
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# DESCRIPTOR FOR IDENTIFICATION OF A PERSON BY THE FACE

**DESKRIPTOR PRO IDENTIFIKACI OSOBY PODLE OBLIČEJE**

**MASTER'S THESIS**
**DIPLOMOVÁ PRÁCE**


**AUTHOR** **Bc. TOMÁŠ COUFAL**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. TOMÁŠ GOLDMANN**
**VEDOUCÍ PRÁCE**



**BRNO 2019**

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

# Zadání diplomové práce


Student: **Coufal Tomáš, Bc.**
Program: Informační technologie Obor: Bioinformatika a biocomputing
Název: **Deskriptor pro identifikaci osoby podle obličeje**
**Descriptor for Identification of a Person by the Face**
Kategorie: Zpracování obrazu
Zadání:

1. Nastudujte a sumarizujte základní principy identifikace osob na základě snímku obličeje. Zaměřte se především na metriky, které se prakticky používají pro identifikaci osob v datech z CCTV systémů.
2. Seznamte se se základními principy neuronových síti. Dále se zaměřte na konvoluční neuronové sítě a shrňte významné frameworky používané pro usnadnění implementace konvolučních neuronových sítí.
3. Navrhněte algoritmus pro extrakci biometrických příznaků ze snímků obličeje, tzv. deskriptor. Předpokládejte, že natočení hlavy dané osoby může být různé napříč snímky. Výsledný deskriptor by měl obsahovat příznaky i pro nefrontální orientace obličeje.
4. Vytvořte aplikaci, která bude provádět vytváření deskriptoru osoby na základě obličeje. Pro realizaci aplikace použijte programovací jazyk Python.
5. Vytvořené řešení otestujte a proveďte experimenty na datové sadě s lidmi různých etnik.

Literatura:

- Reid P., Biometrics for Network Security. Prentice Hall Professional, 2004. ISBN 0-13-101549-4.
- JAIN, Anil K.; LI, Stan Z. Handbook of face recognition. New York: springer, 2011. ISBN 978-0-85729-932-1.

Při obhajobě semestrální části projektu je požadováno:

- Body č. 1 a 2

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Goldmann Tomáš, Ing.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 1. listopadu 2018

# Abstract

Thesis provides an overview and discussion of current findings in the field of biometrics. In particular, it focuses on facial recognition subject. Special attention is payed to convolutional neural networks and capsule networks. Thesis then lists current approaches and state-of-the-art implementations. Based on these findings it provides insight into engineering a very own solution based of CapsNet architecture. Moreover, thesis discussed advantages and capabilitied of capsule neural networks for identification of a person by its face.



# Abstrakt

Práce shrnuje dosavadní poznatky v oboru biometrie při řešení problematiky identifikace osoby podle tváře. Zaměřuje se na konvoluční neuronové sítě a kapsulové sítě. Dále se zabývá současnými, modernímy postupy a jejich implementacemi. V neposlední řadě nabízí vlastní implementaci obdobného řešení na bázi architektury CapsNet – kapsulových neuronových sítí. Práce dále rozebírá přínosy a možnosti využití této architektry pro identifikaci podle obličeje.



# Keywords

face, recognition, identification, neural network, convolution, capsule, CapsNet

# Klíčová slova

obličej, rozpoznávání, identifikace, neuronové sítě, convolution, kapsule, CapsNet


# Reference

COUFAL, Tomáš. *Descriptor for Identification of a Person by the Face*. Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Goldmann

# Descriptor for Identification of a Person by the Face

## Declaration

Hereby I declare that this masters's thesis was prepared as an original author's work under the supervision of Ing. Tomáš Goldmann. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Tomáš Coufal
May 15, 2019

# Contents

# Chapter 1

# Introduction

It is essential to any living species to recognise others in their community. Evolution has allowed animals to develop many methods how to identify their fellow members. Different species use various senses to do so. Smell, vision, and hearing are the most common ones. Humankind relies mostly on their vision. We did not develop strong perception of smell or other senses, that we can utilise on purpose or over longer distances. Therefore we mostly use our vision which we can channel, aim and focus easily.

Moreover our species developed a habit to cover ourselves in clothes, which left only certain parts of our body exposed. Usually that parts are head and hands. And since receptors for most of our senses are placed on head, we are required to have it exposed to be able to orient in surroundings. This makes the frontal part of our head, a face, a great candidate for identifying others [26]. We can recognise many nuances of different elements on the face. Shape of eyes, nose or mouth, colour of our eyes... That's just a subset of all the available features we use.

In era of computers and information technologies, mankind tend to use what we know and offload our skills and capabilities to artificial intelligence. We hope to sharpen and enhance our senses; obtain more knowledge and widen our possibilities. As an example, which is more deeply elaborated in this thesis, we are trying to „teach" machines to recognise faces in the same way as we do. And there are various reasons to do so.

## 1.1 Motivation

Facial recognition in computer science is used in many different ways. Some use cases require people to be recognised for personal or company security purposes, for surveillance or just for our convenience.

Automating security countermeasures is a huge driving factor. It allows us to devote our time to other activities then watching over our belongings and assets. It ensures our privacy and well-being of the community [38]. Let's list some basic use cases:

- **Personal security**: unlocking a smartphone or laptop, etc.
- **Enterprise security and state defence**: access control, customs and border control, etc.
- **Surveillance**: Outlaws identification in public places, law enforcement, riot control, etc.

Furthermore, automated recognition of people's faces and their identification can facilitate many other processes for our convenience [7]. That usually means it can save us time and provide a better service. These use cases spans from face and smile detection when focusing and timing a camera shot to automated tagging of individual's friend on social media networks.

## 1.2 Justification

As in any other field in computer science, there is an ongoing race to provide better service in facial recognition. There are multiple aspects used as a metric to define better solution and this varies for each use case. Once, it is essential to provide the best performing software, for automated recognition, in places where limited computational power is available. Other time, it is rather about precision and time required to recognise a face. Outrunning other solutions means better service and inventing new technologies along the way unlocks better understanding on our own thinking process. In this thesis the existing solutions will be elaborated in detail as well as principles used to achieve such results.

There are also different approaches used to create and design such solutions. Sometimes a naive approach like edge detections and direct vectorisation [32] is used, other times the implementation leverages different aspects of Artificial Intelligence, especially Neural Networks.

## 1.3 Decomposition

To recognise a human face in a picture or video is a fairly complex problem. Therefore it is usually divided into smaller sub-tasks, which are easier to comprehend and solve. The aim of this thesis is to construct a very own solution to this problem, relying strictly on Neural Networks. Each part of the problem is explained, while only the later one is the crucial and key deliverable for this thesis. Therefore only the actual facial recognition over already located faces is implemented. Comparison with existing solutions is offered as a part of the Available solutions chapter 4, later in this thesis.

Firstly, it is required to understand the input data. This understanding provides means to fundamentally divide the problem into easily solvable sub-tasks. Let's define and describe what are the expectations and demands on the input, so we can assess plausible approaches and later understand each sub-task as a separate problem. Then we can assume and thoroughly elaborate the desired outputs and the process of reaching them.

### 1.3.1 In-the-wild pictures

As wide as the use cases are, the variety of input data is vast. Some cases depends on frontal pictures of faces, some accents the so-called *in-the-wild* aspect. This is a term specifically used for pictures captured without cooperation of the subject. A person on the picture is usually captured from an angle, lacking eye contact and staged emotion, etc. This thesis aims to cover a topic of facial recognition over CCTV data. However it can be said that any in-the-wild pictures are suitable for this implementation. The sufficient resolution and recognizability is the most important feature and requirement on the input data. The model has to be able to find a face on the picture. Therefore even if the model would rely solely on CCTV data, the data feed has to provide enough detail of the person's face.

Figure 1.1: Example of pictures in-the-wild. From left: Richard Sutton, Yoshua Bengio, Geoffrey Hinton. *Source: Steve Jurvetson (CC BY 2.0*[2]*)*

CCTV as a source of the input data has some crucial advantages for real world scenarios over any random pictures in-the-wild. Usually CCTV produces feeds [2], therefore the subject of recognition is captured on many frames of a video recording. That means, the model can be provided by many images of the same face and if properly configured and trained, it can leverage this aspect [24].

In our scenario, this data has to be simplified and preprocessed since our main interest lays in the field of identity mapping for each face. Therefore instead of a full, legit CCTV footage, we will mainly work with already located and isolated, face centred images. More detail elaboration of the actual input data in use can be found in Chapter 5.

### 1.3.2 Recognition and identification

Imagine an individual captured on a CCTV footage. To identify a person by face, we have to naturally focus the model on the face. Therefore the first step would be to detect where and if the picture captures a face. When such region is found, it is essential to detect all the facial features the models is trained to focus. Not always all features are available (the face can be partially covered, captured from an angle etc.) – the model has to adapt to such situation. Based on features detected, the model creates an normalised estimation of facial features. This normalised template should represent a frontal scan of facial features of the individual's face. When the model has access to multiple images of the face, it can produce multiple templates and combine them into one unique normalised master template, unique to the person.

### 1.3.3 Output information

Last but not least, let's define what is expected to be found in such pictures. A facial identification model aims to come up with a unique vector (face template) for each person. Unique in a sense of minimising inter-dimensionality and maximising intra-dimensionality [19]. That means the vector extracted from a picture is similar to all other vectors for the same person, as much as possible. It is also the most different to vector assigned to other

[2] https://creativecommons.org/licenses/by/2.0/deed.en

people. Naturally each picture of the same person can result into slightly different template. However this sample specific output vector is expected to be nearly identical to a summarised template for the person. This summarised template is a result of combining many output vectors for the same individual. In this thesis we will elaborate more simplistic approach which will result into comparable results, yet more convenient and better consumable output.

# Chapter 2

# Facial features

## 2.1 Biometrics

The word *Biometrics* comes from Greek words *bíos* (life) and *métron* (measure). In the modern sense of this word, it is used to describe a field in science, which focuses on identification by unique features of human body. First systematic approaches to provide metric description of a human body part dates back to mid-19th century.

### 2.1.1 History

Before we begin to drill down into details of face recognition, let's iterate back and focus on some important steps in history [26] which led researchers towards current automated technologies for recognition and identification.

#### 1858: A first systematic approach to human identification by hand

The very first recorded attempt to systematically track and identify human beings happened in India. Sir W. J. Hershel used a handprint to distinguish employees of Civil Service of India. Each employee had their hand traced on their contract so mistakes were eliminated, when they were about to receive salary.

#### 1880: Bertillonage

Alphonse Bertillon developed a model using anthropometric classification of a human being. He used physical body measurements and photographs to identify criminal offenders. This method aimed to solve a problem when the felons often tried to impersonate someone else by reporting a false name to avoid harsher sentences for repeated offenders. Bertillon stated that despite a different name, they have the same body. This system failed short in 1903 when people with the same set of measurements were objectively found.

#### 1896: Fingerprint classification system

Sir Edward Henry and Sir Francis Galton aimed to replace unreliable anthropometric classification. They came up with usage of fingerprints. Henry's employee, Azizul Haque designed a classification and storage system, so scanned fingerprints could be properly and quickly compared. The Henry Classification System was quickly adopted by many criminal justice organisations around the world.

#### 1936: Conceptual beginnings of iris-based identification

Frank Burch, an ophthalmologist, discovered unique properties of iris patterns and proposed their use to identification.

#### 1960s: Face recognition

Starting from 1960, systematic approach to face recognition is being invented. At first, manual extraction of features from a picture was required. Then, a calculated distance between landmarks and rations were used for automated comparison against records. Later, the feature extraction was pushed to more automated solution, though it still required manual intervention of marking the desired spots on the measured subject's face.

#### 1993: FERET

DARPA created and sponsored a program called *FacE REcogntion Technology (FERET)*. This encouraged competition to create face recognition algorithms and automated solutions. As a result first commercial solutions were invented and made available.

#### 1998: COOIS, a forensic DNA database initiative

Launched by FBI, the COOIS database was created. It was desired to digitally store, retrieve and search for DNA information by law enforcement agencies in the USA.

#### 2002: ISO/IEC committee for biometrics

The International Organization for Standardization established a committee for standardization of biometrics technologies which further accelerated cooperation of researchers in field of biometrics.

### 2.1.2 Challenges

Biometrics tries to enhance security by implementing automated recognition and identification systems for humans. As any computer system designed for human interaction, it faces challenges in **reliability**, **scalability**, and **usability** [37]. There is also an aspect of **confidentiality**, since it directly operates with features unique to a specific individual. Any disclosure of such sensitive information results in violation on privacy of the respected individual.

Providing greater reliability is crucial for achieving system's responsiveness. To define this aspect, we recognise two distinct metrics *inter-dimensional variability* and its counterpart called *intra-dimensional variability* or better *consistency*. The first one ensures that a biometric system applied on two different subjects, no matter how similar they are, would result into two separable profiles. The later, on the other hand, oversees an essential property of consistency: multiple processing of an individual results in the same response from the system.

Scalability aspect is a measure of system's capability to provide the same quality of service, when up-scaled to a greater number of subjects, as it behaves for a small group of people. That involves not just an engineering point of view, like resource limitations, computational power and system design, moreover a uniqueness of tracked features is in play.

On the other hand, the property of usability requires the biometrics system to be user friendly for the subjects. If a process of identification by certain set of features is too complicated and invasive for a user, it might not be viewed as an aid, and would be avoided by users [29]. This aspect has to be considered as well when such systems are designed.

## 2.2 Facial landmarks

In Biometrics, and in image processing in general, we define a term *landmark*. Biometrics specifically understands [20] this term as of a **biometric feature** [30]. It is used to describe and represent a distinct region in certain image, which resembles some important property of the studied object. Since we are mostly interested in facial recognition in this thesis, we will focus just on this part of human body.

Main facial features that are important for face recognition and later in identification of owner of that face are: *Eye, Eyebrow, Nose, Mouth, Chin, Jawline, Ear* [5].

A shape of feature, their location and topological relation to other features are the most important parts of face discovery process. Each feature has an unique shape and size. This can help to localize it and decide if the subject image is really the object we are looking for. Moreover their relation, ratio and angle later helps to identify certain individual [2]. Usually presence of a feature is required to happen in a certain part of an image. Later we will discuss, how are these features detected, and what leads to successful identification.

# Chapter 3

# Current approaches to Facial Recognition

Face detection and recognition is not a new subject to computer science. Through time there were many attempts to provide satisfactory solutions and approximations. Techniques used for face detection spans from a regular well-known image recognition via edge detection, to more advanced, precise, though computationally expensive use of neural networks. This chapter will provide a basic overview of the mainstream approaches, listing the most current state-of-the-art methods and means to the problem.

Each section in this chapter represents a family of solutions. Providing a full description, explanation, and understanding of their respective problem, would span a book of its own, hence we aim to cover only a high level walk-though. We may also revisit certain aspects in more detail later in the text, as well.

## 3.1 Convolutional neural networks

There are many kinds of neural networks, one of which is called Convolutional Network [9, p. 330], or CNN (Convolutional Neural Network). CNN specialises in processing data of known, grid-like structure. That includes for example time-series data, which can be represented as one dimensional grid of samples taken in regular intervals during a period of time. It also includes image data, represented as 2D grid of pixels. As the name suggests, CNNs employs a mathematical operation called **convolution**. In practical application, that stands for a substitution of a matrix multiplication by this linear mathematical operation - convolution. And this is done in at least one of the network's layers.

This section aims to explain what convolution is, what are the motivations behind its usage in neural networks. Later we'll describe a **pooling** operation, which is used in almost all CNNs as well.

### 3.1.1 Convolution

Convolution as a mathematical operation generally symbolise an operation on two functions of real-valued argument. Let's start with an example, paraphrased from Hinton's book [9], of such functions and demonstrate a motivation behind convolution:

Assume we have a vehicle and a laser parking sensor mounted on its front. This sensor is used to measures a distance to some object, let's say a docking station for the vehicle. And we want to park the vehicle at some precise proximity to the object. The sensor provides

a single output $p(t)$, which reads as a position of the vehicle at certain time. Both $p$ and $t$ are real-valued, which means that the sensor can provide a different output value at any instance in time.

We also have to count in that our sensor is not always fully precise and reliable, so the measurements provided may be noisy. To obtain a more relevant data – a less noisy estimate of the position against the object, we can base our measurement on an average of several data samples. Since the vehicle is moving, we have to assume that more recent outputs should provide more relevant reading. So, we can weight our average and give the more recent data points more importance. Let's define this weight function as $w(a)$, where the argument $a$ symbolize age of a measurement.

Calculating the weighted average at every moment, we can get a smoother estimation of the vehicle's position as a new function:

$$s(t) = \int p(a)w(t-a)da \tag{3.1}$$

The function $s$ is formally known as **convolution**. An asterisk symbol is usually used to denote this operation:

$$s(t) = (p * w)(t) \tag{3.2}$$

To complete the definition, we have to note that $w$ has to be a valid probability density function, which in our example has to meet a criteria $a < 0 : w(a) = 0$. That limits our function to weight only past samples, since we can's assume that our sensor can look into the future, that would be silly. That is a limitation to our use case only. In general, convolution is defined for all functions where the integral above is defined.

Convolutional neural networks use a bit different terminology, than we used in the example. The function providing data we want to consume ($p$ from the example) is simply called **input**. The other parameter to the convolution, in our example that was the weight function, is called a **kernel**. The result of convolution ($s$) of input ($p$) over a kernel ($w$) is simply called **output** or **feature map**.

