Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Dipartimento di Fisica e Astronomia
Corso di Laurea Magistrale in Fisica

Capsule Networks: a new approach for brain imaging

Relatore:
Prof. Nico Lanconelli
Correlatore:
Prof. Renato Campanini
Dott. Matteo Roffilli

Presentata da:
Caterina Camborata

Anno Accademico 2017/2018

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

G. Hinton

Sommario

Nel campo delle reti neurali per il riconoscimento immagini, una delle più recenti e promettenti innovazioni è l'utilizzo delle Capsule Networks (CapsNet).
Lo scopo di questo lavoro di tesi è studiare l'approccio CapsNet per l'analisi di immagini, in particolare per quelle neuroanatomiche. Le odierne tecniche di microscopia ottica, infatti, hanno posto sfide significative in termini di analisi dati, per l'elevata quantità di immagini disponibili e per la loro risoluzione sempre più fine. Con l'obiettivo di ottenere informazioni strutturali sulla corteccia cerebrale, nuove proposte di segmentazione possono rivelarsi molto utili.

Fino a questo momento, gli approcci più utilizzati in questo campo sono basati sulla Convolutional Neural Network (CNN), architettura che raggiunge le performance migliori rappresentando lo stato dell'arte dei risultati di Deep Learning.
Ci proponiamo, con questo studio, di aprire la strada ad un nuovo approccio che possa superare i limiti delle CNNs come, ad esempio, il numero di parametri utilizzati e l'accuratezza del risultato.
L'applicazione in neuroscienze delle CapsNets, basate sull'idea di emulare il funzionamento della visione e dell'elaborazione immagini nel cervello umano, concretizza un paradigma di ricerca stimolante volto a superare i limiti della conoscenza della natura e i limiti della natura stessa.

Abstract

In the field of Deep Neural Networks for image recognition, the development of Capsule Networks is one of the most recent and promising innovations.
The purpose of this work is to study the feasibility of a CapsNet approach to images analysis, especially for neuroanatomical ones. The state-of-the-art microscopy techniques pose significant challenges in terms of data analysis, because of the huge amount of available data and for the more and more subtle resolution. For the purpose of acquiring information about the structure of brain cortex, new proposals for image segmentation may be very helpful.

Up until last year, common approaches in this framework were based on Convolutional Neural Networks (CNNs), an architecture that achieves the best performances in Deep Learning.
The aim of this work is to pave the way for a new approach to let it overcome the limits of the CNNs, for example in number of parameters and in accuracy.
The application to neurosciences of CapsNets, based on the idea of emulating the human brain's functioning, in matters of vision and images elaboration, realises a stimulating research paradigm, aimed at overcoming the limits of our knowledge of nature and the limits of nature itself.

CONTENTS

INTRODUCTION

Artificial intelligence (AI) is nowadays one of the most up to date and discussed fields. AI is, by definition, the capability of computers to perform tasks that normally would require human intelligence. There is currently a huge number of active research topics regarding AI, all with the common aim of automating some type of function. Deep learning and artificial intelligence are terms that can be heard almost every day regarding most various tasks. Some examples of automatic functions are understanding and recognizing speech and images, competing in strategic game systems, self-driving cars, interpreting complex data in order to make predictions and making diagnoses in medicine. These techniques are constantly applied to new problems and their full potential has not been totally investigated yet.

It looks like, through deep learning algorithms, automatic systems will be able to revolutionize the world in few years; however, until now the expansion of this discipline is mostly due to the success of Convolutional Neural Networks. CNNs are a type of deep neural networks, which have achieved outstanding results in speech and object recognition and natural language processing tasks. Deep Learning techniques, thanks to the increase in computational power and to the availability of massive new data sets, can impact medicine and healthcare [1]. Furthermore, several studies have successfully applied CNNs for analyzing medical images and per-

forming tasks such as image classification, object detection and segmentation. Recent technical progress in neuroscience opened the possibility to apply deep learning methods also to brain imaging. The final goal of nowadays research is to provide a map of the whole human brain at a cellular resolution in order to open new exciting possibilities for both research and diagnostic purposes. There have been attempts to apply networks designed for object categorization to segmentation, particularly by replicating the deepest layer features in blocks to match image dimensions. Some of these recent approaches that tried to directly adopt deep architectures designed for category prediction to pixel-wise labeling, although very encouraging, appear coarse.

