: *Loss calculated for the model we trained on CIFAR10.*

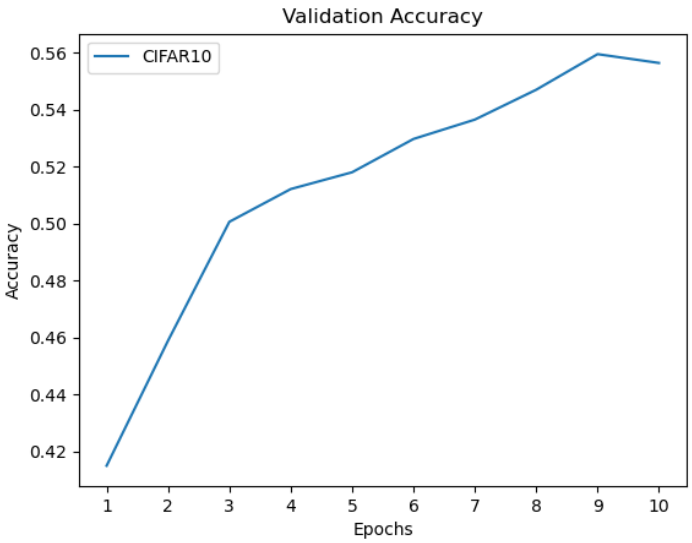


Figure 3.13: *Validation accuracy obtained by the model we trained on CIFAR10.*

In the next Figures, we visualize the outputs produced by Capsule Network Knowledge Extractor on this model, using the following input. As we can see, every technique can also be employed on RGB inputs.

Figure 3.14: *The input we provided to the network trained on CIFAR10. It is the 13th image from the CIFAR10 test set provided by Keras.*