Our expectation that the sensor from the example above, would provide a measurement continuously, in any instant of time, is not realistic. Time is usually discretized in digital world, therefore it is meaningful to assume the sensor is providing measurements at regular intervals. So instead of integrating over time continuum, we can define a discrete convolution as:

$$s(t) = \sum_{a=-\infty}^{\infty} p(a)w(t-a) = (p * w)(t) \tag{3.3}$$

Our example was quite simple and straightforward, usually in artificial intelligence and deep-learning the input is common to be multidimensional array (tensor) of data. Also the kernel happens to be a tensor of parameters which values are often obtained by some learning algorithm.

Finally, when processing multidimensional data, convolution is used over multiple axis at the same time. That's important for our application, because pictures, we are about to process, have more than one dimension. So in case we have an input image $I$ – a two-dimensional bitmap, our convolution should use a two-dimensional kernel $K$ as well:

$$S(x,y) = (I * K)(x,y) = \sum_i \sum_j I(m,n)K(x-i)(y-j) \tag{3.4}$$

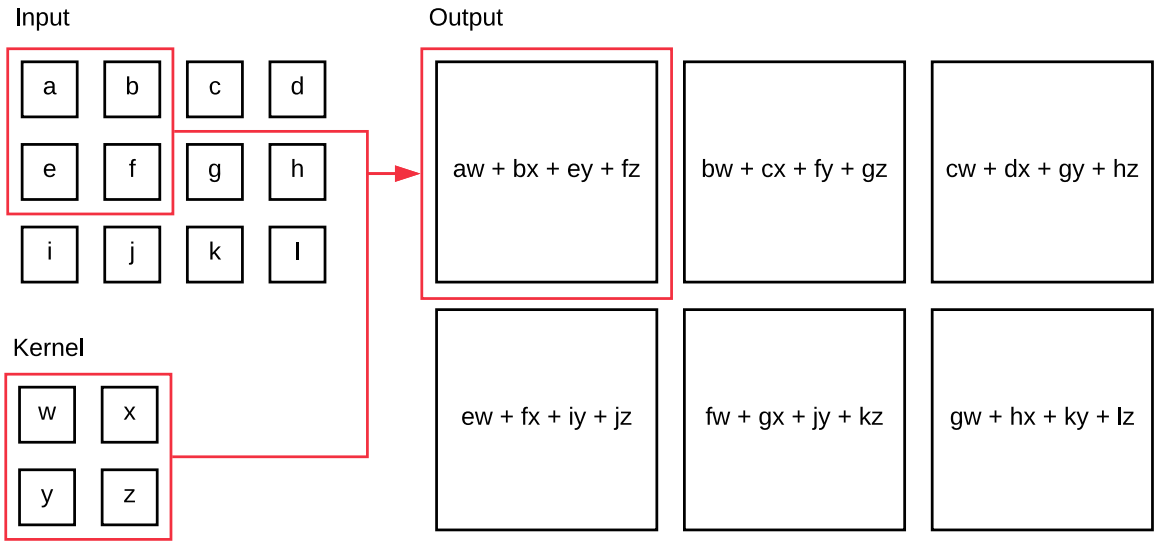


Figure 3.1: Convolution operation with 2D input data and 2D kernel

### 3.1.2 Use of convolution in neural networks

In machine learning, we leverage multiple important properties of the convolution operation:

- Sparse interaction
- Sharing of parameters
- Equivariant representation
- Input of variable size

#### Sparse interaction

As you can notice in Figure 3.1, kernel can be of different size than input. When the input data is for example an image of millions of pixels, the size of kernel allows convolution to select only specific subset of inputs for each output. In traditional neural networks, a matrix multiplication is used instead of convolution. That means every output unit interacts with every input unit. In convolutional networks, the kernel size limits that just to certain subset of input units. It is called **sparse interaction** [9, p. 335] or **sparse weights** and it is accomplished by restricting kernel to a smaller size than the input. That results in smaller memory footprint of the model, since it requires fewer parameters to be stored. while at the same time it improves statistical efficiency. It has also impact on performance, since computing the output requires fewer operations compared to matrix multiplication. A graphical demonstration can be seen on Figure 3.2.

Deep convolutional networks allow indirect interaction between units which would be out of reach for given kernel size. This property is called **receptive field** [9, p. 337] of a unit and can be seen when we look at the network from perspective of the output layer (see figure 3.3).

However, this view limits us just to direct influence on a unit. Receptive field lists also indirect influence, hence when multiple convolutional layers are used by the network the field

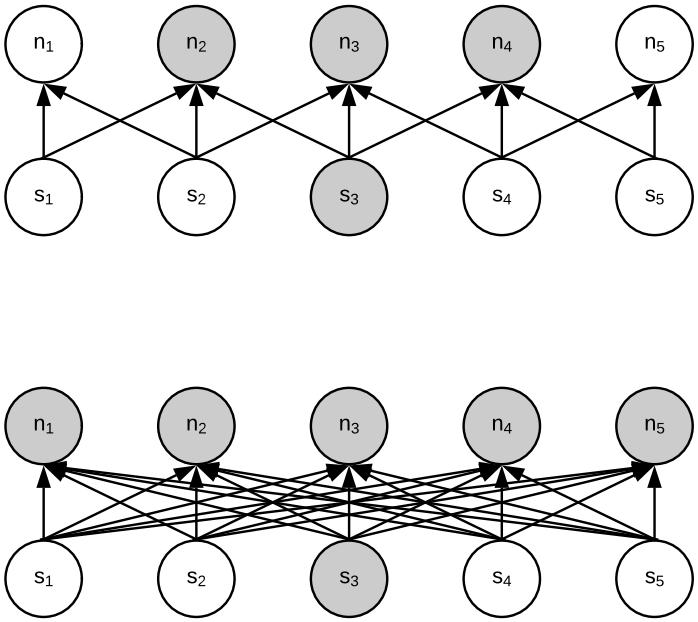


Figure 3.2: *Sparse interaction viewed from below*: The highlighted units demonstrates propagation of one input unit from current layer $s_3$ to the next layer. On the top you can see all the output $n$ units, which are affected by this particular input. On the bottom picture, you can see how the same situation is represented in traditional matrix multiplication.

grows. This can be seen on figure 3.4. The effect can be enhanced when network contains additional features like *strided convolution* 3.1.4 or *pooling* 3.1.3. This means, despite the direct influence is very sparse, the final impact through indirect influence can make the units deeper in the network connected to most of the image on input.

### Parameter sharing

A feature of convolution referring to a reuse of parameters in more than one computation. In matrix multiplication, each weight element is calculated and used only once when computing the output. The weight is multiplied by one element of the input and then never used again. In contrast, convolution keeps it is kernel the same an uses its elements for every output calculation. This brings an advantage of learning just one set of weights for the whole input, rather than computing and remembering a set of weights for each output unit. Parameter sharing has no impact on forward propagation but it does further improve memory efficiency of a stored model.

### Equivariant representation

A function is equivariant to another when $f(g(x)) = g(f(x))$. Convolution is naturally equivariant for example to translation [9, p. 339] . Imagine we have an input image and we shift the image some pixels to any direction of choice. When convolution is performed, the same kernel is applied to any set of pixels, therefore the set of features, our network layer aims to collect, can be found in the shifted image as well as in the original. It would just appear shifted in the output feature map. Here we can benefit from parameter sharing in use cases like edge detection. We're interested in the same feature, no matter where it

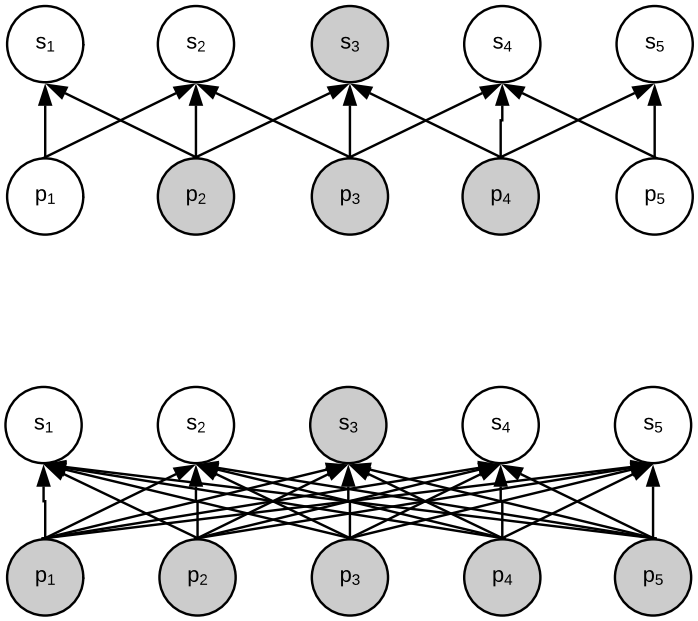


Figure 3.3: *Sparse interaction viewed from above*: Highlighted portion of the image shows all units affecting the current layer ($s$ units). The amount of units is smaller on the top picture. That represents sparse interaction. On the bottom is the same situation, when the current layer is formed by matrix multiplication in a traditional network.

appears on the image. In other use cases, like for example face detection, we might not be interested in the full parameter sharing. Imagine we have a kernel which is trained to detect a mouth. In order to work properly, we should restrict this kernel to look for the feature only in bottom portion of the picture, because detecting a mouth on forehead would not result in proper outputs. We'll cover more about multi-kernel convolution layers later.

#### Variable size of the input

Convolution can process data samples of different sizes. When the use case requires the network to be robust enough to properly process for example images, where each of the sample has different dimensions, this is a problem in matrix multiplication. The network can't apply the fixed size weight matrix on an input of different size. On the other hand, convolution is easy to perform, the situation is really similar to input of a fixed size, just the kernel is applied different amount of times.

### 3.1.3 Pooling

In neural networks, convolution layer does not mean solely a convolution operation is applied [23]. Typically such layer consists of multiple stages:

1. **Convolution stage**: Multiple parallel convolutions are computed, which produces a set of linear activations.
2. **Detector stage**: Each of the linear activations from previous stage is run through a nonlinear activation function.
3. **Pooling stage**: Further modification of the layer.

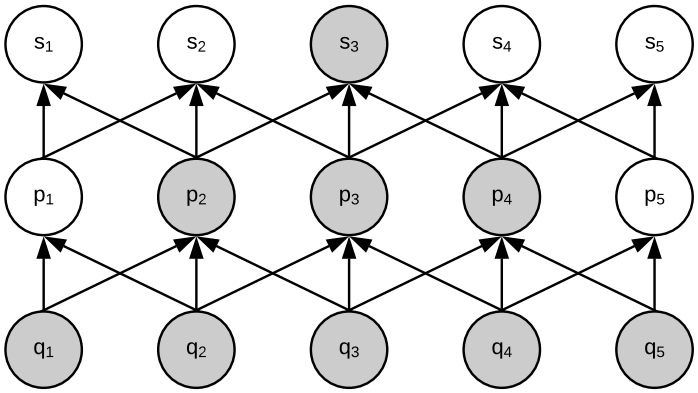


Figure 3.4: Receptive field of unit $s_3$

The last, pooling stage, provides better understanding of the convolution output. Instead of returning a set of features, it reduces redundancy of neighbouring outputs and provides a summary statistics. This helps the network to be invariant to small transition of the input. This means that if we translate the input of small amount of pixels, the output stays the same. This makes such network more robust. In a situations like a face detection, we don't need to know exact pixel coordinates of a feature, of an eye for example. We just need to define a region, where we are looking for such feature.

There are many pooling operations, let's list some of the most popular ones like **max pooling** [42] (reports the maximum output in rectangular neighbourhood), average of rectangular neighbourhood, $L^2$ norm of the rectangular neighbourhood, and weighted average of distance from the central pixel.

Invariance to transition is produced by pooling over spatial regions, however pooling layer can learn an invariance to a transformation of other kinds as well. That happens if we pool over outputs of other separately parametrized convolutions. As an example you can see an invariance to slant in cursive on figure 3.5. This principle is accented mainly in max-out networks [10].

Since pooling can summarise a response of layer over whole neighbourhood of input units, it is not necessary to have the same amount of pooling units as the detectors. We can leverage pooling to provide down-sampling as can be seen in Figure 3.6. That further improves performance of the network since it lowers the amount of inputs for the next layer.

**Pooling with down-sampling** is an essential step when dealing with input of variable size. Let's say we want to use the convolution to detect a face on images with different resolutions. We have learned our detectors to register mouth in the bottom half of the image and another two sets of detectors to locate eyes, each in one of the top quadrants. We can use convolution layer with down-sampling pooling to provide the required classification. We expect to be provided by 3 activations on the output, and each of the detector has assigned its portion of the input. It does not matter, if the portion contains one amount of pixels or much more.

In deep learning this is often the case. We don't refer to convolution as a simple single operation as described in the beginning of this chapter. Such convolution layer with a single kernel would be capable of extracting only a single feature, although in many spatial locations. Usually many convolutions are applied and performed in parallel. That can provide many different kernels for different features, which are interesting for the use case. As a result, the network can locate many kinds of features at many locations.

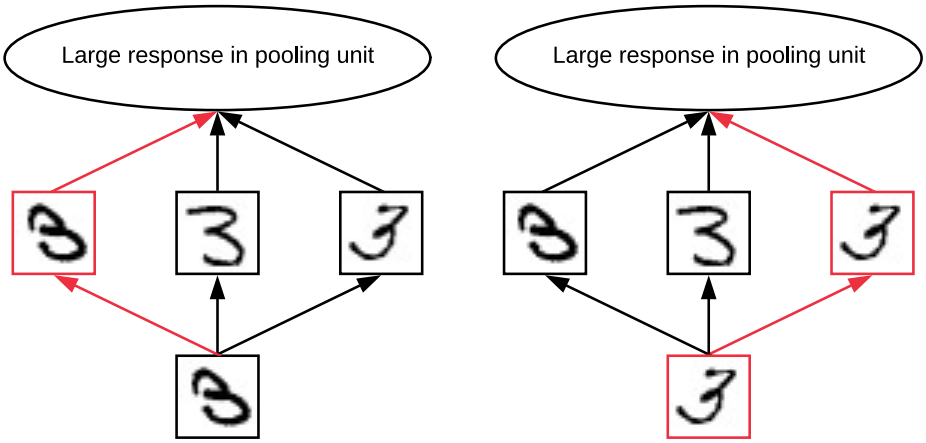


Figure 3.5: *Pooling response with learned invariance to slant*: Here we have 3 filters, where all of them had a task to learn a handwritten number 3. Each of them resulted in learning of different slant of the number. When a number 3 is given as the input, one of the filters will match it and cause a high activation in corresponding detector unit. Due to use of pooling, the max pooling unit has a large activation as well, no matter which filter matched the number.

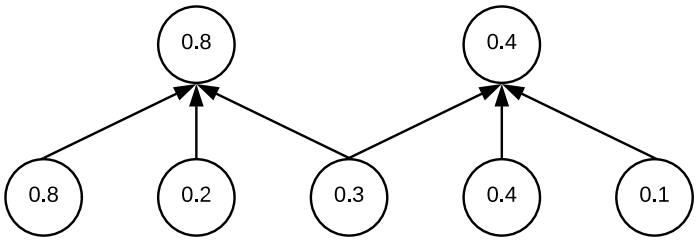


Figure 3.6: Pooling with down-sampling

### 3.1.4 Strided convolution

Also when we have stumbled upon the down-sampling in pooling, this step can be sometimes omitted and simplified even more. Although the result is similar to the pooling with down-sampling, the logic in based on different assumptions. In pooling operation, we leverage all the information retrieved by the convolutions and simplify the output.

A strided convolution is rather avoiding some of the convolutions at all. That further lowers the computational costs, hence at a risk of not extracting all the features in such detail. In this case we sample pixels in every direction with a step of $s$. The step $s$ is called a **stride**. It is also possible to define a separate stride for each step direction.

As it can be clearly seen on the images 3.7 and 3.8 the two step down-sampling is computationally more expensive than the strided optimisation, while it can provide similar results.

### 3.1.5 Zero padding

Another feature which is essential to CNNs implementation is padding with zeros. This maintains the network's ability to preserve the width of its input if needed. The input tensor is padded with zeros on the ends, so a convolution operation does not shrink the

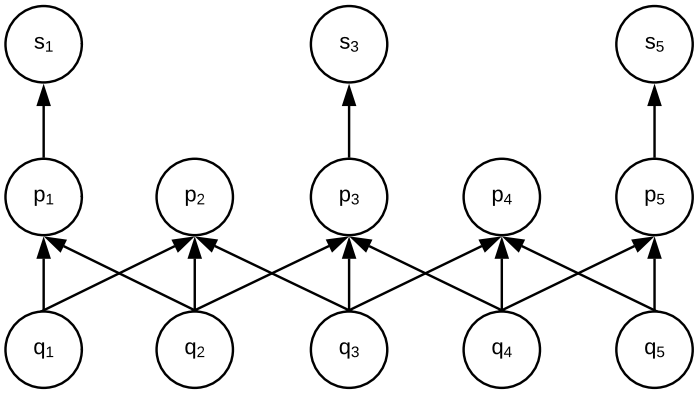


Figure 3.7: Convolution and pooling with down-sampling

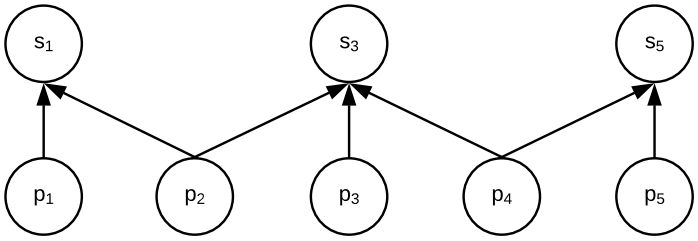


Figure 3.8: Strided convolution

size of input vector by a fraction of kernel. Without padding we are forced to either keep kernels small, or let the networks shrink in spatial extent. Both are extreme limitations of network's power. Padding allows to control independently both: the size of kernel and resolution of the output.