Geoffrey Hinton, considered by some as the "Godfather of Deep Learning", is without any doubt a leading figure in the deep learning community. In a famous talk that he gave at MIT ([2]) he explained its *Capsule Theory* starting from his idea of computer vision and from the state-of-the-art deep learning performance. So he analyzed the problems of CNNs and proposed an alternative, exotic, deep learning architecture that, in his opinion, could take the place of CNNs for most of imaging tasks. Recently, thanks to the innovation in technology, he was able to test his idea and now the challenge is to lead this new architecture to be really better than the standard CNNs.

In the present work we want to go through the revolutionary idea of Capsule Networks, pointing out its pros and cons.

CapsNet is based on an idea of vision that would like to emulate human brain vision with both theoretical and practical advantages. This work's aim is also to study this new architecture as a novel approach in particular for brain imaging.

In the first chapter we make a review of the Deep Learning techniques, starting from the definition of a deep neural network to the description of the CNN architecture. Then we introduce the Capsule Networks architecture in its innovative aspects.

In the second chapter we study the state-of-the-art techniques used for brain imaging in mice and human data. This study was very useful to get into the problem of brain imaging from data acquisition to computational

problems. For this kind of data set, in fact, a common approach is based on object segmentation. So it was necessary to introduce the state-of-the art methods for this task, based on CNNs, and the current attempt to use CapsNets for segmentation tasks.

In the third chapter we applied the CapsNets approach to classify a known data set in order to verify the potentiality of this architecture and to compare the results with the ones of a standard CNN. Both of the experiments are made using Keras, a TensorFlow API widely used in machine learning. More details on this tool are explained in Appendix A.

CHAPTER 1

DEEP LEARNING FOR IMAGE RECOGNITION: STATE OF THE ART

"The machine does not isolate man from the great problems of nature but plunges him more deeply into them."
A. de Saint-Exupéry

Humans, since the ancient Greeks, have always dreamed about thinking machines. This dream became a real achievement to work at when, in 1950, Turing conceived and invented the very first programmable computer.

When he talked about his most famous paper where he introduced the so called "Turing test" he said [3]:

> *There would be plenty to do in trying to keep one's intelligence up to the standard set by the machines, for it seems probable that once the machine thinking method had started, it would not take long to outstrip our feeble powers.*

Nowadays computers capabilities are comparable to those of biological organisms insomuch we can talk about "artificial intelligence" to distinguish from natural intelligence derived from human brain.

From the outset, one of the goals of artificial intelligence has been to equip machines with the capability of dealing with sensory input [4]. First of all AI was used to solve problems that are difficult to human beings but relatively straightforward for computers, but then the true challenge began to solve tasks that are easy for people to perform but hard to describe formally.

These tasks are intuitive because people can use intrinsic information learned automatically in their whole life. One of these intuitive problems concerns computer vision: understand and recognize images, for example to make diagnoses in medicine or to support basic scientific research. Computer vision is the construction of explicit, meaningful descriptions of physical objects from images [4] and this explicitness is the challenge. The question is: what computation must be performed to extract information about a scene from an image, using only very basic assumptions?

The answer at this question results to be Deep Learning. This solution provides that computers have to learn from experience and understand the world in terms of hierarchy of concepts, each of one is related with simpler concepts [5].

The approach used in machine learning is a data-driven approach that consists in provides the computer with many examples of each class of objects and then develop learning algorithms that look at these examples and learn about the representation of each class.

Computer vision tasks usually are solved starting with the assignment of a label to an input image from a set of categories. Image classification presents a lot of challenges, for example viewpoint variation, scale variation, deformation and occlusion, illumination conditions, background clutter and intra-class variation [6]. The ability to acquire knowledge is known as *machine learning* and it works by extracting patterns from raw data.

A machine learning task could be described as how the system should process an example, that is a set of features. If the example is an image we can represent it as a vector $\vec{x} \in \mathbb{R}^n$ where each component x_i is a pixel value. Mathematically the classification task that a learning algorithm is asked to achieve is to produce a *score function* $f : \mathbb{R}^n \to \{1, \ldots, k\}$ where

k is the number of possible categories in which we want the computer to classify the data, and n is the dimension of the input. So when computer applies this function to an input example it can assign that example to a category y if the numeric code $y = f(\vec{x})$.