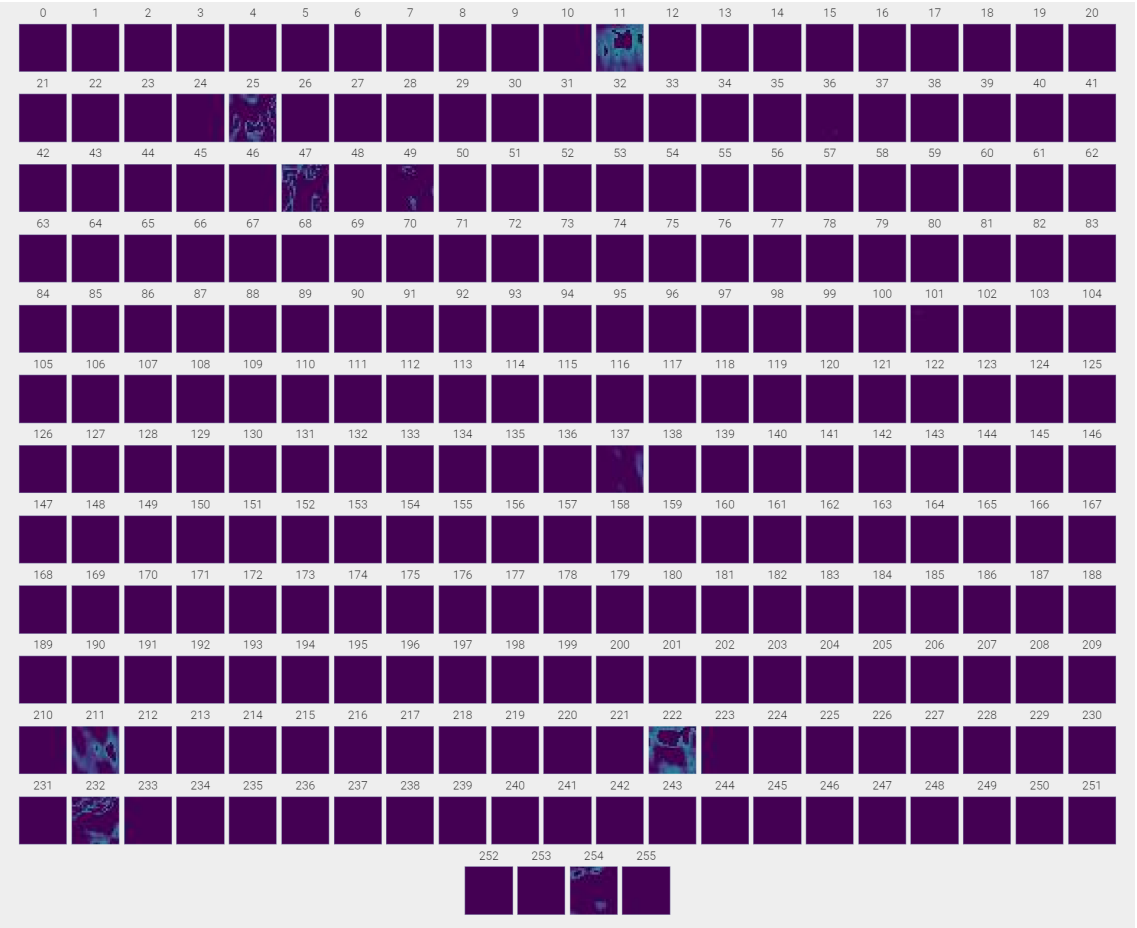


Figure 3.15: *Visualization of the Convolutional layer produced using the baseline model trained on CIFAR10.*

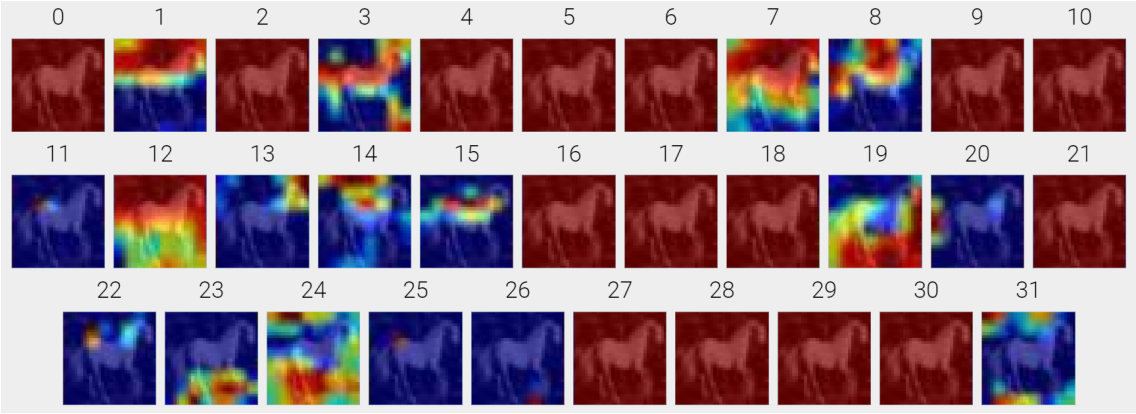


Figure 3.16: *Visualization of the Primary Capsules layer produced using the baseline model trained on CIFAR10.*

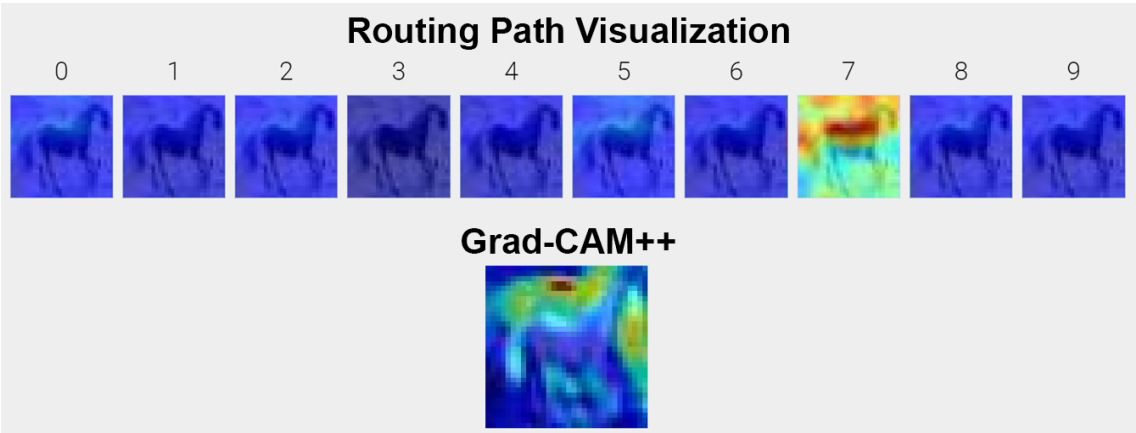


Figure 3.17: *Routing Path Visualization and Grad-CAM++ outputs produced using the baseline model trained on CIFAR10.*