### 3.1.6 Other convolution layer types

Sometimes our desire is to rather use locally connected layers. This resembles a discrete convolution with small kernel, but without parameter sharing. Therefore this layer type is sometimes called **unshared convolution** [22]. Unshared type of convolutional layer is useful when we aim to detect features, which are local and there's no assumption that the same feature should occur across all the input.

Another type available is a **tiled convolution** [11] layer. A layer type meant to offer a compromise between locally connected layers and convolutional layers. Instead of learning a set of weights for every spatial location, this layer type provides tiling. That means a single set of weights is learned and it is later applied in rotation, providing different set of weighs for neighbouring locations. This makes the outcome similar to locally connected layer, while keeping the benefit of lower cost of convolutional layer, since requirements to store parameters would grow by factor of kernel set size, instead of size of a whole feature map. A comparison overview of different convolution layer types can be seen on figure 3.9.

## 3.2 Capsule neural networks

Convolutional neural networks are a state-of-the-art of current deep learning. They are hugely popular and they can provide great results and solve problems, which were unimag-

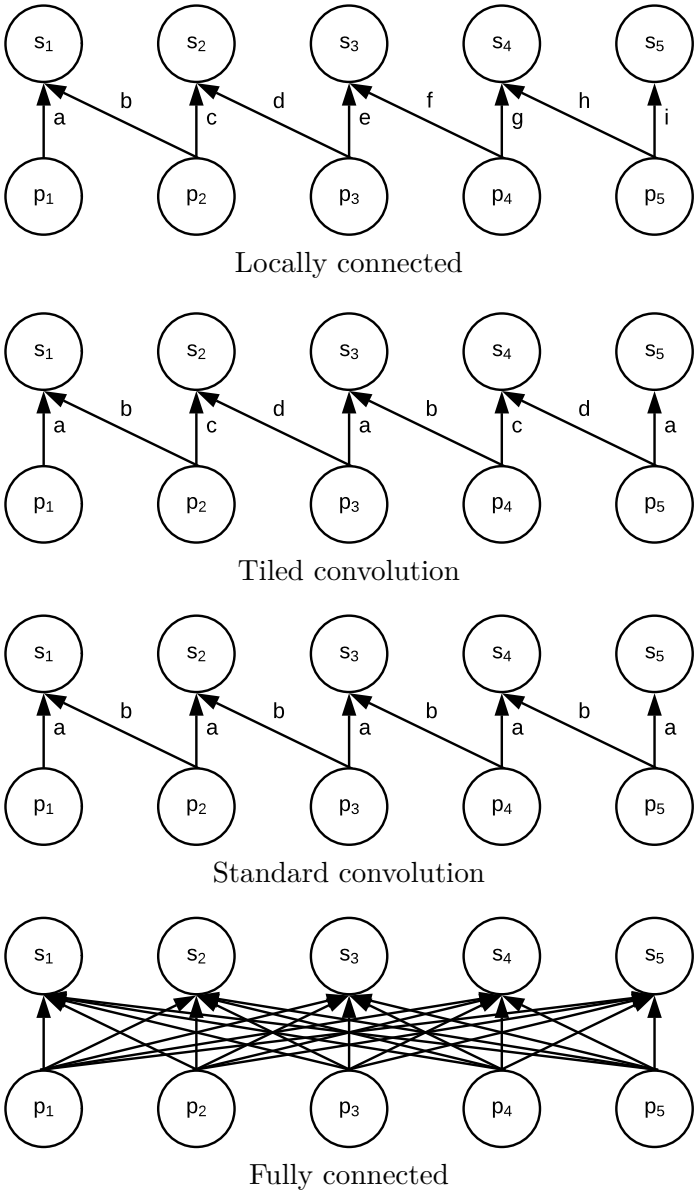


Figure 3.9: Comparison of convolutional layers

inable before. Despite all their power they embody fundamental drawbacks and limitations [16]. This and availability of grater computational power lead to creation of Capsule neural networks (CapsNET) [15].

### 3.2.1 Rationale

CNNs operate with features and their recognition. Deeper convolution layer detect simple features, for example edges or colour gradients. Layers higher in the network are designed to detect specific combinations of such features and creates more complex ones. And finally, on the very top of the network, a dense layer takes the very high level features and provides a prediction of classification. And since many times the convolution is made invariant to different transformations, this can lead to impossible results. Mere presence of an object provides indication of feature presence. Relation between detected features is not considered at all. When simple features are composed to a more complex ones, translational or rotational relationship does not play any role.

We have already tackled the way CNN is using to deal with this problem. Pooling and more convolutional layers of smaller kernels are applied to reduce spatial size of information lost during each convolution. This aims to increase the field of view for convolutional layers higher in the network, therefore allowing them to locate features in larger portion of the input image. Pooling made CNNs surprisingly effective and one of the top performing architectures, though still enhancing loss of information.

Prof. Geoffrey Hinton, one of fathers of deep learning and praised founder of many principles and author of algorithms, is also the author of capsule neural networks. Hinton wrote [1]:

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

To demonstrate this drawback imagine a CNN face detector. Deeper layers of network detects parts of facial features. The higher layers combine these parts into complex features like an *eye*, *nose* or *mouth*. Our multi-kernel convolution would allow us to define regions, where we expect such features, though we can't define the relation between them. The network can simply recognise an image as a valid face, despite for example the *eye* is rotated to the opposite direction than a *mouth*. An extreme demonstration of such problem can be seen on figure 3.10.

### 3.2.2 Inverse graphics

Hinton, inspired by computer graphics, tried to explain and reconstruct human brain's visual cognitive functions. The brain itself in fact does the opposite process to *rendering* we know from computer graphics. Hinton calls it **inverse graphics**. A visual information is decomposed into a hierarchical representation of objects, which are matched against known, learned patterns. This relationship matrix is stored in our brains. One key factor is that object representation is not dependent on view angle.

So how do we model this hierarchical relationship inside a neural network? Here we can learn from solutions already discovered in another field – in computer graphics. 3D modelling uses something called a **pose**. This represents relation between 3D objects and provides a rotation and translation transformation matrix. In neural networks we represent

[1] https://www.reddit.com/r/MachineLearning/comments/2lmo0l/ama_geoffrey_hinton/clyj4jv/

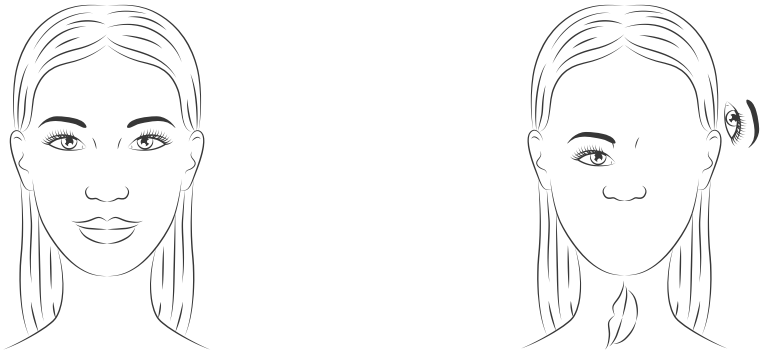

Figure 3.10: Both of these images can be evaluated to be a face. Spatial location, relation and pose between simpler features are not considered by this type of network. Both images appear similar to CNN. *Original artwork provided by Freepik [6]*

that as a 4D *pose matrix* [15]. This results in combination of information about object relations with internal representation of object data. Hence it becomes easy for a model to recognise that it just sees a different view of something it already saw before.

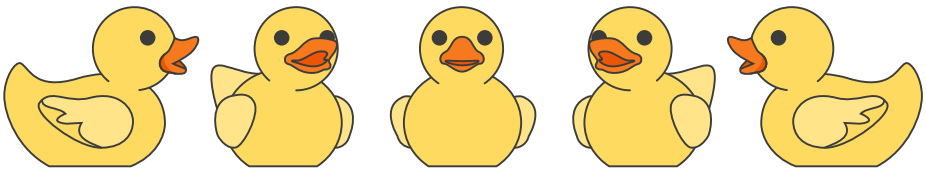

Figure 3.11: CNN struggles to recognise this is the same object. Human brain, on the other hand, immediately understands that objects on the picture are identical, despite being viewed from multiple different angles. *Original artwork provided by Freepik [6]*

Let's discuss a situation on the Figure 3.11. A human looking at this picture can easily recognise that it is a rubber duck viewed from different angles. Our internal representation of a rubber duck is independent on the viewing angle. It may be the first time that these particular pictures are shown to you, although you simply know their meaning. Since there's no internal representation of 3D space in CNNs, it really struggles. On the other hand, for CapsNET, this problem is easy, since the 3D relations are explicitly modelled. Experiments shown that usage of capsule neural networks can further reduce the error rate [16].

And this is not the only benefit of usage of capsules. It also significantly lowers the amount of data required to train such network to achieve comparable performance to a CNN. And this make sense, since capsule theory is much closer to the way how brain does work. If a brain tries to learn to distinguish a horse from a cow, it only needs to be presented with few images, at most couple of dozens. CNN would require to be presented by many thousands of image samples, to achieve comparable performance. In this particular aspect we can compare CNNs to a brute-force approach to deep learning [27].

### 3.2.3 Understanding capsules

Let's step back and introduce capsules in the CapsNet networks. What does a term *capsule* actually mean? What is the mathematical representation and what key principles are used? How we can represent it as a neural network architecture? How does a network

using capsules look like from the engineering aspect? And how should such network look like in our particular use case?

There are many ways to implement capsules and in this example we will follow Hinton [15] and use a simple, 3 layer shallow capsule network with dynamic routing. As shown later, such shallow network and straightforward implementation works well when used together with dynamic routing and can achieve results comparable to much deeper CNN networks.

Desired representation of a capsule layer output in CapsNet network represents a probabilistic likelihood of the entity mapped by one particular capsule being present in the current input. To facilitate this we use a non-linear activation called **squash** and is shown as Equation 3.5. It ensures that the length of a vector is converted into a probability score of feature presence in current input kernel for each capsule. The aim is to resize vector based on their size. Shorter vectors get shrunk to minimal lengths while long vector are shrunk to size slightly below 1. This non-linearity can be later leveraged in discriminative learning process.

$$v_j = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|} \tag{3.5}$$

In this equation a squash activation is computed for a capsule $j$ where $v_j$ would represent squashed output and $s_j$ is the total input.

In any later layer, except the first layer of capsules, the capsule can understand the $s_j$ input as a weighted sum mapping over all prediction vectors $\hat{u_j}|i$ decided by the capsule layer below. This prediction vectors are a product of $u_i$ output of previous capsule layer and its $W_{ij}$ weights matrix:

$$s_j = \sum_i c_{ij} \hat{u}_{j|i}, \quad \hat{u}_{j|i} = W_{ij} u_i \tag{3.6}$$

Here we are introducing $c_{ij}$ coupling coefficients which are determined by the dynamic routing process in multiple iterations. The $c_{ij}$ coefficients are computed for each capsule $i$ in relation to every capsule $j$ in the layer above. For each capsule $i$ the sum of these coefficients is equal to 1 and represents a special soft-max for routing. This activation uses initial logits $b_{ij}$ that are log prior probabilities [16] of the likelihood a capsule $i$ is coupled to capsule $j$. In general, routing provides each capsule $j$ a mean to determine which capsules $i$ are interesting enough and aligned in the same fashion for it to base its prediction upon. This coupling is determined independently on current input.

$$c_{ij} = \frac{exp(b_{ij})}{\sum_k exp(b_{ik})} \tag{3.7}$$

Later on we will define two distinct layer types: *prediction capsules* and *primary capsules*. These are required to communicate and rate the activations based on accuracy of the prediction. The algorithm used is called **dynamic routing by agreement** and is described in greater detail by Hinton in publication *Dynamic routing between capsules* [15]. It utilizes the already described coupling coefficients and log prior. The algorithm goes as this:

**Algorithm 1** Dynamic routing by agreement

1: **procedure** ROUTING($\hat{u}_{j|i}, r, l$)
2: **for all** capsule $i$ in layer $j$ and capsule $j$ in layer $(l+1)$ **do** $b_{ij} \leftarrow 0$
3: **for** $r$ iterations **do**
4: **for all** capsule $i$ in layer $l$ **do** $c_i \leftarrow softmax(b_i)$
5: **for all** capsule $j$ in layer $(l+1)$ **do** $s_j \leftarrow \sum_i c_{ij}\hat{u}_{j|i}$
6: **for all** capsule $j$ in layer $(l+1)$ **do** $v_j \leftarrow squash(s_j)$
7: **for all** capsule $i$ in layer $l$ and capsule $j$ in layer $(l+1)$ **do** $b_{ij} \leftarrow b_{ij} + \hat{u}_{j|i}v_j$
8: **return** $v_j$

The aim is simple, yet harder to achieve. All and every capsule in *primary capsule* and *prediction capsule* layers are made to agree on the location of the features in space and their orientation. At first all the routing logits $b_{ij}$ are initialized to zero. That means each input capsule's output is sent to capsules in the next layer with equal probability $c_{ij}$. In time, the logits are learned and since they are independent on the current image[2], they can be trained at the same time as all the other weights. Their only dependency is the type and location of the capsules involved. Its training process involves iteratively adjusting and refining the logits based on measurement of agreement between the prediction $\hat{u}_{j|i}$ made by capsule $i$ in the underlying layer and current output $v_j$ of each capsule $j$ in the layer above.

$$a_{ij} = v_j \cdot \hat{u}_{j|i} \tag{3.8}$$

The measuring of agreement is simplified to a scalar product as per equation 3.8. This rate is used to as an addend to the log prior $b_{ij}$ and used to compute new value of the coupling coefficient $c_{ij}$.

### 3.2.4 Architecture using capsules

Before we move on and utilize this principles, we need to step back and understand the capsule networks as whole. Generally speaking a Capsule network (CapsNET) consists of two parts where each has a distinct use: **Encoder** and **Decoder**:

A decoder network is a fully connected (dense) or convolutional network, which purpose is simply to reconstruct an image based on prediction. This is used to provide proper feature adaptive learning experience in order to maximize classification potential of the encoder, in other words its a regularization method for capsule networks to reach valid conclusions. Based on recognized features and selected activation in the encoder, decoder attempts to recreate the input object. In simple use cases like MNIST classification a 3 layer dense network can be used. This was demonstrated by Hinton and can be seen on figure 5.2. In more feature rich environment with bigger resolution required on input a fully connected layer doesn't provide enough interpolation and resizing capabilities so a convolutional network with resize (enlarge) has to be used. Such network can be of up to 10 layers total. In later chapter we will provide examples of such decoder networks.

Encoder is simply the gro of capsule network. This is the truly capsulized network. There are many configurations available, though with growing complexity – meaning adding capsule layers, the computational complexity grows exponentially. This is due to routing and amount of connections required between each capsule. In this thesis, we will be using

[2]Each image is expected to include one and just one face, not more.

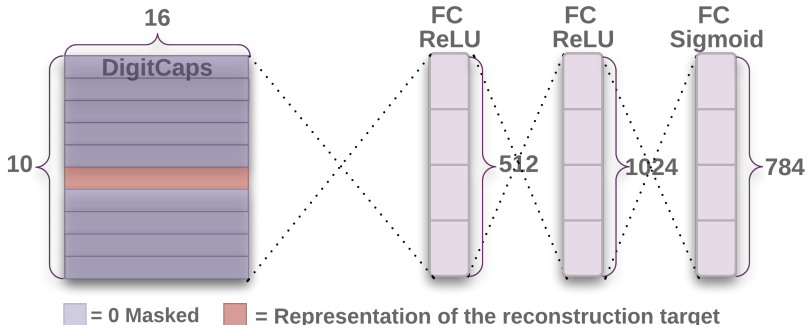


Figure 3.12: Decoder architecture as per Hinton [15]

the Hinton's hinted layering with a slight twist to facilitate more detailed and granular feature recognition across much wider space.

As said the 3 layer encoder consist of two convolutional layers (one traditional, one capsule-based) and one capsule-based fully connected layer. The first layer *encoder_conv1* is a traditional 2D convolution. Each pixel intensity of the input image gets converted in this layer to the activation in a detector of local features. Detected local features are used as input for next layer, primary capsules.

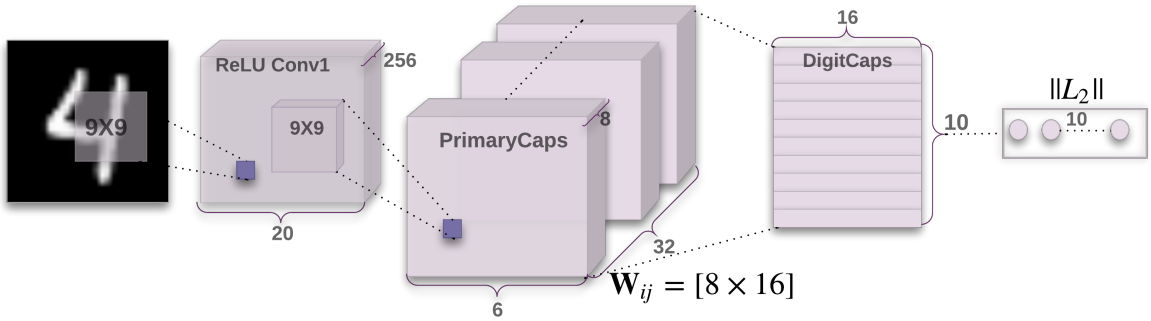


Figure 3.13: Encoder architecture as per Hinton [15]

Naturally each of these parts of the network can be assigned their own loss function. A different balance – loss weight enforced over each of these functions can greatly improve the overall loss of the network. For *decoder* a loss function would be straightforward to guess. Since we're dealing with images, 2D data comparison, for example a simple *mean squared error* is a suitable metric.