The performance of this kind of algorithms depends on the representation of the data they are given, in other words on the features chosen to identify and classify the data. Many artificial intelligence tasks can be solved by designing the right set of features to extract for that task, then providing these features to a simple machine learning algorithm. Sometimes we don't know what feature should be extracted or, as often happens in computer vision, we don't know how to described the features in terms of pixel values. In these cases we use an approach called *representation learning* that uses machine learning to discover the right representation, the features, and then it maps them to output. Manually designing features requires a huge human effort that is really not necessary even when it's easily possible. An example of a representation algorithm is the autoencoder, a combination of an encoder that extracts information from data and a decoder that reproduces the input from the representation.

The goal in extracting feature is to separate factors of variation, that is to separate the properties useful to distinguish features, and assign them to an object in order to classify it through them. In many real-world AI applications these factors of variation influence every single piece of data and it could be very difficult to extract useful information. Deep learning solves this problem in representation learning by introducing a hierarchical organization of representations. This approach consists of expressing complex features in terms of other simpler features, using an architecture made up of many layers. For example, to map a set of pixels to an object identity, the very first layer has to recognize edges, then another layer has to recognize corners and contours, and so on layers that identify object parts increasing the complexity of the features with the depth of the net.

There aren't a single unique definition for depth, in general we can say that deep learning is the study of models that involves a greater amount of composition of either learned functions or learned concepts than traditional machine learning does.

Traditional machine learning techniques are based on neural networks model starting from the simple perceptron to the multi layer perceptron (MLP). So deep learning born from two basic ideas: learning the right representation for the data and let the computers do it with multi-step program made up of a certain number of layers.

The number of layers usually defines the depth of the net and so its capability to achieve complex tasks with great performance. Goodfellow gives a perfect definition for deep learning in his book [5]:

> *Deep learning is a particular kind of machine learning that achieves great power and flexibility by representing the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones.*

The basic idea is that we first decide an architecture, a structure for our deep neural network with parameters initialized opportunely, and we define a *loss function* to measure the performance of the model. Then we choose a learning algorithm that is what we use to minimize this function and optimize all the parameters to achieve the best performance in solving the task. It must decide how to use the layers to produce the desired output.

In this chapter we go through the main characteristics of deep learning in details and then we focus on the most used model called Convolutional Neural Network (CNN). Finally we emphasise CNNs flaws introducing the Capsule Network architecture with its generalities and its details.

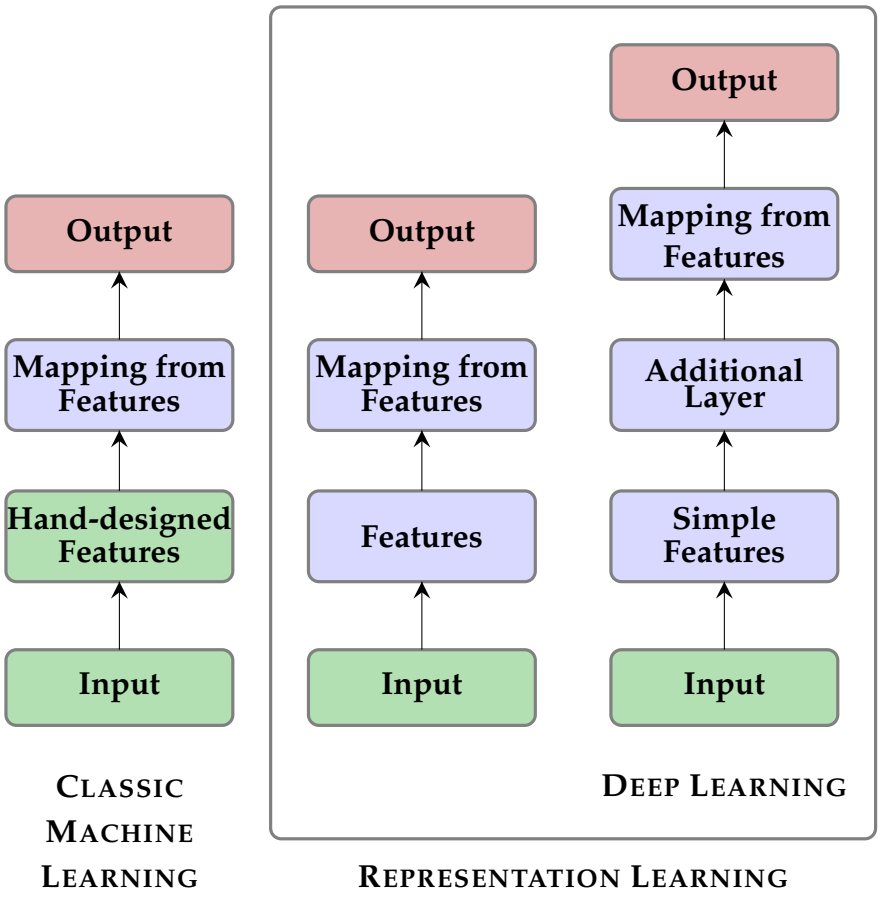


Figure 1.1: This is the framework in which Deep Learning is introduced.