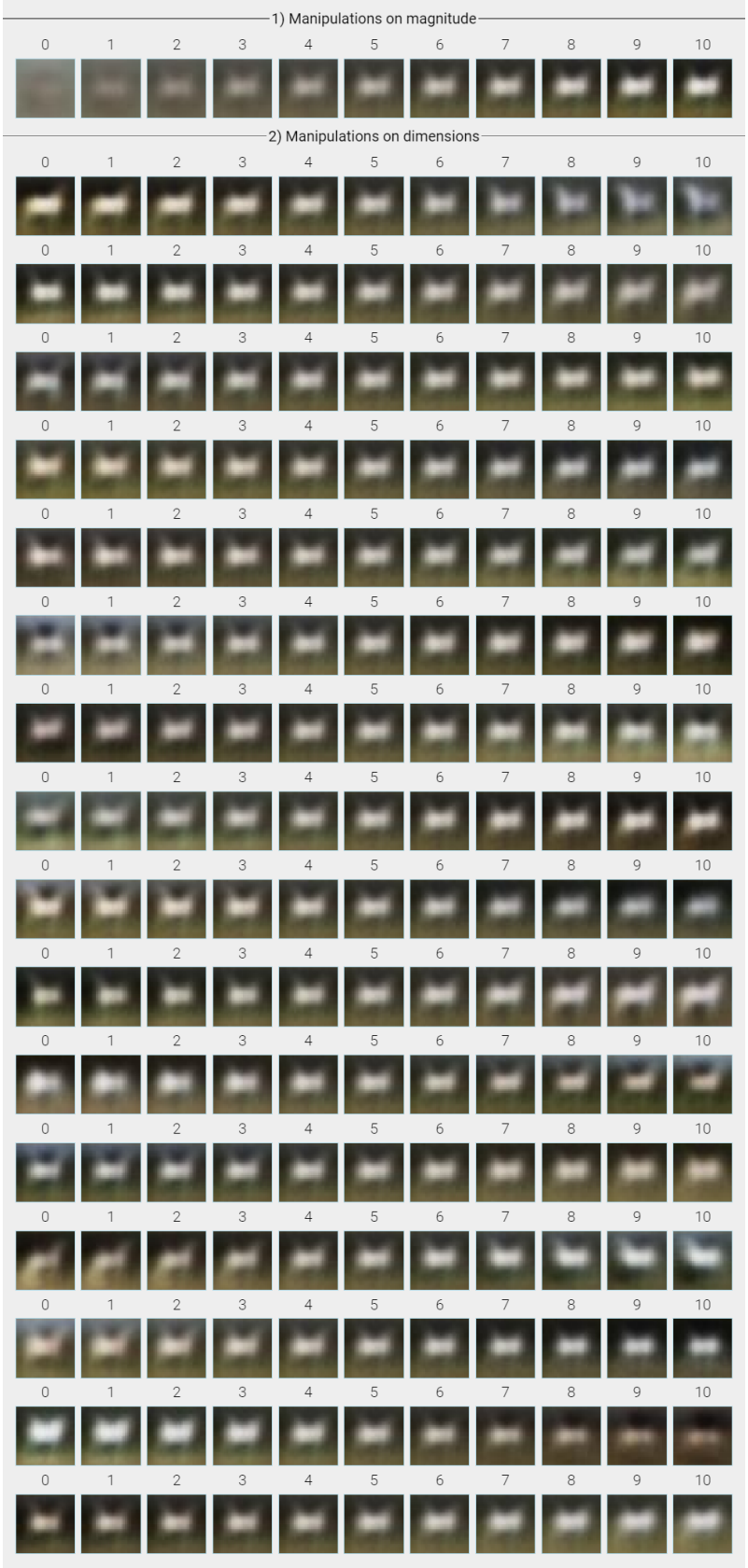


Figure 3.18: *Visualization of the Decoder network produced using the baseline model trained on CIFAR10.*

4 Conclusions

In this thesis we investigated visualization techniques for Capsule Networks and how they could be affected by major changes in the architecture employed. We showed that by visualizing learned features it is possible to interpret the purpose of each part of the network, providing concrete evidence of its generalization ability.

Specifically, we contributed to the Network Visualization field with **two new visualization techniques**, usable to evaluate the contributions of the **Primary Capsules** and the **Decoder network** inside the architecture. Both the proposed techniques have been proven effective and useful: the first one allowed us to **identify recognized minor entities in localized parts of the input** image, while the second one showed **reconstructions of the representative entity of a class when the network is not sure about its existence**.