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(Y_i - \hat{Y}_i)^2 \tag{3.9}$$

On the other hand, the encoder part required more sophisticated mean to compute its inconsistency between the predicted value and its ground truth. Therefore a *margin loss* is used for each label in which we're classifying the images. For each of them and individual loss is calculated and the total loss is said to be a simple sum of these partial losses:

$$L_k = T_k max(0, m^+ - \|v_k\|)^2 + \lambda(1 - T_k)max(0, \|v_k\| - m^-)^2 \tag{3.10}$$

where, $T_k = 1$ is a hot one encoded truth of the capsule belonging to that particular label. Then $m^+ = 0.9$ and $m^- = 0.1$ are the loss boundaries and we use the down-

weighting factor of $\lambda = 0.5$ to stop shrinking in lengths of activation vectors when initial learning happens [15].

### 3.2.5 Primary capsules

This composite layer encapsulates a convolution with vectorization and *squashing*. Therefore this layer is also called *convolutional capsule layer*. It provides knowledge about greater context of each local feature, and makes more abstract detection about overall shape and orientation of such feature. Key factor in this layer is a non-linear activation called *squashing*. We've already covered the squashing function by equation 3.5. Hinton, describes this layer [15] as:

> In convolutional capsule layers each unit in a capsule is a convolutional unit. Therefore, each capsule will output a grid of vectors rather than a single output vector.

The quoted statement stipulates boundaries by which this layer is different to a standard convolution. Since we're not interested in a scalar feature identity, and require more complex result, a whole vector of attributes that holds much broader context of the feature location. This brings couple of limitations of the convolution itself though. For example a strided convolution or pooling would mean disaster for the spatial collocation of a feature. Therefore we eliminate our use of these advanced convolutional optimizations and we are obliged to use a convolution in its most simple and pure form.

### 3.2.6 Prediction capsules

Next layer is understood to be a fully connected capsule layer, which purpose is to rate the features located by the previous *primary capsule* layer and bundle the smallest amount of unique activation per label. As it may sound confusing, we need to elaborate further and step by step. We already know, that the output of previous layer in each of its capsules is a local grid of vectors which are unique to each its member as well as for each capsule. Key word is „grid". It signifies the output is three dimensional. However traditionally a feature is understood as a 2D location. Our primary capsules holds also another interesting factor to this. We can now understand and recognize a feature in relation to others. And that's precisely what is done in this layer. The amount of prediction capsules corresponds to the amount of target labels. Therefore each capsule is mapped to one particular label, in our case an identity, and is expected to locate the features which are unique to this particular identity as well as all the common features which define a face. However this is not an easy task, and a simple weighted multiplication over a kernel would not provide enough insight into the complicated data. Therefore an algorithm called *dynamic routing* is used. This routing algorithm has been already discussed and can be seen as Algorith 1.

The main purpose is to recognise that particular features which holds the same or similar context as the others in each particular label. The context can be of many different types, from simple distance of multiple features in 3D space, over a consistent angle hold between two features, to for instance a simple contextualized location relative to other features.

# Chapter 4

# Available solutions

Before implementing an own solution, let's uncover some available and current implementations to this very problem. Naturally, big progress and research is performed in this topic, therefore this chapter is not meant to provide full market scan, or list every solution available. It rather aims to discover some fine examples of neural network implementations which perform well.

Later on you'll stumble upon a list of available face recognition databases and data sets, which can be used for training and evaluating our implementation. Each of the data set description is accompanied by a short list of their respective metrics.

## 4.1 Existing CNN implementations

There are many available solution and existing implementations [39]. They naturally differs, their models compete and cover different use cases [23]. We have already established, that the face recognition is a dynamic field with competitive nature. Open challenges and competitions are enabling greater progress in the field. However that makes it impossible to provide an always up-to-date review of available solutions and implementations.

Here's a short list of example challenges and competitions held in the face recognition field. It provides an insight into how wide the field is and how many different architectures can be used:

- MSCeleb challenges [1]
- Deep learning benchmark [2]
- NIST: Fusion of Face Recognition Algorithms 2018 [3]
- Surveillance Face Recognition Challenge [4]

Therefore let's rather focus on one single solution and pick some details of its design to better understand, how such implementation is made and what does it mean.

[1] https://www.msceleb.org/celeb1m/1m and https://www.msceleb.org/challenge2/2017
[2] https://github.com/u39kun/deep-learning-benchmark
[3] https://www.nist.gov/programs-projects/fusion-face-recognition-algorithms-2018

### 4.1.1 FaceNet

This state-of-the-art solution backed by Google [34] is one of the leading solutions in the field. Its implementation is open source [4] including pre-trained models. FaceNET is focused on the same kind of input data as this thesis, therefore it is really great example of an existing solution to discover. It is inspired by proposed implementations of Visual Geometry Group, University of Oxford [31].

FaceNET is a CNN based solution. It operates in multiple steps. At first it locates a face on given image by using multi-task cascaded convolutional network [41]. Then it builds a face identification network using multiple architectures. It provides models for **Inception ResNet** and **SqueezeNet** architectures.

### 4.1.2 ResNet

Older architecture of CNN provides reliable solution for image classification [35]. This architecture is mainly accented in Google AI workshop. It is a certainly complex architecture 4.1 producing models of many parameters and great size, therefore it might not be very useful in use cases with limited resources.

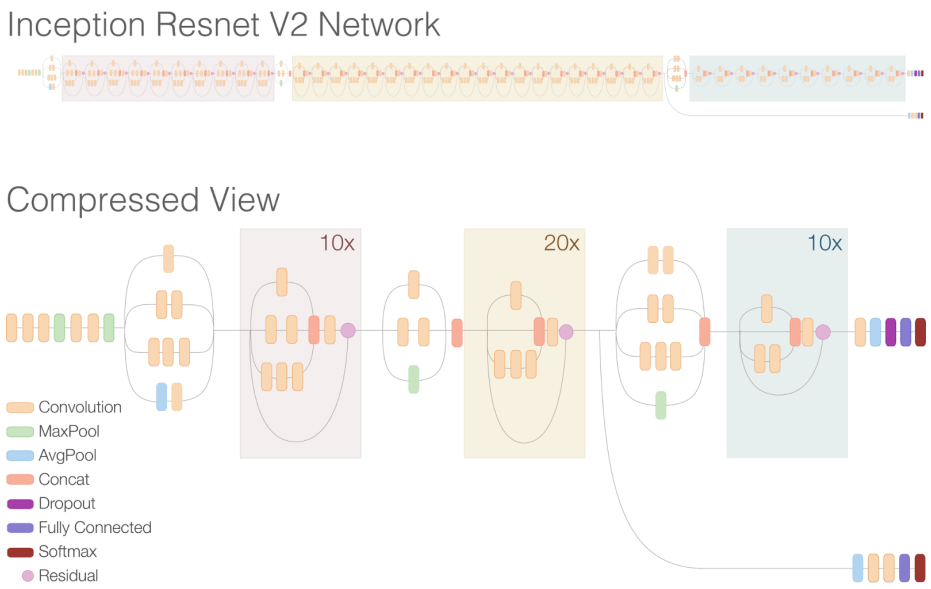


Figure 4.1: ResNet model structure. *Source: Google AI blog* [5]

### 4.1.3 SqueezeNet

SqueezeNet [18] is a totally different architecture of deep neural networks than the previous mention. It introduced itself as an network with AlexNet accuracy, despite having half of the amount of parameters. It is designed to create small networks with fewer parameters than any other common network type.

Initially this network was released in 2016 as a Caffe implementation. Later it found its way into other frameworks and was hugely adopted in use cases with limited resources.

[4] https://github.com/davidsandberg/facenet
[5] https://ai.googleblog.com/2016/08/improving-inception-and-image.html

Therefore it makes applications, which would require a model to be run on low-powered processing platforms like FPGAs and smartphones, available.

It is capable of optimizations like *Deep Compression* [14], which allows trained models to be compressed. In case of SqueezeNet it can reduce model size 10×.

## 4.2 Experimental implementations of CapsNet

Convolutional neural networks are already heavily established in the field, and since *CapsNet* is a new and emerging architecture, it haven't proved itself in this particular field [27]. Couple of different implementations exists, though only few are aiming for the facial recognition. However this architecture is the key delivery for this thesis, so it would be beneficial at least list the solutions and attempts to provide a reader more comprehensive view.

### 4.2.1 CapsNet4Faces

The only attempt available at the time of publishing this thesis, which aims to solve the same problem as this thesis does. However the implementation [33] seems incomplete and is based on the later mentioned *Traffic sign classifier* 4.2.2. The basic description states 100% accuracy although I was not able to replicate that and moreover all listed example Jupyter notebooks are faulty and do not contain expected outputs. In many places outdated results from Thibault Neveu's traffic sign classifier is shown. Despite the fact this solution is most likely not complete, it has to be listed, since it's the only publicly available solution using the same architecture found. This solution is featured in an unpublished paper[6] with questionable conclusion since the described architecture doesn't match their very own implementation and rather follows a MNIST classifier demonstrated by Hinton [15].

### 4.2.2 CapsNet – Traffic sign classifier

The most complete implementation of capsule network on a more complex problem than a MNIST data set. In this particular implementation [28], the author Thibault Neveu focuses to create a classifier of German traffic signs. A network of 42 prediction capsules is able to assign correct label to a traffic sign with 98% validation accuracy (97% for testing). It follows similar architecture pattern as described above, at least for the *encoder* part. The decoder is made differently, since a fully connected 3 layer decoding is not considered helpful for a image data in colour. Instead a convolutional decoder with resizing – up-scaling to nearest neighbour is used to provide reconstruction network. As per the documentation, this proved much more useful along with reduction in routing. The network is implemented in pure *TensorFlow* framework. In our own solution later, we will use this network as a guidance and great inspiration for our work. The implementation is not very clean, though provide great insight into how one can manipulate and classify RGB data with emphasis on small feature size. We will borrow some visualization methods later in our example notebooks as well.

### 4.2.3 CapsNet-Keras

A thoughtful implementation [13] of a capsule network in *Keras* using *TensorFlow* as backend. Despite it sole focus on MNIST data set, it provides the most understandable and

[6] https://lindawangg.github.io/projects/capsnet.pdf

cleanest implementation of such network. As we will talk about later the chosen framework is a great advantage here. It follows very closely the implementation accented by Hinton [15]. It features a 3 layer encoder with one composite primary capsules layer and one layer of prediction capsules, each per a digit. The architecture shown on figures 3.13 and 5.2 applies. As you may already recognize, that means this network has 10 prediction capsules, which greatly affects its size and keep it fairly small.

## 4.3 Frameworks

Each of the previously mentioned solutions has something in common. They have used a AI framework or library of some sort. Writing neural networks from scratch is obviously a complex task, therefore there are many initiatives, which seeks to simplify and facilitate access to such complex structures. Let's list some of the most common ones and list some of their basic advantages and disadvantages.

### 4.3.1 Torch and PyTorch

Torch is a deep-learning and computational framework written in Lua. While very powerful, its design prevented from being adopted by users and researchers. The main problem was in usage of rather an exotic programming language, which created barrier for users. It is been decided to create a Python clone of the framework, which brought PyTorch [7] to existence. It was created by Facebook and released in 2007 under an open source license. One of its key features are dynamic computation graphs, which can serve well when processing inputs or outputs of variable length.

+ Many pluggable modules
+ Easy to integrate different extensions
+ Simple to define own layers
+ Straightforward access to running code on GPU
+ Offers dynamic computation graphs
+ Broad community and wide audience
+ Easier to inspect and monitor training of models
+ Intuitive API
− Requires custom training code

### 4.3.2 TensorFlow

This library created by Google was designed as a replacement for their previous project called Theano. TensorFlow [8] is a heavyweight framework written as a Python API in C/C++. It simplifies researcher's task in many ways better than other framework. For example it generates a computational graph and performs automatic differentiation. That

[7] https://pytorch.org/
[8] https://www.tensorflow.org/

means the user is not required to write a training code (back-propagation) every time, he's experimenting with a new network topology. Since this framework is backed by Google, it thrives in many different applications and can scale across devices. Many possibilities for applying saved models in different environment enables use of AI in mobile devices and even in web browsers. However it is broad possibilities and options make this framework hard to understand and for newcomers it can be confusing and too complex. Therefore an abstraction layer above TensorFlow had been created, but more about that in the *Keras* subsection below.

+ Native Python and Numpy integration
+ Automatic training code
+ Broad community and wide audience
− Heavyweight frameworks
− A bit slower than PyTorch

### 4.3.3 Caffe and Caffe2

Caffe is another competition framework, which is widely popular among researchers. It started as a C/C++ port of Matlab's implementation of fast convolutional networks. It is mainly oriented on feed forward networks and image processing and is not intended for other deep learning application like text processing or 1D series data. Later on it became performance wise obsolete and community of Caffe developers decided to start from scratch and created a long-awaited successor Caffe2 [9]. Backed by Facebook, as their second deep learning toolkit after PyTorch, it provides more lightweight and scalable solution than before. It is main area of focus is enterprise grade production environments.

+ Great for image processing and feed-forward networks
+ Automatic training code
+ Lightweight
+ BSD license

### 4.3.4 Keras

A modern abstraction layer above TensorFlow. Authors and users of TensorFlow suffered from heavy and complex code structures and when PyTorch appeared with their light and straightforward Python API, they have started adopting the same principles for the TensorFlow as well. Therefore a project named Keras [10] was created. It provides intuitive API inspired by Torch and while starting from TensorFlow it outgrown this base and spread across many deep learning libraries as its back-ends - Theano, Deeplearning4j, and CNTK. In addition to its high level abstraction over the back-ends, Keras also provides means to drill down and optimize and manipulate the underlying code.

+ Intuitive API

[9] https://caffe2.ai/
[10] https://keras.io/

+ Multiple back ends to choose from
+ Lightweight
+ Fast growing community
+ Recognized as a standard Python API for neural networks

## 4.4 Data Sets

In order to better understand the nature of CCTV imagery, pictures in-the-wild and the source data we are about to work with, let's describe some commonly used databases of face images and face recognition data. It is crucial to understand the variety and differences between subjects captured on sample images, like their age, sex, and ethnicity. Also we need to pay attention to circumstances of the photo setup. That means for example consistency of resolution across samples, variety of poses and angles, etc.

### 4.4.1 FDDB: Face Detection Data Set and Benchmark

This data set provides annotations for Faces in the Wild [1] database. FDDB [21] lists coordinates for bounding boxes for over 5 thousands faces located on pictures from Faces in the Wild database. Usually multiple faces are located on a single picture. This data set can provide ground for face detection algorithms and therefore it can be benefited from in the first step of our implementation. For recognition of individuals, whom such face belongs to, another data set has to be used. A great accompanying data set can be the LFW, mentioned in next subsection.

| | |
|---|---|
| Number of subjects | 5171 |
| Total images | 28 045 |
| Samples per subject | Varies, many have just one, others up to 40 |
| Resolution | All kinds, even blurred faces |
| License | Creative Commons |

Table 4.1: FDDB data set metrics

### 4.4.2 LFW: Labeled Faces in the Wild

LFW [17] provides labels for images from Faces in the Wild data set mentioned before. Therefore when used in conjunction with FDDB, this data set can provide a robust base for face recognition. The database spans many identities, though it lacks volume – many subjects have only one image in the data set. That does not provide enough coverage to train a network to recognize that individual.

### 4.4.3 The Extended Yale Face Database B

Extended version of original Yale Face Database. The extension was provided by UCSD [8]. This database comprises of over 16 000 images of 28 unique subjects. They are fitted to same size and resolution, covering various angles of the face. There are 9 poses provided for each person, each of them covering 64 different illumination condition. When compared to

| | |
|---|---|
| Number of subjects | 5749 |
| Total images | 13 233 |
| Samples per subject | Varies, many have just one, others up to 40 |
| Resolution | 250 × 250 |
| License | Creative Commons |

Table 4.2: LWF data set metrics

a large scale data set this database lacks volume, however it maintains consistency across its samples.

| | |
|---|---|
| Number of subjects | 28 |
| Total images | 16 128 |
| Samples per subject | 576 |
| Poses | 9 |
| Resolution | 168 × 192 pixels |
| License | Free to use for research purposes |

Table 4.3: Extended Yale Face Database B metrics

### 4.4.4 SCface - Surveillance Cameras Face Database

A face database originated from University of Zagreb. Quality data set of surveillance-like face images. It aims to simulate a CCTV captured images by maintaining an uncontrolled indoor environment. Each of 130 subjects is captured by up to 8 video surveillance cameras. Some of them even capable of IR capturing. Each camera produces images of different resolution and sharpness. Cameras are also set in different angles against the subject. SCFace [12] mimics real-world circumstances and use cases of CCTV, therefore this data set can be used to train robust solutions for face recognition targeting CCTV and surveillance cameras.

A disadvantage is the size of this data set, where we can find 4160 image samples only. When compared to large-scaled data set like the VGGFace and VGGFace2 mentioned in next subsection, this data set lacks volume. Also the variety of subjects is not robust enough in comparison to other data set. As said, SCFace captures 130 subjects. Most of them are of the same sex, all of the same ethnicity.

| | |
|---|---|
| Number of subjects | 130 |
| Total images | 4160 |
| Samples per subject | Fixed amount of 32 images per person |
| Resolution | Varies, 3 different sizes |
| License | Custom, research purpose only |

Table 4.4: SCFace data set metrics

This database is available for research purposes and upon written request to the authors.