1.1 The architecture

The word *architecture* refers to the overall structure of the network: how many units it should have and how these units should be connected to each other [5]. The architecture of a typical neural network is based on the idea of the perceptron. A perceptron is a single neuron model used to implement binary classifier that, as a biological neuron, has dendrites as input, nucleus as activation function and synapses as output. A neural network can been seen as a series of neurons connected in an acyclic graph [1], this is the reason why deep neural networks are called *feedforward*, because the flow of information is unidirectional from input to output.

As a consequence of the similarity to biological neurons, the architecture of a typical deep neural network is made up of three main structures: an input layer that's the set of data, some hidden layers consisting in different units (neurons), an output layer that's taken to represent the class scores. Because the training data does not show the desired output for each of those layers, they are called *hidden*. Hidden layers' units act in parallel, each representing a vector-to-scalar function called *activation function*.

The depth of the net is connected with the number of layers: a neural network could be considered deep if it has about 10 hidden layers [6]. As we increase the size and number of layers in a Neural Network, the capacity of the network increases. That is, the space of representable functions grows since the neurons can collaborate to express many different functions. This is positive since it can classify more complicated data, but could be also negative because it's easy to do overfitting. So larger networks will always work better than smaller ones, but their higher model capacity must be appropriately addressed with stronger regularization or they might overfit. The ideal network architecture for a task must be found via experimentation guided by monitoring the errors.

Several architectures of deep networks can be studied and the discrim-

[1] It's a graph with no cycles that consists of finitely many vertices and edges, with each edge directed from one vertex to another, such that there is no way to start at any vertex and follow a consistently sequence of edges that eventually loops back to it again.

ination within all the possible variants is in the organization of neurons' layers. The kind of layers differ one each other for the activation function that their units compute and for the connections between the layers.

Units functions

Hidden Units. The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles [5].

The hidden layers are the part of the net where magic happens, here the program computes the learning algorithm that realizes the *machine learning* of the parameters to give the right prediction. Most hidden units can be described as accepting a vector of inputs $\vec{x}$, computing an affine transformation $\vec{z} = \vec{w}^T \vec{x} + \vec{b}$ and then applying an element-wise nonlinear function $g(\vec{z})$. In the transformation $\vec{w}$ represents the weights that have to be inferred during training and $\vec{b}$ represents a bias. A commonly used trick is to combine these two sets of parameters into a single matrix that holds both of them and consecutively extend the vector $\vec{x}$ with one additional dimension that always holds the constant 1. Hidden units are distinguished from each other only by the choice of the form of this activation function. Below we can list three examples for activation function usually used in hidden units (Figure 1.2).

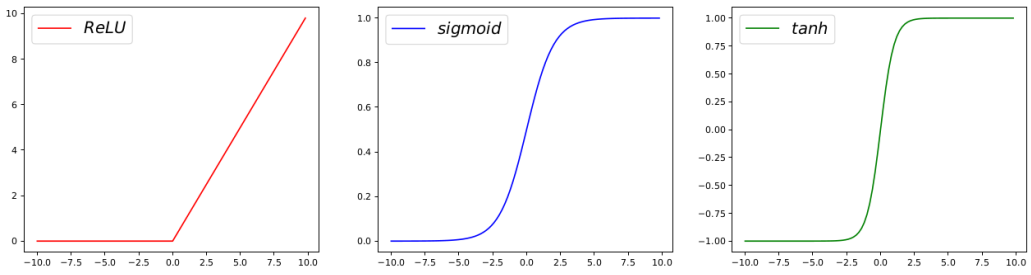


Figure 1.2: Three examples for activation function used in hidden layers.