Moreover, we studied other techniques already proposed in literature and their usability on Capsule Networks: **Grad-CAM++** [25] and **Routing Path Visualization** [28]. While Grad-CAM++ technique was proven to be **easily usable and interpretable on Capsule Networks** by means of the *tf-keras-vis* library, Routing Path Visualization outputs have not always been proven to be so clear. Our analysis on this technique led to an **improved** version, making the **outputs less flat** and more easily interpretable. Furthermore, Routing Path Visualization's output was **interestingly shown comparable** to the one of the already established Grad-CAM++ technique, provided that Primary Capsules employ a *sufficiently large grid of capsules*. Lastly, we analyzed how changes to a Capsule Network architecture could lead to major changes on the **manipulated Class Capsules reconstructed through the Decoder**, confirming the potential of this technique.

All these techniques have been collected and employed in a new software named **Capsule Network Knowledge Extractor**. Through our software, it is possible to easily **produce visualizations**, grouped by layer, of a Capsule Network model created with **Keras under TensorFlow 2.0**. The software is also able to **load weights saved at different training steps**, allowing to analyze how visualizations have changed during the training, and perform some transformations to an input, with the intent of analyzing how they affect the visualizations. We hope to see our software's development going further, since it has been completely open sourced and uploaded on GitHub.

4.1 Further improvements

Although we have shown that in some cases Grad-CAM++ outputs are comparable to Routing Path Visualization ones, it would be interesting to test how those techniques behave on **data sets employing larger images**. In fact, our tests are not meant to be exhaustive, but should be considered a cue to deepen the topic.

Finally, further investigation on Activation Maximization as an effective method to interpret Capsule Networks are encouraged. We discussed about how this technique has been revealed successful on CNNs when instilled with natural priors. We think it would be a major challenge to **study an effective way to produce good results on a Capsule Network**.

References

[1] Geoffrey E. Hinton Alex Krizhevsky Ilya Sutskever. *ImageNet Classification with Deep Convolutional Neural Networks*. 2012. URL: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[2] Geoffrey E. Hinton Sara Sabour Nicholas Frosst. *Dynamic Routing Between Capsules*. 2017. URL: https://arxiv.org/pdf/1710.09829.pdf.

[3] Wikipedia. *Timeline of machine learning*. URL: https://en.wikipedia.org/wiki/Timeline_of_machine_learning.

[4] Tom Mitchell. *Machine Learning*. McGraw Hill., 1997.

[5] DeepMind. *AlphaZero: Shedding new light on chess, shogi, and Go*. URL: https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go.

[6] dewangNautiyal. *Underfitting and Overfitting in Machine Learning*. URL: https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/.

[7] Daniel Tan. *Deep Learning Glossary*. URL: https://hackmd.io/@birdx0810/B1GpzTWN7.

[8] Aaron Courville Ian Goodfellow Yoshua Bengio. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[9] J. Williams D. Rumelhart G. Hinton. *Learning representations by back-propagating errors*. 1986. URL: http://www.cs.utoronto.ca/~hinton/absps/naturebp.pdf.

[10] Frederik Kratzert. *Understanding the backward pass through Batch Normalization Layer*. URL: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html.

[11] Yoon Kim. *Convolutional Neural Networks for Sentence Classification*. 2014. URL: https://arxiv.org/pdf/1408.5882.pdf.

[12] Himadri Sankar Chatterjee. *A Basic Introduction to Convolutional Neural Network*. URL: https://medium.com/@himadrisankarchatterjee/a-basic-introduction-to-convolutional-neural-network-8e39019b27c4.

[13] user194703. *Visualizing Matrix Convolution*. URL: https://tex.stackexchange.com/questions/522118/visualizing-matrix-convolution.

[14] Mohit Deshpande. *Introduction to Convolutional Neural Networks for Vision Tasks*. URL: https://pythonmachinelearning.pro/introduction-to-convolutional-neural-networks-for-vision-tasks/.

[15] Ashutosh Kumar. *Why Do Capsule Networks Work Better Than Convolutional Neural Networks?* URL: https://analyticsindiamag.com/why-do-capsule-networks-work-better-than-convolutional-neural-networks/.

[16] Max Pechyonkin. *Understanding Hinton's Capsule Networks.* URL: https://pechyonkin.me/capsules-1/.

[17] Geoffrey E. Hinton. *AMA Geoffrey Hinton*. URL: https://www.reddit.com/r/MachineLearning/comments/2lmo0l/ama_geoffrey_hinton/clyj4jv/.