### 4.4.5 VGGFace2

Visual Geometry Group produced a second iteration of their face recognition data set [3]. This is one of the widest data sets which are publicly available. It provides a wide-scale data for face recognition for over 9000 different identities. Distribution of individuals varies though, with minimal 87 images up to 843 per identity. Average number of images per subject is 362. The data set contains over 3.3 million of images in total. Subjects varies in ethnicity, age and profession, while the images varies in angles or poses. Collection of such vast amount of images is a product of web scraping, so the images might not always be consistent with definition of pictures in-the-wild.

The data set is made available under Creative Commons license [11], therefore it is available for broad use to any project.

| | |
|---|---|
| Number of subjects | 9294 |
| Total images | 3 311 286 |
| Samples per subject | Varies, 87–843 per subject |
| Resolution | Varies, many different sizes |
| License | Creative Commons |

Table 4.5: VGGFace2 data set metrics

This project also provides sample models trained on their data set. However, the provided example pre-trained neural network models are not the sole representation of its usage. Many popular face recognition models are trained on this data set. For example FaceNET, mentioned before 4.1.1, can be seen as one of the popular projects which benefits from this data set.

### 4.4.6 MSCeleb

This data set is provided by Microsoft company and various challenges and competitions were held against it. Similarly to the previous one, MSCeleb [40] is a large scale data set, though oriented specifically on celebrities. Each challenge announced by their researcher team is backed by a specific subset of the data set. These selections are usually oriented on certain aspects of face recognition, therefore can be proven valid for use case covered in this thesis.

Subjects vary in all desired aspects and images provide enough variety in poses and background noise.

| | |
|---|---|
| Number of subjects | 99 892 |
| Total images | 8 456 240 |
| Samples per subject | Varies, average 85 per entity |
| Resolution | Varies, up to 300 × 300 |
| License | Research purposes only |

Table 4.6: MSCeleb data set metrics

[11] https://creativecommons.org/licenses/by-sa/4.0/

### 4.4.7 CelebA

CelebA [25] is another large scale database of faces. The focus is on celebrities faces, the same as in the MSCeleb data set. Pictures cover wide variety of poses and background noises. Each image is provided with 5 landmarks locations and 40 binary attributes.

Subjects vary in ethnicity, age, sex as well as in appearance. Data set includes faces with facial hair, poses and with different emotions.

| | |
|---|---|
| Number of subjects | 10 177 |
| Total images | 202 599 |
| Samples per subject | Varies |
| Resolution | Varies |
| License | Research purposes only |

Table 4.7: CelebA data set metrics

### 4.4.8 Aligned Face Dataset from Pinterest

Aligned Face Dataset from Pinterest (PINS) is a custom data set available on Kaggle [36]. It provides normalized, aligned and feature centred image data collected from Pinterest and processed via *dlib*. The image distribution is not strictly uniform, although it provides at least 100 images per identity. The only drawback is that images listed are not always in-the-wild. Many times it include stages photos, many of them post processed and modified. Therefore despite interesting metrics, the data set proved unusable because of it great great inconsistency within each identity domain.

| | |
|---|---|
| Number of subjects | 100 |
| Total images | 10 770 |
| Samples per subject | Varies |
| Resolution | Varies |
| License | Research purposes only |

Table 4.8: PINS data set metrics

# Chapter 5

# Implementation

Implementation chapter is meant to cover the actual experimentation and programming part of this thesis. Reader of this document is lead through series examples and is taken on journey to a reliable solution of the matter. On following pages you will find and reveal complexity of this problem. This chapter aims to show and uncover every detail the author stumbled upon when he tried to implement the solutions proposed by Hinton [15], which we elaborated in detail in Chapter 3.

## 5.1 Preparation and prerequisites

*Keras* with *TensorFlow* back-end was selected as the key framework to use for this implementation. That inherently means, we are bound to use Python as a programming language. However, selection of Python is natural and reasonable anyways, since it is the most used language in the field of machine learning and artificial intelligence experimentation. Moreover due to technical limitation and proven better performance, Anacoda Python distribution is selected as the proper back-end. According to various researches[1] and projects, *TensorFlow* performance fluctuates a lot since the pre-compiled packages are not allowed to use all the capabilities of each and every specific hardware combination. Therefore projects like Thoth[2] were created to provide dependency mesh mapping. In our example we can be satisfied by the enhanced performance of *Anaconda/Conda TensorFlow* distribution (from either Anaconda or Intel channels). Moreover, running *TensorFlow* locally on CPU is used for quick prototyping. For more demanding executions, Google Colab[3] is selected as Jupyter notebook execution provider. All source codes to this implementation were released under Apache 2.0 license on Git Hub[4].

## 5.2 Keras on TensorFlow

*Keras* is a high level API for machine learning. It provides unified means to define and access models and layers as well as most common mathematical principles and functions in a highly polished package. This package relies on a back-end provider to implement the solutions behind the scenes. *TensorFlow* is one of these back-end. Principles of this

[1] Stop Installing Tensorflow using pip for performance sake! https://medium.com/p/5854f9d9eb0c
[2] http://thoth-station.ninja
[3] https://colab.research.google.com
[4] https://github.com/tumido/capsnet-face

cooperation and more elaborate description of other back-ends and solutions can be found in Chapter 4.

Let's provide a basic overview of the available API and bindings that will be used later on in our implementation:

```
from keras import models, layers

# Procedural model declaration
model = models.Sequential(
    name="sequential_model",
    layers=[
        layers.Dense(...),
        layers.Conv2D(...),
        ...
    ]
)

# Functional model definition
input_layer = layers.Input(shape=(...))
output = layers.Dense(...)(input_layer)
output = layers.Conv2D(...)(output)
...
final_layer = layers.Conv2D(...)(output)

model = models.Model(
    name="raw_model",
    inputs=[x]
    outputs=[final_layer]
)
```

Listing 5.1: Keras example

As you can see using *Keras* is straightforward. It allows an easy model definition using various approaches. This provides a great benefit, which we will use later on – it allows model stacking, permutation of layers, combination of layouts while easily sharing parameters and behaviour. This gives the researcher a really powerful mean to manipulate a model and shape it to achieve desired behaviour, even multiple distinct behaviours for each phase of the model's life cycle. It allows easy way to inject or extract auxiliary inputs and outputs into the model configuration. This is a really important feature especially in our case, as you will learn later on next pages, since we desire a much different model behaviour when the network is trained to the one when we ask for a prediction.

In next few paragraphs we will show what network configuration we chose for our CapsNet as well as observe proven and experimental configurations of the layers involved. As you might know, layers are the basic building blocks of neural networks. And stacking these layers one on top of another creates more complex behaviours. Certain patterns of layers usually results into an architecture. Models can use a straight, classic scheme of layering one layer onto one another layer, or it can diverge at certain point and result into multiple behaviours. The first one uses `keras.models.Sequential` type of neural network. This means a single set of inputs is passed to the model, the model processes

the data through each and every layer in the same order, and at the end, the last layer in the sequence, produces the desired output. The later mentioned behaviour, required more complex yet precise handling. This allows the researcher much greater control over what inputs are passed to which layer, and which outputs are collected. *Keras* allows this type of modelling via `keras.models.Model` class.

Later on, we will find out that even combination of these methods are possible. This allows to inject and join models, reuse a model in multiple parts of the network and most importantly it allows sharing of trained parameters. In our case, we will later on describe *Encoder* and *Decoder* logic as separate models or network prototypes which are combined into a greater models for two distinct purposes, training the network and testing. The trained network has one configuration, while the model used for prediction consists of partially different layers. But since this reuse of network parts is possible, we can leverage the trained parameters from the training phase, and use them in a different network which provides predictions.

### 5.2.1 Layers

In next few paragraphs we will introduce all the layers used in our solution and get familiar with their respective API first before we start putting them together into an actual network model configuration. Just before we do that, let's describe the common API for all layers:

```
layers.Layer(
    name="layer_name",  # Allows to name layer for proper storing
    input_shape=,       # Required input shape
    output_shape=,      # Required output shape
    trainable=,         # Can make the layer static
    weights=,           # Preset weights
    ...
)
```

Listing 5.2: Common layer API

Most of the arguments above can be defined on the fly, omitted or abstracted. Except for one, which is really important to use, in case we desire to store the model for later use. And that's the `name` parameter. This string allows user to specify unique name within the architecture for this particular layer. And since we can share layers between models, this feature can be leveraged to load the proper weights data into a different model, despite being exported from another one. This topic will be covered more in the Section 5.6.4.

#### `keras.layers.Input`

Fundamental layer which allows to pass input data to a model. This layer has the biggest say in the shape of consumed data.

```
layers.Input(
    shape=input_shape  # A shape in a tuple format without
                       # the first batch_size dimension
)
```

Listing 5.3: Input layer

#### keras.layers.Dense

A *Dense* stands for a well known, fully connected neural networks layer. It's product can be represented as Equation 5.1, where the `act` is an activation function. This activation is performed over a element-wise multiplication of the input and a weight matrix kernel with additional bias added. Both kernel and bias are learned through training.

$$o_{ij} = act(i_{ij} \times k + b_{ij}) \tag{5.1}$$

This layer provides more extensive API, but in our implementation we will be satisfied with the basics. The example calls for each layer in later text is used directly from our *CapsNet* implementation.

```
layers.Dense(
    units=400,          # Sets dimensionality of the output
    activation='relu',  # Desired activation function
    input_dim=prediction_caps_dim * bins, # Input dimensionality
    ...
)
```

Listing 5.4: Dense layer

#### keras.layers.Conv2D

Provides a two dimensional convolution. Convolution as operation as well as its meanings were already described in Section 3.1.1. This layer type utilizes these principles in 2D space. *Keras* provides also other layer types for 1D and 3D convolution. However, the matter of our use case lays in image processing. And since images are a spatial two dimensional space, it dictates the use of `keras.layers.Conv2D`. In *Keras*, there are also other layers like `keras.layers.Convolution2D`, though this is just an alias which points to the same implementation as `keras.layers.Conv2D`.

```
layers.Conv2D(
    filters=init_conv_filters,     # Amount of filters
    kernel_size=init_conv_kernel,  # Specifies dimensions of kernel
    strides=1,                     # A number of strides to use
    padding='valid',               # Sets padding on outer borders
    activation='relu',             # Desired activation function
)
```

Listing 5.5: 2D convolution layer

**keras.layers.Dropout**

An over-fit prevention layer, which randomly sets each particular input to 0 with probability of `rate`. This is performed on each update during the training phase.

```
layers.Dropout(
    rate=.3  # Fraction of input to drop
    ...
)
```

Listing 5.6: Dropout layer

**keras.layers.Reshape**

A simple layer which allows to modify the shape of the input data. The result shape consists of `batch_size` as first dimension and `target_shape` as the rest. A special value of $-1$ can be used, which is treated as a variable, calculated, dimension.

```
layers.Reshape(
    target_shape=[-1, capsule_dim], # Desired shape on output
    ...
)
```

Listing 5.7: Reshape layer

**keras.layers.Lambda**

This is the first more complex layer. Tt might not seem so, however this layer allows to add custom behaviour. Its name is derived from lambda function, anonymous functions which are invoked in situ. The `keras.layers.Lambda` layer allows user to invoke any operation and transformation defined as a function. We use this type of layer in multiple scenarios, but for clarity we will list here just a single one – a calculation of length of each vector in the tensor.

```
def length(inputs):
    return k.sqrt(k.sum(k.square(inputs), axis=2))

layers.Lambda(
    length
    ...
)
```

Listing 5.8: Lambda layer

**keras.layers.Layer**

This abstract class serves as a base for all standard as well as all any custom layers in *Keras*. It's behaviour and shape is fully customizable. Inheritance from this class allows user to

invent and define a brand new layer, while maintaining the same API and compilation strategy as for any standard layer. While this might sound confusing the authors of *Keras* framework made it really easy to comprehend and straightforward to implement. Let's take a look:

```
class PredictionCapsule(layers.Layer):
    def __init__(self, custom_arg, **kwargs):
        """Init takes all custom parameters required for the behaviour."""
        self.custom_arg = custom_arg

        # Pass the standard params to base class
        super(PredictionCapsule, self).__init__(**kwargs)

    def get_config(self):
        """Configuration of layer, used when saving a model."""
        return dict(
            custom_arg=self.custom_arg,
            **super(PredictionCapsule, self).get_config()
        )

    def compute_output_shape(self, input):
        """Allows layer to tell output shape based on input."""
        return (None, self.custom_arg * 5)

    def build(self, input_shape):
        """Called when model is created, allows weights init."""
        assert len(input_shape) == 3, "Wrong shape"
        self.W = self.add_weight(
            shape=[1,2,3],
            name='W',
            trainable=True,
        )
        # Required to be called when done
        super(PredictionCapsule, self).build(input_shape)

    def call(self, inputs, **kwargs):
        """Custom behaviour. Needs to return agreed output shape."""
        ...
        return outputs
```

Listing 5.9: Custom layer example

## 5.3 Architecture

Now, when we understand what types of layer we have available, we can dive in and start building from these basic blocks a full *CapsNet* network. As we've already discussed in the Chapter 3, the base structure of a capsule network is not as complicated as a CNN would

be, however it requires some additional, custom entities and treatment as well. Starting from the simple to the more difficult, we are about to discover the overall architecture first and granularly enhance and dig deep into detail later.

Overall, the architecture is straightforward, but for good results it requires to be build from two separate distinct units, which when combined can be successfully trained. As per Hinton [15], we'll use the same naming conventions:

1. **Encoder** is the actual network of containing capsules. It serves for feature recognition and classification.

2. **Decoder** on the other hand is a helping force in training. It tries to reconstruct the input image based on prediction provided by the encoder.

Also, it's worth mentioning a *CapsNet* is not a *single-input, single-output* type of network. For training purposes, it consumes both the image and the label and outputs a predicted label along with a reconstructed image. This behavior changes when the network is used for predictions. At that point the neural network is expected to behave and process a single input image and provide one output for it – the predicted label. However more outputs might be required in inference, since instead of a prediction we can be more interested in a similarity vector, which might be a better fit for unknown identity description. One way or another, any of these scenarios requires a different model layout. And since it's not possible to achieve a multiple layouts with a single model, we're destined to use multiple models, each with different layer structure. So now, we need to make sure that once we train one model the other one is capable to benefit from it as well. And now it's the right time to use the layer sharing between models as described above.

## 5.4 Encoder

Encoder serves as the true capsule network. It is expected to consume an image containing a face and to produce a prediction of an identity the input belongs to. This is achieved by a series of layers as described in the Chapter 3. The structure prescribed by the following image is then implemented using layers described in this chapter.

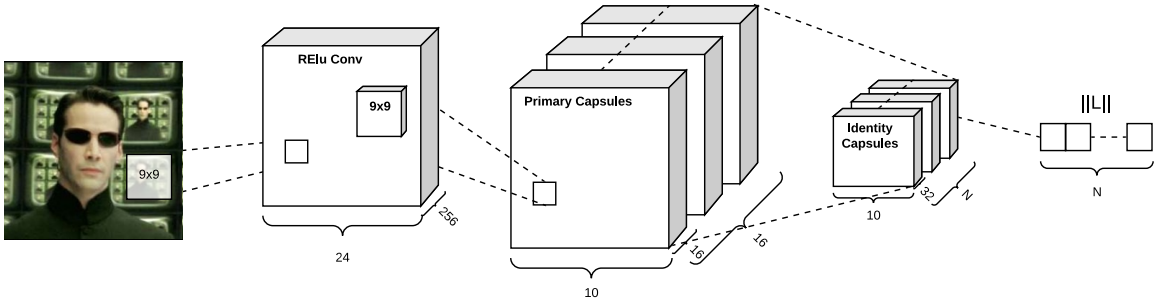


Figure 5.1: Encoder visualization

1. Input image is consumed as a

2. Then standard 2D convolution is applied

3. Convoluted output is routed towards the primary, feature capsules

4. Prediction capsules measures activation on the previous layer and provides a weight based decision of likelihood that the capsule label matches the input.

Additionally we add a *Dropout* layer in between step 1 and 2 to prevent over-fitting on over-represented labels.

As you can see, the network itself is not complicated much. Let's take look at the implementation itself:

```
x = layers.Input(name='input_image', shape=input_shape)

conv = layers.Conv2D(
    filters=init_conv_filters,
    kernel_size=init_conv_kernel,
    strides=1,
    padding='valid',
    activation='relu',
    name='encoder_conv2d'
)(x)

dropout = layers.Dropout(.3, name='encoder_dropout')(conv)

feature_caps = FeatureCapsule(
    capsule_dim=feature_caps_dim,
    channels_count=feature_caps_channels,
    kernel_size=feature_caps_kernel,
    strides=2,
    padding='valid',
    name='encoder_feature_caps'
)(dropout)

prediction_caps = PredictionCapsule(
    capsule_count=bins,
    capsule_dim=prediction_caps_dim,
    routing_iters=routing_iters,
    kernel_initializer=kernel_initializer,
    name='encoder_pred_caps'
)(feature_caps)

output = layers.Lambda(
    length,
    name='capsnet'
)(prediction_caps)
```

Listing 5.10: Features capsule with squash activation

As you can see the implementation is quite versatile and allows a lot possibilities to configure and tune it's behaviour. We have available quite few adjustable parameters and some more dependent on circumstances in which the model is deployed. We can convey the later group first and then provide explanation of the tunable parameters.