- **Rectified Linear Unit (ReLU)**
  This activation function accelerate the convergence of the optimiza-

tion, simplify the thresholding providing activation at 0 but can also lead neurons to never activate:

$$g(\vec{z}) = \max\{0, \vec{z}\} \quad \text{with} \quad g(\vec{z}) \in [0; \infty) \tag{1.1}$$

- **Logistic Sigmoid**
  This activation function can lead to a *zig-zagging* dynamics in the optimization process because of the non-linearity:

$$g(\vec{z}) = \frac{1}{1 + e^{-\vec{z}}} \quad \text{with} \quad g(\vec{z}) \in [0; 1] \tag{1.2}$$

- **Hyperbolic Tangent**
  This activation function is a sigmoid neuron scaled with output centered in zero:

$$g(\vec{z}) = \tanh(\vec{z}) = 2\sigma(2\vec{z}) - 1 \quad \text{with} \quad g(\vec{z}) \in [-1; 1] \tag{1.3}$$

It can be difficult to determine when to use which kind, predicting in advance which of the above functions will work best is usually impossible. The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance.

Output Units. Any kind of neural network unit that may be used as an output can also be used as a hidden unit. Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well. The role of the output layer is to provide some additional transformation from the features to complete the task that the network must perform.

- **Linear Unit**
  Given features $\vec{x}$, a layer of linear output units produces a vector:

$$\hat{y} = \vec{w}^T \vec{x} + \vec{b} \tag{1.4}$$

Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms and may be used with a wide variety of optimization algorithms [5].

- **Sigmoid Unit**
  Whenever we have binary label variable, for example a classification problems between only two classes, it's preferred to calculate the output as follow:

$$\hat{y} = \sigma(\vec{z}) \tag{1.5}$$

  We can think of the sigmoid output unit as having two components. First, it uses a linear layer to compute $\vec{z} = \vec{w}^T\vec{x} + \vec{b}$. Next, it uses the sigmoid activation function to convert $\vec{z}$ into a probability. The sigmoid activation function saturates to 0 when $\vec{z}$ becomes very negative and saturates to 1 when $\vec{z}$ becomes very positive. The gradient can be too small to be useful for learning when this happens, whether the model has the correct answer or the incorrect answer. For this reason, maximum likelihood is almost always the preferred approach to training sigmoid output units. So the right approach is based on using sigmoid output units combined with maximum likelihood to ensure there is always a strong gradient whenever the model has the wrong answer.

- **SoftMax Unit**
  Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function:

$$\hat{y} = \frac{e^{\vec{z}_i}}{\sum_j e^{\vec{z}_j}} \tag{1.6}$$

  where as explain before first of all we compute $\vec{z}$ and then the $\mathsf{softmax}(\vec{z})$. This can be seen as a generalization of the sigmoid function but are more often used as the output of a classifier, to represent the probability distribution over n different classes.

Layers connections

The most common organization of connections is the Fully Connected Layer in which neurons between two adjacent layer are fully pairwise connected and neurons within the layer share no connections (Figure 1.3).

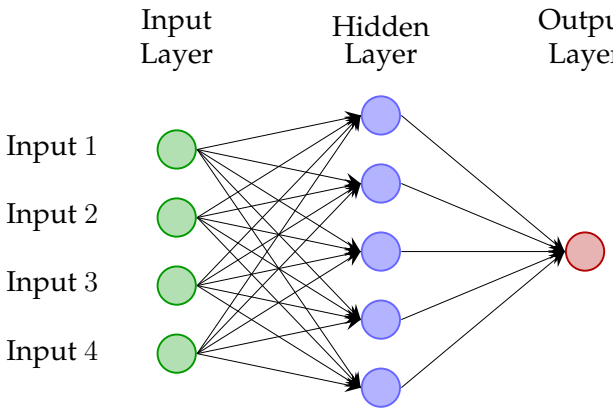


Figure 1.3: An example of a 2-layer neural network with Fully Connected Layers. There are the first input layer with 4 input data, one hidden layer with 5 units and an output; connections are between the layers but not within a layer.

Fully connected are the most popular kind of layer in machine learning architecture but it'is not the most useful, and nether the must used, in image classification, because of the huge number of parameters that it has to learn.

1.2 Loss functions and regularization

In a learning problem the loss function measures the compatibility between a prediction and the ground truth label. It is a non-negative value, where the robustness of model increases along with the decrease of the value of loss function. The data loss takes the form of an average over the data losses for every individual example:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{1.7}$$

where N is the total number of examples. We can choose between a lot of different definition for loss function, here follow some popular examples [7].

- **L_1**
  It's a quantity used to measure how close predictions are to the real class:
  $$L_i = \vec{y_i} - f(\vec{x_i}) \tag{1.8}$$
  It represents the very first definition for a loss function but it's preferred to use other functions that resulted better for the optimization step.