[18] S. D. Wang G. E. Hinton A. Krizhevsky. *Transforming Auto-encoders*. 2011. URL: http://www.cs.toronto.edu/~bonner/courses/2020s/csc2547/papers/capsules/transforming-autoencoders,-hinton,-icann-2011.pdf.

[19] Konstantinos N. Plataniotis Atefeh Shahroudnejad Arash Mohammadi. *Improved Explainability of Capsule Networks: Relevance Path by Agreement*. 2018. URL: https://arxiv.org/pdf/1802.10204.pdf.

[20] Aggelos K. Katsaggelos Arjun Punjabi Jonas Schmid. *Examining the Benefits of Capsule Neural Networks*. 2020. URL: https://arxiv.org/pdf/2001.10964.pdf.

[21] Zachary C. Lipton. *The Mythos of Model Interpretability*. 2016. URL: https://arxiv.org/pdf/1606.03490.pdf.

[22] Aaron Courville Dumitru Erhan Yoshua Bengio and Pascal Vincent. *Visualizing Higher-Layer Features of a Deep Network*. 2009. URL: https://pdfs.semanticscholar.org/65d9/94fb778a8d9e0f632659fb33a082949a50d3.pdf.

[23] Anh Nguyen et al. *Synthesizing the preferred inputs for neurons in neural networks via deep generator networks*. 2016. URL: https://arxiv.org/pdf/1605.09304.pdf.

[24] Abhishek Das... Ramprasaath R. Selvaraju Michael Cogswell. *Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization*. 2016. URL: https://arxiv.org/abs/1610.02391.

[25] Prantik Howlader Aditya Chattopadhyay Anirban Sarkar and Vineeth N Balasubramanian. *Grad-CAM++: Improved Visual Explanations for Deep Convolutional Networks*. 2017. URL: https://arxiv.org/pdf/1710.11063.pdf.

[26] Yasuhiro Kubota and contributors. *tf-keras-vis*. https://github.com/keisen/tf-keras-vis. 2019.

[27] Nick Bourdakos and contributors. *CapsNet-Visualization*. https://github.com/bourdakos1/CapsNet-Visualization. 2018.

[28] Aman Bhullar. *Interpreting Capsule Networks for Image Classification by Routing Path Visualization*. 2020. URL: https://atrium.lib.uoguelph.ca/xmlui/bitstream/handle/10214/17834/Bhullar_Aman_202003_Msc.pdf?sequence=1&isAllowed=y.

[29] Rob Fergus Matthew D Zeiler. *Visualizing and Understanding Convolutional Networks*. 2013. URL: https://arxiv.org/pdf/1311.2901v3.pdf.

[30] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. "Feature Visualization". In: *Distill* (2017). https://distill.pub/2017/feature-visualization. DOI: 10.23915/distill.00007.

[31] Jeff Clune Anh Nguyen Jason Yosinski. *Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images*. 2014. URL: https://arxiv.org/pdf/1412.1897.pdf.

[32] Xifeng Guo. *CapsNet-Keras*. 2017. URL: https://github.com/XifengGuo/CapsNet-Keras.

[33] Ulas Bagci Rodney LaLonde. *Capsules for Object Segmentation*. 2018. URL: https://arxiv.org/pdf/1804.04241.pdf.

Acknowledgements

Desidero esprimere i miei più sentiti ringraziamenti alla professoressa Valentina Poggioni, per aver creduto in me ed avermi affidato una tesi di una certa importanza, nonostante la mia inizialmente scarsa conoscenza in materia. Ringrazio poi la dottoranda Alina Elena Baia, per avermi assistito e dedicato il suo tempo e professionalità. L'esperienza e competenza di entrambe si sono rivelate determinanti per il buon esito della tesi, nonostante le problematiche legate alla distanza.

Ringrazio la mia famiglia per il loro affetto incondizionato verso di me, sacrificandosi e sostenendomi particolarmente in questi 3 anni.

Un ringraziamento speciale lo rivolgo ai più cari amici che l'esperienza universitaria mi ha regalato: Daniele e Alessio. Entrambi hanno contribuito ad alcuni miei cambiamenti profondi e lasciato un'impronta indelebile nella mia vita. Senza di loro, non sarei chi sono oggi.

Infine, ringrazio amici e colleghi che mi hanno dedicato del tempo, aiutandomi o regalandomi emozioni in questi tre, sfortunatamente corti, anni.

Antonio Strippoli