- `input_shape` is an essential attribute based on the input data. It allows the network to be prepared for input images of certain height and width. Multiple layers are directly or indirectly dependent on proper settings of this size.
- `bins` Allows researcher to set the amount of identities which we want to classify. This should also be dependent on the input data, since setting this value to different amount might result in false positive identifications.

Now to the more interesting arguments, which doesn't affect the model boundaries, but rather focus on performance and accuracy. It's worth mentioning that any of these arguments has impact on the architecture and model size, since they will change the number of trainable parameters.

- `init_conv_filters` sets how many different features should be detected in the very first convolution layer.
- `init_conv_kernel` modifies the 2D dimensions of the kernel used in the first convolution layer.
- `feature_caps_kernel` is used to adjust the dimensions of convolutional kernel in the *Primary feature capsules*.
- `feature_caps_dim` defines dimensionality of a capsule in the *Primary feature capsules* layer.
- `feature_caps_channels` is another attribute of the *Primary feature capsules*, which signifies amount of channels from which each capsule should build its grid of feature
- `prediction_caps_dim` sets the dimensionality of each capsule in *Prediction capsules* layer. This effect the amount of features which are accented - sets the maximum connection limit for each prediction capsule towards the feature capsules.
- `kernel_initializer` specifies the weight initialization in *Prediction capsules*.
- `routing_iters` allows a user to select the amount of iterations the dynamic routing should take.

Now once the scheme in general has been communicated, it's obvious that key knowledge is hidden in the implementation of the capsule layers. That would also provide explanation to the parameters mentioned above and their importance.

### 5.4.1 Primary feature capsules layer

Name of this layer and expectations set by previous chapters promise that great deal of invention is present in this layer. However that's not entirely true. More interesting behaviour can be found in the next prediction capsules layer. The primary capsules are designed for a simple feature extraction as it might be known from traditional convolutional networks. This feature extraction has a twist to it, though. As prescribed in the Chapter 3, a non linear *squash* activation is used here. We've already conveyed the Equation 3.5 describing the real nature of the squash function. Now we have the opportunity to cover this in a code.

The feature capsules layer is in it's true nature a composite layer of three layers stacked and with the squash activation on output.

1. At first a convolutional layer is present. This ensures extraction of multiple interesting patterns from each image. Patterns like hue, edges, orientation, dark spots etc. are recognized and represented by their belonging kernels. The amount of kernels corresponds to number of desired patters and dimensionality of each capsule.

2. As a next step the convoluted vector is reshaped to be bundled by capsule dimensionality. This essentially splits the input tensor into separate capsules.

3. And as the last step the *squash* activation is used to provide non-linear normalization of each vector.

```
def squash(inputs, axis=-1):
    inputs += k.epsilon()   # Avoid ZeroDivisionError
    s_norm = k.sum(k.square(inputs), axis, keepdims=True)
    scale = s_norm / (1 + s_norm) / k.sqrt(s_norm)
    return scale * inputs

# Locate features
layers.Conv2D(
    capsule_dim*channels_count,
    kernel_size,
    strides=strides,
    padding=padding,
    name='feature_capsules_conv2d'
)(inputs)

# Split into capsules (concatenate kernels for each)
outputs = layers.Reshape(
    [-1, capsule_dim],
    name='feature_capsules_reshape'
)(outputs)

# Normalize
outputs = layers.Lambda(
    squash,
    name='feature_capsules_squash'
)(outputs)

# Result tensor
output
```

Listing 5.11: Features capsule with squash activation

As you might see, there's nothing really complex happening in this layer. However when this layer is connected to prediction capsules, interesting things will start happening to properly determine the true bond between capsules.

### 5.4.2 Prediction capsules layer

In Hinton's [15] architecture suggestion, this is the final capsule layer. Intention is to classify feature capsule activations and provide routes to data from interesting feature capsules for each particular label. As this might suggest, each prediction capsule is tightly mapped and bonded to a specific label. Therefore as much labels the network aims to classify, that much prediction capsules it has to contain. This is for sure a great scaling set back and this and few other drawbacks will be discussed as a part of our conclusion. For now, let's focus on how this capsule type is implemented and how we deal with the dynamic routing as described in Chapter 3.

There are few shared properties in this layer. These are:

- `capsule_count` is a mandatory argument responsible for setting the amount of capsules available in this layer. This number corresponds to the labels we want the network to classify to.
- `capsule_dim` is a second mandatory parameter defining dimensionality of each capsule. Recommended settings is 32 or greater. Increasing any of these two parameters has a great impact on size of the network.
- `kernel_initializer` provides a string pointer or a object from `keras.initializers`. This parameter allows user to define initial values of capsule's weight matrices.
- `routing_iters` allows to change and experiment with the amount of iterations of *dynamic routing*. Default value is 3, although the best value may differ.

| | |
|---|---|
| Count of capsules | `capsule_count` |
| Dimensionality of each capsule | `capsule_dim` |
| Amount of routing iterations | `routing_iters` |
| Count of input layer capsules (feature capsules) | `input_capsule_count` |
| Dimensionality of input layer capsules | `input_capsule_dim` |
| Weight matrix | `W` |

Table 5.1: Prediction capsule layer attributes

These are the building blocks and a complete context boundaries in which a prediction capsule is defined.

1. Let's assume we have the input shape for this layer as `(None, input_capsule_count, input_capsule_dim)`.
2. To allow space for manipulation in the prediction capsule space, we need to inject a new dimension into the input tensor and tile the `capsule_count` over it. That way we can achieve a separate set of the initial input for a capsule in the current layer. Now we have shape `(None, capsule_count, input_capsule_count, input_capsule_dim)`.
3. Implementing the operation described in Equation 3.6 we multiply the weight matrix `W`

```
# Prepare inputs
# inputs == [None, input_capsule_count, input_capsule_dim]
u = k.expand_dims(inputs, 1)
# u == [None, 1, input_capsule_count, input_capsule_dim]
u = k.tile(u, (1, capsule_count, 1, 1))
# u == [None, capsule_count, input_capsule_count, input_capsule_dim]

# Perform: inputs x W by scanning on input[0]
u = tf.einsum('iabc,abdc->iabd', u, W)
# u == [None, capsule_count, input_capsule_count, capsule_dim]
```

Listing 5.12: Prediction capsule call without routing

After the initial weight propagation is done, proper activation has to be ensured. Therefore using the dynamic routing is essential. This procedure was already described as Algorithm 1, however implementation-wise in *Keras* it would look more like this:

```
# Init log prior probabilities to zeros:
b = tf.zeros(
    shape=(k.shape(inputs)[0], capsule_count, input_capsule_count, 1)
)
# b == [None, capsule_count, input_capsule_count, 1]

for i in range(routing_iters):
    with tf.variable_scope(f'routing_{i}'):
        c = tf.keras.activations.softmax(b, axis=1)
        # Perform: sum(c x u)
        # c == [None, capsule_count, input_capsule_count, 1]
        # u == [None, capsule_count, input_capsule_count, capsule_dim]
        s = tf.reduce_sum(tf.multiply(c, u), axis=2, keepdims=True)
        # s == [None, capsule_count, 1, capsule_dim]
        # Perform: squash
        v = squash(s)
        # v == [None, capsule_count, 1, capsule_dim]
        # Perform: sum(output x input)
        v_tiled = tf.tile(v, (1, 1, input_capsule_count, 1))
        b += tf.reduce_sum(
            tf.matmul(u, v_tiled, transpose_b=True),
            axis=3, keepdims=True
        )

# Squeeze the extra dim (used for manipulation, not needed on output)
# v == [None, capsule_count, 1, capsule_dim]
v = tf.squeeze(v, axis=2)
# v == [None, capsule_count, capsule_dim]
```

Listing 5.13: Prediction capsule routing

The routing eliminates the `input_capsule_count` and `input_capsule_dim` from the tensor and instead provides new dimension of `capsule_dim` which applies the attribute of current prediction capsule layer. As you can see the resulting shape of a prediction capsule classification activations per each image over each capsule dimension is the proper output shape of the layer. This would eventually tell us, which capsule found that particular image most interesting and the likelihood that it belongs to a class mapped to a capsule.

## 5.5 Decoder

Based on the findings in Chapter 3, a decoder is a sequential model meant to provide an image reconstruction feedback. The gro is to implement a training helper, which covers the key spatial feature areas of a particular identity, so we can later compare and match the input image to measure accuracy of our prediction. In case of a MNIST data-set this factor is of a great help, though in case of coherent input data with small difference on large spatial feature scale, as a human face image is, the importance of a decoder seems diminished. Moreover a simple 3 layers deep decoder comprising of fully connected layers offered by research on MNIST can't be successfully used on RGB data with granular features. Therefore we leverage a solution introduced by Thibault Neveu in his traffic signs classifier 4.2.2. We will build a convoluted reconstruction model, which essential building blocks are a dense layer, resizing layers and convolutional layers. In the end this forms a neural network of 10 layers, which provides more granular control than a 3 layer model of fully connected layers.

The decoder consumes a prediction provided by *encoder* as well as the true image and aims to recreate a image of a face based on the feature grid activations in the prediction. The result can be understood as a master template for that particular identity. We will look into that in our model evaluation later.

Now, let's describe the flow we want the layers to convey in our decoder unit:

1. `keras.layers.Dense` which normalizes and unifies the input mapping enabling later convolutions to run efficiently and select proper input activations.

2. `keras.layers.Reshape` layer converts the fully connected output from one dimensional array to a base image matrix with $5 \times 5$ pixels over 16 channels.

3. Now an alternating pattern of `keras.layers.Conv2D` and `keras.layers.Lambda` with `resize` function provides a resizing, up-scaling of the image, while interpolating its properties back to $32 \times 32$ pixels in a convoluted fashion.

4. Before we finalize the decoder, another `keras.layers.Conv2D` convolutional layer is used to reduce populated channels and provide only one best channel for each colour.

5. As the last layer we chose to chain an activation layer `keras.layers.Activation` transforming our matrices to a ReLU activated image data.

In case this is not explanatory enough, let's look at a diagram:

### 5.5.1 Masking layer

Since we want our *decoder* to provide as accurate reconstruction we need to tell it, when the predicted output matches our expectation on what capsule. So before we pass our *encoder*

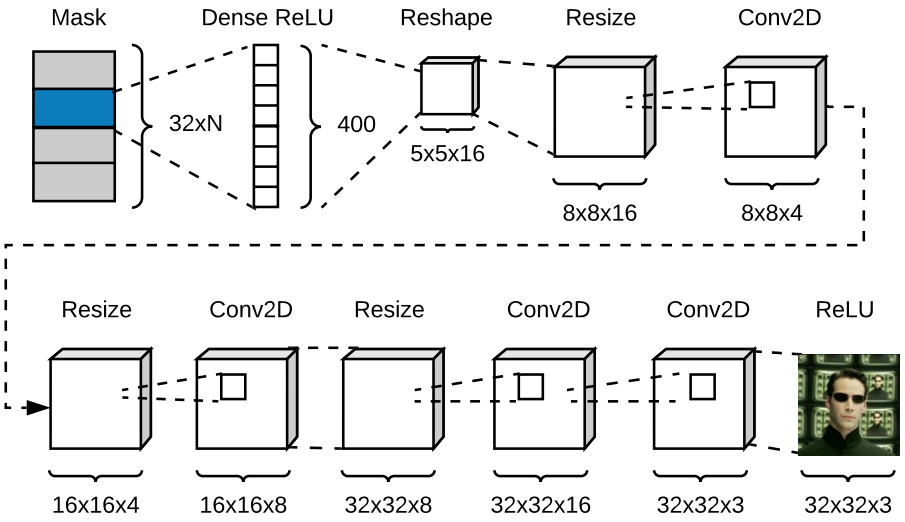


Figure 5.2: A convolutional decoder for a *CapsNet* used in our implementation.

output to the *decoder* defined above, we need to build a mechanism which would combine true labels and predicted activations in our capsules. This can be easily done by introducing an intermediate layer, which can nullify every other capsule vector than the correct one. And since we already have our labels defined as a hot one encoding, we can just simply multiply our capsule network encoder findings as a tensor for each input by the label vector. That will ensure propagation of correct capsule activations, because they happen to be on the only (hot) index, while other capsules are deactivated because their vector is multiplied by a zero, therefore they won't provide any input value for this image. Therefore we end up with set of vectors per each label in size of `capsule_dim` × `labels_count`. Despite we consume two outputs in this layer, the output shape can be computed solely from the first input – the *prediction capsule* layer shape. The dimension of true labels is shared because we keep the number of identities and amount of capsules consistent (1 to 1 mapping).

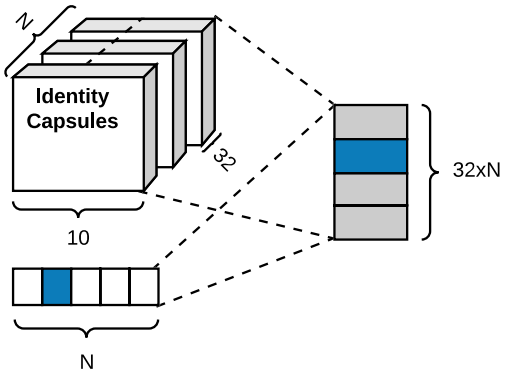


Figure 5.3: Graphical representation of the masking layer. Colo coded in blue is the reconstruction target, while in gray are the suppressed inputs.

This layer is straightforward to implement. Despite the simple calculation, it would be difficult to use `keras.layers.Lambda` since we are combining 2 inputs and calculating

the output shape out of them, therefore we take the base `keras.layers.Layer` and extend it's `call` and `compute_output_shape` methods. Since this layer doesn't require trainable weights, there's no need to overwrite the default `__init__` on `build`.

```
class Mask(layers.Layer):
    def call(self, inputs, **kwargs):
        # Unpack
        capsule_output, labels = inputs
        # A vector is multiplied by a scalar hot one encoding.
        # Therefore only one vector is kept.
        return k.batch_flatten(capsule_output * k.expand_dims(labels))

    def compute_output_shape(self, input_shape):
        # PredictionCapsule layer shape
        # input_shape[0].shape == (None, capsule_count, capsule_dim)

        # True labels shape
        # input_shape[1].shape == (None, capsule_count)

        # Calculate from PredictionCapsule shape
        return (None, input_shape[0][1] * input_shape[0][2])
```

Listing 5.14: Masking layer

### 5.5.2 Recapitulation

Now we understand the building blocks of our network. Let's recapitulate the whole architecture to better understand the complete flow, how the data will be processed and classified. We begin with an image of a human face. This image is set to be $32 \times 32$ pixels and the ground truth labels are expected to be a hot one encoding of the vectorized identity bins.

Then we can expect the *encoder* to adhere to the architecture shown on Fig 5.1. When the model is validated or we are running an inference, we just calculate a norm by length over the outputs are we have a prediction. On the other hand, when the model is being trained, we need to add a *decoder*. However, in between we can't forget about the intermediate layer encoding the true labels for reconstruction. So we plug there the *masking layer*, after which we can successfully chain the *decoder*. Let's have a look how a summary of such model would look like:

## 5.6 Life cycle of a model

Traditionally in machine learning a model needs to be trained, then it is tested and validated. Since there's no reason to change this workflow this section will follow the established scheme and walk you through the required steps in the particular order in which they need to be examined:

1. Data set selection, collection and pre-processing

| Name | Type | Output shape | Parameters | Connected to |
|---|---|---|---|---|
| input_image | Input | $(None, 32, 32, 3)$ | 0 | |
| encoder_conv2d | Conv2D | $(None, 24, 24, 512)$ | 124 928 | input_image |
| encoder_dropout | Dropout | $(None, 24, 24, 512)$ | 0 | encoder_conv2d |
| encoder_feature_caps_conv2d | Conv2D | $(None, 10, 10, 256)$ | 3 277 056 | encoder_dropout |
| encoder_feature_caps_reshape | Reshape | $(None, 1600, 16)$ | 0 | encoder_feature_caps_conv2d |
| encoder_feature_caps_squash | Lambda (squash) | $(None, 1600, 16)$ | 0 | encoder_feature_caps_reshape |
| encoder_pred_caps | PredictionCapsule | $(None, 42, 32)$ | 34 406 400 | encoder_feature_caps_squash |
| capsnet | Lambda (length) | $(None, 42)$ | 0 | encoder_pred_caps |
| input_label | Input | $(None, 42)$ | 0 | |
| mask | Mask | $(None, 1344)$ | 0 | encoder_pred_caps<br>input_label |
| decoder_dense | Dense | $(None, 400)$ | 538 000 | mask |
| decoder_reshape_1 | Reshape | $(None, 5, 5, 16)$ | 0 | decoder_dense |
| decoder_resize_1 | Lambda (resize) | $(None, 8, 8, 16)$ | 0 | decoder_reshape_1 |
| decoder_conv2d_1 | Conv2D | $(None, 8, 8, 4)$ | 580 | decoder_resize_1 |
| decoder_resize_2 | Lambda (resize) | $(None, 16, 16, 4)$ | 0 | decoder_conv2d_1 |
| decoder_conv2d_2 | Conv2D | $(None, 16, 16, 8)$ | 296 | decoder_resize_2 |
| decoder_resize_3 | Lambda (resize) | $(None, 32, 32, 8)$ | 0 | decoder_conv2d_2 |
| decoder_conv2d_3 | Conv2D | $(None, 32, 32, 16)$ | 1168 | decoder_resize_3 |
| decoder_conv2d_4 | Conv2D | $(None, 32, 32, 3)$ | 435 | decoder_conv2d_3 |
| decoder_activation | Activation | $(None, 32, 32, 3)$ | 0 | decoder_conv2d_4 |
| Total | | | 38 348 863 | |

Table 5.2: Implemented CapsNet architecture: Each model in different conditions can differ in number of parameters and and each layer shapes. For example initial convolution can be set to different amount of filters which is determined by experiment. Other example can be the amount of capsules in prediction layer (and its output shape) since that is tightly bonded to the amount of identity bins. Listed layer types are understood to belong to `keras.layers` namespace. In case of `Lambda` layers, the function in use is listed in parenthesis.

2. Establishing model
3. Training a model
4. Validation of trained accuracy
5. Publishing results

### 5.6.1 Data set preparation

In previous chapters suitable data sets were already mentioned and elaborated. Since size of our solution is greatly dependent on the amount of identities, we need to select a data set with fair ratio between the size of labels vector and amount of samples per each identity. For this particular demonstration we chose the *Labeled Faces in the Wild* 4.4.2 due to fair distribution when limited to 25 samples per label or greater, and its simplicity to collect via *scikit-learn*[5] library. We also tried and demonstrated in our Jupyter notebooks the *PINS* 4.4.8 data set, but the diversity in data proved itself to be of no use. However the collection code remains in the thesis sources and in the notebooks, so it is available to be experimented with.

For next parts of this text, we will continue to use the *LFW* data set as the sole example for input data. In order to collect this data set we don't need to be inventing the wheel again and we can leverage the capabilities granted via `sklearn` library.

[5] https://scikit-learn.org

```
people = fetch_lfw_people(
    color=True,
    min_faces_per_person=25,
)
```

Listing 5.15: Collect Labeled Faces in the Wild data set

By default the images are $300 \times 300$ pixels big, though since we aim to process images as small as $32 \times 32$ pixels we need to resize the images. We can either try to reason with a `resize=` parameter, though this size fraction can prove itself unreliable so we will use help of the *Pillow*[6] library:

```
x = people.images

def downsample(image):
    image = Image.fromarray(image.astype('uint8'), 'RGB')
    image = image.resize(resize_to, Image.ANTIALIAS)

    return np.array(image)

x = np.array([downsample(i) for i in x]) / 255
```

Listing 5.16: Pre-processing of the data set

Now we have solved the image sizes, we can prepare the data set into two distinct bundles, one for training and one for validation. This means the data we are training the model against are not the same we use later for validation, therefore the result of validation is pure and not affected by over-fitting. Here we can use the help of *scikit-learn* library once again, since it provides a `train_test_split` function which allow us to separate these two sets randomly, with a respective desired ratio. And last but not least, we need to convert our labels to a hot one encoding. This time we can use a *Keras* native function `to_categorical`.

The data set is now prepared, let's take a look at the distribution of images magnitude per label that we can expect as well as other metrics we may consider before training. As we can see on following Fig 5.4, the distribution is not ideal, and our model will have a tendency to over-fit on certain overrepresented labels and under-fit on others. This is unfortunate and we can try to eliminate such behaviour by a *dropout* layer and with additional augmentation.

### 5.6.2 Train

Training of a *Keras* model can be handled in multiple different ways. Since our model requires multiple input and provides multiple outputs when trained, the `fit_generator` approach was used. It provides greater control over data passed to the model than a simple `fit`, while it remains a pretty simple to implement than a full custom training. Naturally, the first and foremost in *Keras*, we need to compile a model, then we can use the already mentioned `fit_generator` method with a data generator to process the training.

[6] https://pillow.readthedocs.io

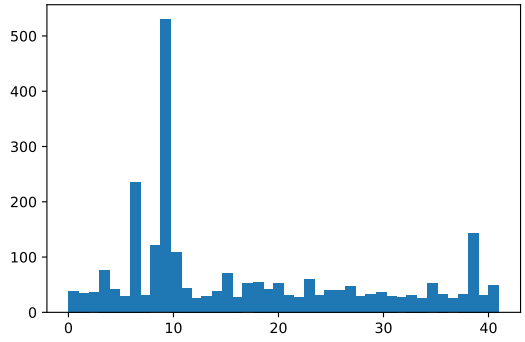


Figure 5.4: LFW subset distribution

| | |
|---|---|
| Number of subjects | 42 |
| Total amount images | 2588 |
| Amount of images for **training** | 2070 |
| Amount of images for **testing** | 518 |
| Minimum samples per subject | 25 |
| Maximum samples per subject | 521 |
| Resolution | $32 \times 32$ px |
| Channels | 3 color channels, RGB |

Table 5.3: Metrics of the used subset of *LFW* data set

```
model.compile(
    optimizer=optimizers.Adam(lr=lr),
    loss=[margin_loss, 'mse'],
    loss_weights=[1., decoder_loss_weight],
    metrics={'capsnet': 'accuracy'}
)

history = model.fit_generator(
    generator=dataset_gen(x_train, y_train, batch_size=batch_size),
    steps_per_epoch=len(x_train) / batch_size,
    epochs=epochs,
    validation_data=[[x_test, y_test], [y_test, x_test]],
    verbose=1,
    callbacks=cb
)
```

Listing 5.17: Training a *Keras* mode using `fit_generator`

As you may have noticed, you can see that we've used `keras.optimizers.Adam` optimizer and allow to configure the learning rate (`lr`), weight of the decoder loss function (`decorator_loss_weight`) as well as a batch size (`batch_size`) and final amount of epochs. The training uses the margin loss function according to Eq. 3.10 and a data generator `dataset_gen`. This generator provides additional data augmentation via standard

*Keras* image processors and yields two sets of data, one for each input with their respective validation truths:

```
def dataset_gen(x, y, batch_size):
    datagen = ImageDataGenerator(
        width_shift_range=0.1,
        height_shift_range=0.1,
        rotation_range=20,
        # horizontal_flip=True
    )
    generator = datagen.flow(x, y, batch_size=batch_size)
    while 1:
        x_batch, y_batch = generator.next()
        yield ([x_batch, y_batch], [y_batch, x_batch])
```

Listing 5.18: Data generator example

### 5.6.3 Test model and predict labels

Testing and providing predictions is a simple process, since everything comes already prepared with *Keras* models. For testing purposes a model is required to be compiled, otherwise it can't provide loss calculations and therefore it can be tested. On the other hand, when we desire simple predictions the model doesn't need to be compiled, only presence of weights is required.

```
def test(model, x_test, y_test, batch_size=10):
    model.compile(
        optimizer='adam',
        loss=margin_loss,
        metrics={'capsnet': 'accuracy'}
    )
    return model.evaluate(x_test, y_test, batch_size=batch_size)
```

Listing 5.19: Test run of a *Keras* model example

```
def predict(model, x, batch_size=10):
    return model.predict(x, batch_size=batch_size)
```

Listing 5.20: Prediction example for a given model

### 5.6.4 Save and load a model

This is a last part of a model life cycle. Previous text showed how each phase contributes to creation of a model which finally allows a researched to observe and measure their experiments. To have a model last, we need a meaning to store it's trained properties as well as layout. Since our models allow great customization and invasive changes in the core architecture, we are obliged to persist not only the weight of every trainable parameter,

but also a configuration of each layer, amounts of neurons in each, probability rates preset before even any training begun. *Keras* library provides a mean how to achieve that. Each model can be transformed into a H5, JSON or YAML format, while only the H5 format allows us to save weights along with it. However, since the *CapsNet* model is not a single model and we define two different models, one for training and one for inference, we have to keep our weights separate. Hence we save them as a H5 file separately and then proceed to store the model architectural configuration settings as a separate YAML file for each model. And just to keep everything together and bundled, we pack these 3 files into a gun-zipped tarball. This is how we can do it:

```
with open('train.yml', 'w') as f:
    f.write(training_model.to_yaml())

with open('test.yml', 'w') as f:
    f.write(testing_model.to_yaml())

training_model.save_weights('weights.h5')
```

Listing 5.21: Saving a model in *Keras*

The process shown here is significantly simplified, since our tarball scenario requires more complex handling, but the principles stays the same. A similar process, yet reversed procedure can be used later to load the data back into a model instance. The implementation provided with this thesis allows passing a special parameter to `CapsNet` constructor which would skip the network initialization phase and allow to create a model from a class method. This `load` method uses native function `model_from_yaml` to load up the network configurations and later `load_weights` methods to upload trained values. Since we store the weights from the more complex model only, the simplified testing model, is required to load the weights by layer name. Here is how it might look like in a simplified scenario. If a reader is interested in more details of this process, feel free to consult our sources.

```
custom_objects = {
    # Custom layers unknown to Keras
    'PredictionCapsule': PredictionCapsule,
    'FeatureCapsule': FeatureCapsule,
    'Mask': Mask,
    # TensorFlow and Keras back-end functions used in lambda layers
    'tf': tf,
    'k': k
}

with open('train.yml', 'r') as f:
    training_model.model_from_yaml(f.read(), custom_objects=custom_objects)
training_model.load_weights('weights.h5')

with open('test.yml', 'r') as f:
    testing_model.model_from_yaml(f.read(), custom_objects=custom_objects)
testing_model.load_weights('weights.h5', by_name=True)
```

Listing 5.22: Loading a model in *Keras*

## 5.7 Running an experiment

Based on previous sections we have prepared our network to be successfully trained and evaluated. This can be either initiated locally, or via a hosted *Jupyter* hub. In this case, we used *Google Colab* as a *Jupyter* hosted runtime. Notebooks with execution details are provided as part of the `capsnet` package sources. Our library provides all necessary APIs to maximize user friendliness:

```
model = CapsNet(
    x_train.shape[1:],
    len(np.unique(y_train, axis=0)),
    routing_iters=3,
    kernel_initializer=initializers.random_normal(stddev=0.01, seed=0),
    dropout_rate=.7,
    init_conv_filters=256,
    init_conv_kernel=9,
    feature_caps_kernel=5,
    feature_caps_dim=16,
    feature_caps_channels=16,
    prediction_caps_dim=32,
    image_size=(32, 32)
)

model.train(data, batch_size=10)

model.test(x_test, y_test)

model.predict(x_test)
```

Listing 5.23: Running an experiment using `capsnet`

### 5.7.1 Model evaluation

The model has been trained in different conditions and settings and multiple times. The mean values of accuracy and loss is listed in the following Table 5.4 .In this case we experimented with routing iterations and comparing 1 routing to 3 consecutive ones. As proved by Hinton [15] the amount of iterations has impact on network accuracy and can make the training converge faster. Each model was trained 3 times and the accuracy and loss values are calculated as ann average.

The showed image evaluations are just samples of the whole testing scenario. To better understand the whole picture we should consider a confusion matrix, which shows, how was each image belonging to a label actually classified. Due to readability we are listing only the confusion matrix for the model of 12 individuals in this paper. The confusion matrix for the 42 identities is available as part of the notebooks provided along with this thesis.

| Identities | Data set (images) | Routing iterations | Accuracy | | | Loss |
|---|---|---|---|---|---|---|
| | | | Train | Validation | Test | |
| 42 | LFW (2588) | 1 | 46.2% | 42.5% | 42.5% | 0.5002 |
| 42 | LFW (2588) | 3 | 56.4% | 53.7% | 42.5% | 0.3915 |
| 12 | LFW (1560) | 1 | 52.6% | 63.2% | 61.5% | 0.2952 |
| 12 | LFW (1560) | 3 | 69.3% | 75.0% | 73.7% | 0.2013 |

Table 5.4: Model evaluations when trained with focus on different amount of identities and routing iterations impact.

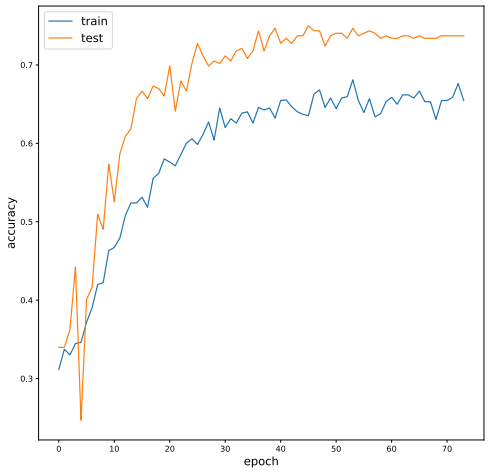


Figure 5.5: History of training of a 11 identities model showing the testing and training accuracy development.

The confusion matrix states some interesting findings. First of all *George W. Bush* and *Colin Powell* are over-fitting labels. *John Ashcroft* is not recognized on any of his photos, *Hugo Chaves* successfully blends in the crowd as well as *Gerhard Schroeder*. The reasons varies, each may have different explanation:

1. *George W. Bush*: It is the label with the biggest number of images in the data set. Therefore it is likely that this identity would over-fit.

2. *Colin Powell*: This identity features significant amount wrinkles. Therefore it may be easy to match on this character, since it would provide high activation by default when wrinkles are present.

3. *John Ashcroft*: This individual's face doesn't hold many characteristic markers at this resolution. Unfortunately his face at $32 \times 32$ pixels seems generic and blends with others, for instance with *Colin Powell*.

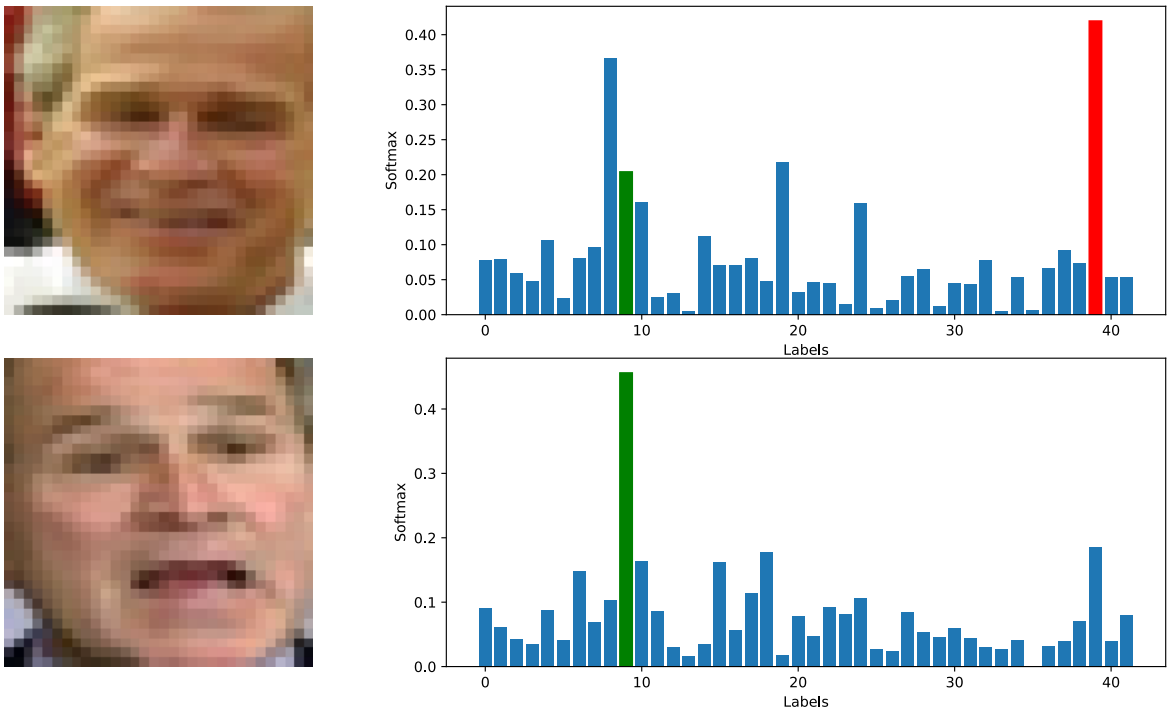


Figure 5.6: *Top*: Tony Blair predicted, while correct is George W. Bush. As you may notice the correct capsule holds the 4th largest activation, however the result is strongly in favor of the faulty label. The faulure may seem to occur despite the fact that George W. Bush has the most images in data set and may be due to the bright spots and overall pale appearance which may remind Tony Blair. *Bottom*: This time George W. Bush is predicted correctly. Activations taken from 42 identities model. A bin labelled in green belongs to the identity capsule, a red bin shows the predicted activation.

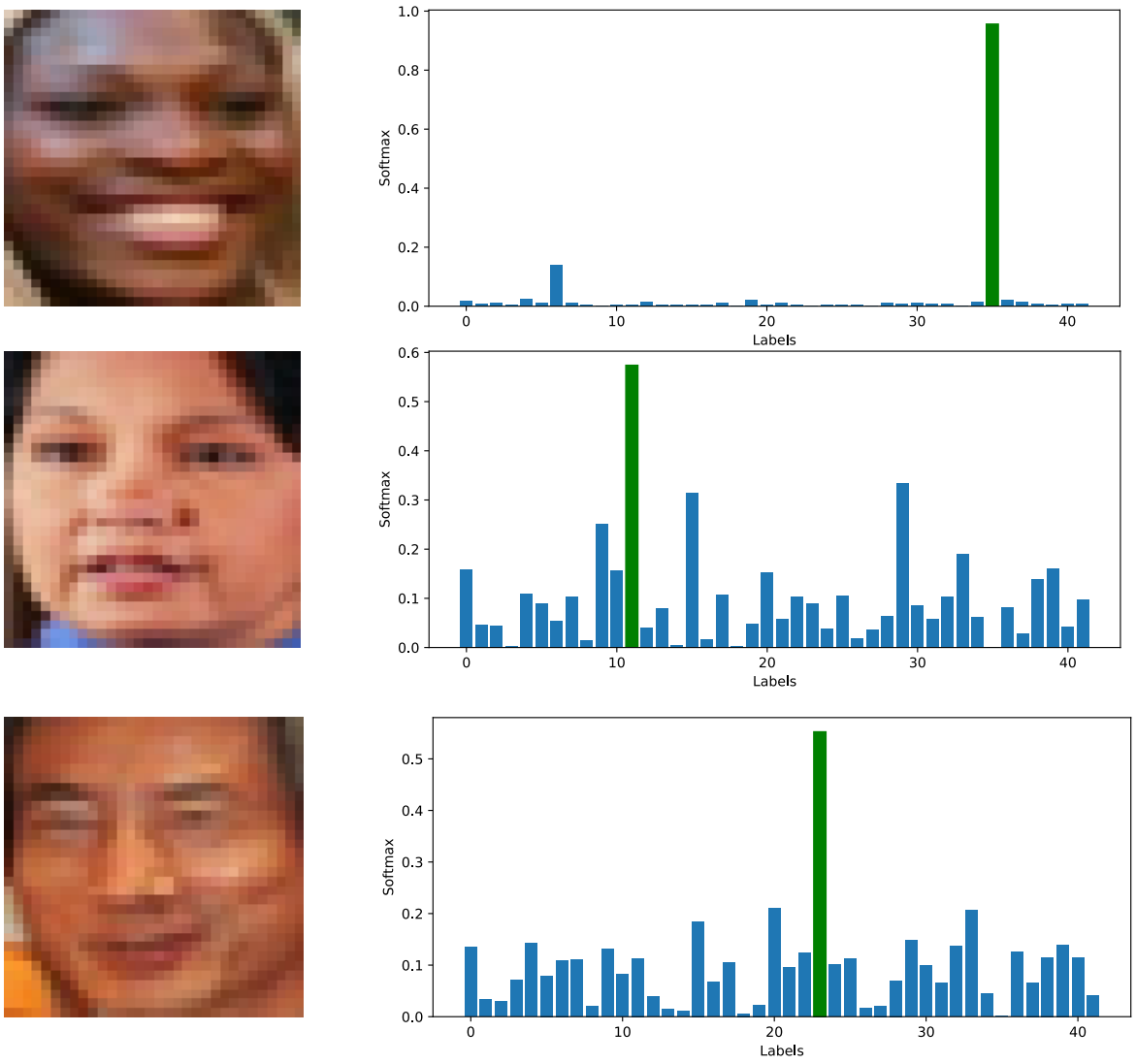


Figure 5.7: Both images show the network can handle people of various ethnicities fine. *Top*: Serena Williams. *Middle*: Gloria Macapagal Arroyo. *Bottom*: Junichiro Koizumi. What seems to matter more is the actual settings and lightning of the input image, in which many features may stood out better or fade away and blend with the rest of the image. Activations taken from 42 identities model.

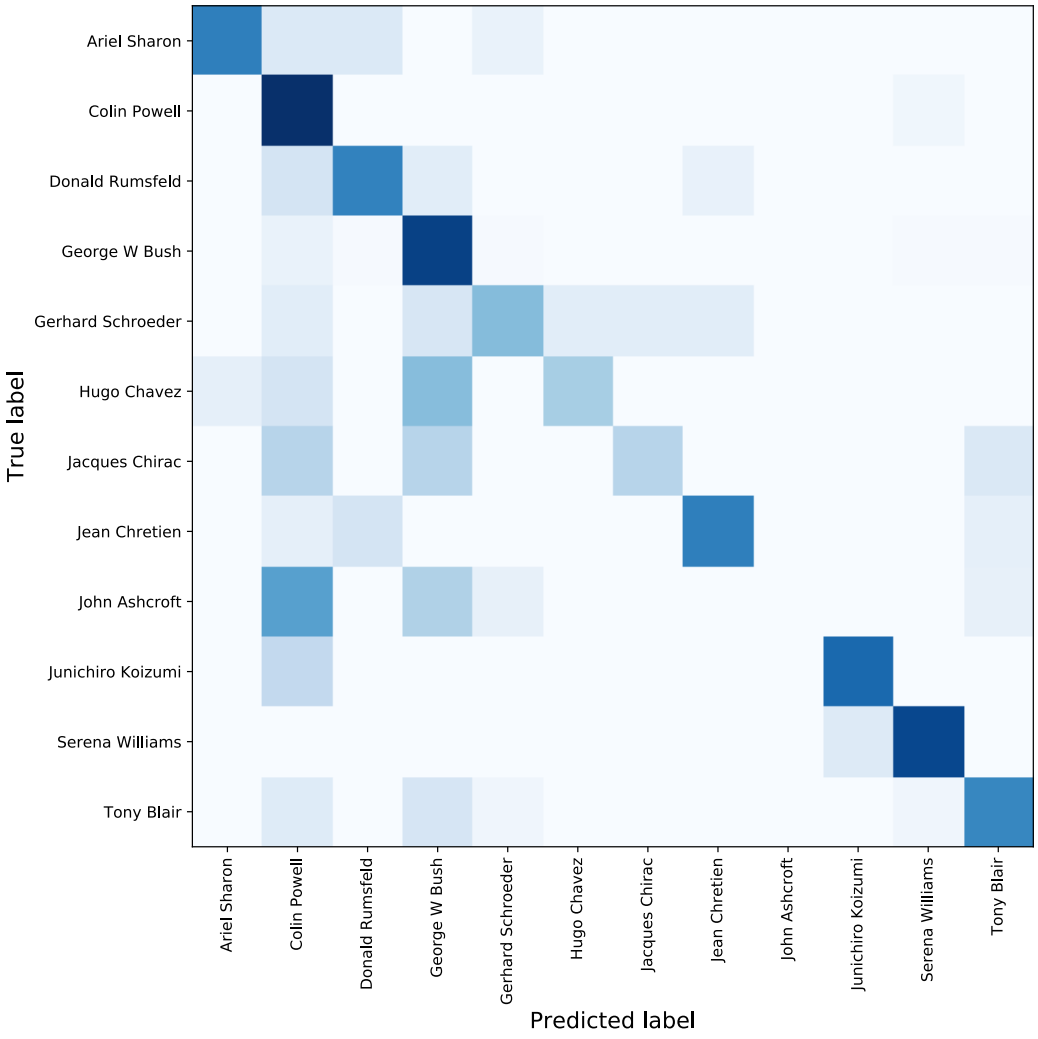


Figure 5.8: Confusion matrix for the smaller model which identifies 11 individuals.

# Chapter 6

# Conclusion

Thesis aimed to cover, explain and experiment with new approaches to facial recognition. The research in Chapter 3 listed and covered mathematical and engineering principles of construction of convolutional neural networks. It stumbled upon some drawbacks of such design and proposed available solutions. It discusses convolutional neural networks as well as the newest capsule network approach. This throughout review of principles behind successful identification is later transformed into a state-of-the-art market scan, providing a quick insight into publicly available solutions as well as available frameworks. Fundamental benefits and advantages of each framework and solution is listed in the Chapter 4.

The very same chapter also provides an overview of available training data sets are listed, with focus on in-the-wild image data. This review includes a short description of each, accompanied by basic metric. Large scale face databases are listed as well.

This knowledge is later leveraged to create own implementation of a capsule network. The Chapter 5 covers the actual design and implementation of this solution. A part from this implementation we also demonstrated how such model can be trained and shown the results.

## 6.1 Experiment discussion

Overall our implementation proved itself to be successful in recognition of 75% individuals when trained with 11 indentities. This accuracy decreases dramatically with either more identities added or less training data per identity. These effects correlate but the actual cause may be n one factor only. That would require testing against a data set greater in magnitude, while balanced in amount of images per identity.

In the case of a model trained to recognize 11 different identities, we had available 50 or more images per each individual. These images covered different settings, angles and poses. This might have proved to be a great benefit to our model. On the other hand, the fewer amount of individuals to recognise there is much less parameters to be learned in the network and therefore the error margin is smaller. And since our data are small images of $32 \times 32$ pixels, the capsule network struggles to successfully match a greater amount of identities.

## 6.2 Improvements and suggestions

A capsule network provides great power at smaller scales, however, when it is facing big problems, it demands great computational powers and resources. Hence quick prototyping and full scale training result in big differences in network configurations and therefore the observed behaviours. To achieve better results with this type of network, over more identities, it would be necessary to provide sufficient amount of input data, great GPU resources. Enhancing detail on the input data form $32 \times 32$ pixels to more, comes with a great cost as well, since nearly every layer's parameter count is dependent on amount of input pixels. That means the resource demand is raising again. This problem may seem easy to bypass – forced retraining of the capsule network on a new label. This is topic is even newer than capsule networks itself and so far was not solved sufficiently.

# Bibliography


[1] BERG, T. L.; BERG, A. C.; EDWARDS, J.; et al.: Who's in the picture. Technical report. Neural Information Processing Systems (NIPS). 2004.

[2] BHANU, B.; Ju HAN, J.: *Human Recognition at a Distance in Video*. Springer London. 2010. doi:10.1007/978-0-85729-124-0.
Retrieved from: https://doi.org/10.1007/978-0-85729-124-0

[3] CAO, Q.; SHEN, L.; XIE, W.; et al.: VGGFace2: A dataset for recognising faces across pose and age. *CoRR*. vol. abs/1710.08092. 2017. 1710.08092.
Retrieved from: http://arxiv.org/abs/1710.08092

[4] CHENG, Z.; ZHU, X.; GONG, S.: Surveillance Face Recognition Challenge. *CoRR*. vol. abs/1804.09691. 2018. 1804.09691.
Retrieved from: http://arxiv.org/abs/1804.09691

[5] DRAHANSKÝ, M.; ORSÁG, F.; DOLEŽEL, M.: *Biometrie*. Computer Press, s.r.o. first edition. 2011. ISBN 978-80-254-8979-6. 294 pp.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=9468

[6] Freepik: Hand drawn woman face.
https://www.freepik.com/free-photos-vectors/hand. last retrieved 2019-01-10.

[7] GATES, K.: *Our Biometric Future: Facial Recognition Technology and the Culture of Surveillance*. NYU Press. 2011. ISBN 0814732100, 9780814732106.

[8] GEORGHIADES, A.; BELHUMEUR, P.; KRIEGMAN, D.: From few to many: illumination cone models for face recognition under variable lighting and pose. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. vol. 23, no. 6. jun 2001: pp. 643–660. doi:10.1109/34.927464.
Retrieved from: https://doi.org/10.1109/34.927464

[9] GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A.: *Deep Learning (Adaptive Computation and Machine Learning series)*. The MIT Press. 2016. ISBN 0262035618.
Retrieved from: http://www.deeplearningbook.org

[10] GOODFELLOW, I.; WARDE-FARLEY, D.; MIRZA, M.; et al.: Maxout Networks. In *Proceedings of the 30th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, vol. 28, edited by S. Dasgupta; D. McAllester. Atlanta, Georgia, USA: PMLR. 17–19 Jun 2013. pp. 1319–1327.
Retrieved from: http://proceedings.mlr.press/v28/goodfellow13.html

[11] GREGOR, K.; LECUN, Y.: Emergence of Complex-Like Cells in a Temporal Product Network with Local Receptive Fields. *CoRR*. vol. abs/1006.0448. 2010. 1006.0448. Retrieved from: http://arxiv.org/abs/1006.0448

[12] GRGIC, M.; DELAC, K.; GRGIC, S.: SCface – surveillance cameras face database. *Multimedia Tools and Applications*. vol. 51, no. 3. 2009: pp. 863–979. doi:0.1007/s11042-009-0417-2.

[13] Guo, X.: CapsNet-Keras. https://github.com/XifengGuo/CapsNet-Keras. last retrieved 2019-03-05.

[14] HAN, S.; MAO, H.; DALLY, W. J.: Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR*. vol. abs/1510.00149. 2015. 1510.00149.
Retrieved from: http://arxiv.org/abs/1510.00149

[15] HINTON, G.; SABOUR, S.; FROSST, N.: Dynamic Routing Between Capsules. *CoRR*. vol. abs/1710.09829. 2017. 1710.09829.
Retrieved from: http://arxiv.org/abs/1710.09829

[16] HINTON, G.; SABOUR, S.; FROSST, N.: Matrix capsules with EM routing. In *International Conference on Learning Representations*. 2018.
Retrieved from: https://openreview.net/pdf?id=HJWLfGWRb

[17] HUANG, G. B.; RAMESH, M.; BERG, T.; et al.: Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. Technical Report 07-49. University of Massachusetts, Amherst. October 2007.

[18] IANDOLA, F. N.; MOSKEWICZ, M. W.; ASHRAF, K.; et al.: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *CoRR*. vol. abs/1602.07360. 2016. 1602.07360.
Retrieved from: http://arxiv.org/abs/1602.07360

[19] JAIN, A.; HONG, L.; PANKANTI, S.: Biometric identification. *Communications of the ACM*. vol. 43, no. 2. 2000: pp. 90–98.

[20] JAIN, A. K.; FLYNN, P.; ROSS, A. A. (editors): *Handbook of Biometrics*. Springer US. 2008. doi:10.1007/978-0-387-71041-9.
Retrieved from: https://doi.org/10.1007/978-0-387-71041-9

[21] JAIN, V.; LEARNED-MILLER, E.: FDDB: A Benchmark for Face Detection in Unconstrained Settings. Technical Report UM-CS-2010-009. University of Massachusetts, Amherst. 2010.

[22] LECUN, Y.: Learning processes in an asymmetric threshold network. In *Disordered systems and biological organization, Les Houches, France*. Springer-Verlag. 1986. pp. 233–240.

[23] LI, S. Z.; JAIN, A. K. (editors): *Handbook of Face Recognition*. Springer London. 2011. doi:10.1007/978-0-85729-932-1.
Retrieved from: https://doi.org/10.1007/978-0-85729-932-1

[24] LIU, S.; SILVERMAN, M.: A practical guide to biometric security technology. *IT Professional*. vol. 3, no. 1. Jan 2001: pp. 27–32. ISSN 1520-9202. doi:10.1109/6294.899930.

[25] LIU, Z.; LUO, P.; WANG, X.; et al.: Deep Learning Face Attributes in the Wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, vol. abs/1411.7766. 2014. 1411.7766.
Retrieved from: http://arxiv.org/abs/1411.7766

[26] MAYHEW, S.: History of Biometrics [online]. URL „http://www.biometricupdate.com/201501/history-of-biometrics“. 2015. [cit. .1.2019].

[27] MUKHOMETZIANOV, R.; CARRILLO, J.: CapsNet comparative performance evaluation for image classification. *CoRR*. vol. abs/1805.11195. 2018. 1805.11195.
Retrieved from: http://arxiv.org/abs/1805.11195

[28] NEVEU, T.: Capsnet – Traffic sign classifier. https://github.com/thibo73800/capsnet-traffic-sign-classifier. last retrieved 2019-03-05.

[29] NIETO, M.; JOHNSTON-DODDS, K.; SIMMONS, C. W.: *Public and private applications of video surveillance and biometric technologies*. California State Library, California Research Bureau California. 2002.

[30] NIXON, M. S.; AGUADO, A. S.: *Feature extraction & image processing for computer vision*. Academic Press. 2012.

[31] PARKHI, O. M.; VEDALDI, A.; ZISSERMAN, A.: Deep Face Recognition. 01 2015. pp. 41.1–41.12. doi:10.5244/C.29.41.

[32] PRABHU, U.; SESHADRI, K.: Facial recognition using active shape models, local patches and support vector machines. *contrib. andrew. cmu. edu*. 2009: pp. 1–8.

[33] RAMESH, K.: CapsNet4Faces. https://github.com/krishnr/CapsNet4Faces. last retrieved 2019-03-05.

[34] SCHROFF, F.; KALENICHENKO, D.; PHILBIN, J.: FaceNet: A Unified Embedding for Face Recognition and Clustering. *CoRR*. vol. abs/1503.03832. 2015. 1503.03832.
Retrieved from: http://arxiv.org/abs/1503.03832

[35] SZEGEDY, C.; IOFFE, S.; VANHOUCKE, V.: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *CoRR*. vol. abs/1602.07261. 2016. 1602.07261.
Retrieved from: http://arxiv.org/abs/1602.07261

[36] Toy, B.: Aligned Face Dataset For Face Recognition – Aligned Face Dataset from Pinterest. https://www.kaggle.com/frules11/pins-face-recognition. last retrieved 2019-01-26.

[37] VACCA, J. R.: *Biometric technologies and verification systems*. Butterworth-Heinemann. 2007.

[38] VIOLA, P.; JONES, M. J.: Robust real-time face detection. *International journal of computer vision*. vol. 57, no. 2. 2004: pp. 137–154.

[39] WECHSLER, H.; PHILLIPS, P. J.; BRUCE, V.; et al. (editors): *Face Recognition*. Springer Berlin Heidelberg. 1998. doi:10.1007/978-3-642-72201-1.
Retrieved from: https://doi.org/10.1007/978-3-642-72201-1

[40] YANDONG, G.; LEI, Z.; YUXIAO, H.; et al.: MS-Celeb-1M: A Dataset and Benchmark for Large Scale Face Recognition. In *European Conference on Computer Vision*. Springer. 2016.

[41] ZHANG, K.; ZHANG, Z.; LI, Z.; et al.: Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks. *IEEE Signal Processing Letters*. vol. 23, no. 10. Oct 2016: pp. 1499–1503. ISSN 1070-9908. doi:10.1109/LSP.2016.2603342.

[42] ZHOU, Y. T.; CHELLAPPA, R.: Computation of optical flow using a neural network. *IEEE International Conference on Neural Networks*. 1988: pp. 71–78. doi:10.1109/icnn.1988.23914.
Retrieved from: https://ieeexplore.ieee.org/iel5/763/907/00023914.pdf