- **L_2**
  It's the squared version of the previous definition and it's widely used in linear regression:
  $$L_i = (\vec{y_i} - f(\vec{x_i}))^2 \tag{1.9}$$

- **Logarithmic Error**
  It is usually used when you do not want to penalize huge differences in predicted and actual values when both predicted and true values are huge numbers, it penalizes under-estimates more than over-estimates:
  $$L_i = (log(\vec{y_i} + 1) - log(f(\vec{x_i}) + 1))^2 \tag{1.10}$$

- **Cross Entropy**
  It's commonly-used in binary classification, measures the divergence between two probability distribution: if the cross entropy is large, it means that the difference between two distribution is large, while if the cross entropy is small, it means that two distribution is similar to each other:
  $$L_i = -\left[\vec{y_i} log(f(\vec{x_i})) + (1 - \vec{y_i}) log(1 - f(\vec{x_i}))\right] \tag{1.11}$$

- **Hinge Loss**
  It's used for "maximum-margin" classification, most notably for Support Vector Machines. For a binary classification it is defined as:

$$L_i = max\{0, 1 - \vec{y_i} \cdot f(\vec{x_i})\} \tag{1.12}$$

  but a more general expression exists with a margin m customize value:

$$L_i = max\{0, m - \vec{y_i} \cdot f(\vec{x_i})\} \tag{1.13}$$

- **Squared Hinge Loss**
  It's a variant of Hinge Loss, it solves the problem in hinge loss that the derivative of hinge loss has a discontinuity:

$$L_i = (max\{0, 1 - \vec{y_i} \cdot f(\vec{x_i})\})^2 \tag{1.14}$$

In all these definitions a problem may arise: supposing that it results $L = 0$, the weight matrix W is not necessary unique. We have $2W$ possible values of the this W. In fact if for a certain W we have $L = 0$ then any multiple of these parameters λW with $\lambda > 1$ will also give $L = 0$ because this transformation uniformally stretches all score magnitudes and hence also their absolute differences.

We can remove this ambiguity by extending the loss function with a *regularization penality*:

$$R(W) = \sum_k \sum_l W_{k,l}^2 \tag{1.15}$$

So the full loss becomes:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W) \tag{1.16}$$

where λ it's an hyperparameter usually determined by cross-validation. There are many desirable properties to include the regularization penality. The most appealing property is that penalizing large weights tends to improve generalization because it means that no input dimension can have

a very large influence on the score by itself. This penality prefers smaller and more diffuse weight vectors: the final classifier is encouraged to take into account all input dimensions to small amount rather than a few input dimension and very strongly. This effect can improve the generalization performance and lead to less overfitting.

1.3 The learning algorithms

Learning is the means of attaining the ability to perform a task [5]. A machine learning algorithm is an algorithm that leads computer program learn from experience, so the performance of the program measured to achieve a task improves with experience.

The performance can be measured calculating the accuracy of the model, that is defined as a portion of the examples for which the model produces the correct output. Another equivalent method to measure the performance is to calculate the error rate usually called $0 - 1$ loss: it's 0 when the classification is correct and 1 when it's incorrect.

To measure the accuracy we have to calculate the loss function defined above.

The kind of experience that is strictly connected with the calculation of the performance can be different depending on the kind of data set. If we have a data set containing many features and we have to extract information about the set itself from them, then we need unsupervised learning algorithm. Otherwise if we have a data set containing both features and label associated at each example then we need supervised learning algorithms.

Unsupervised learning involves observing several examples of an input vector $\vec{x}$ and learn the probability distribution $p(\vec{x})$ or some other interesting properties of that distribution. In this case the model should show the hidden structure of the data set.

Supervised learning should predict $\vec{y}$ from $\vec{x}$ estimating $p(\vec{y}|\vec{x})$, observing several examples of vector $\vec{x}$ and associated value of a vector $\vec{y}$.

Unsupervised and supervised learning are not formally defined terms, moreover other variants of learning paradigm are possible with a fixed

data set and also with a not fixed data set (reinforcement learning).

The aim of a machine learning algorithm is to lead the machine learn in order to perform well on new, previously unseen inputs.

This ability is called *generalization*.

The main steps of such an algorithm are:

1. train a data set to learn parameters;
2. learn parameters as to minimize training error (*optimization problem*);
3. use the model to classify new data;
4. modify the model to achieve minimum test error (*generalization problem*).

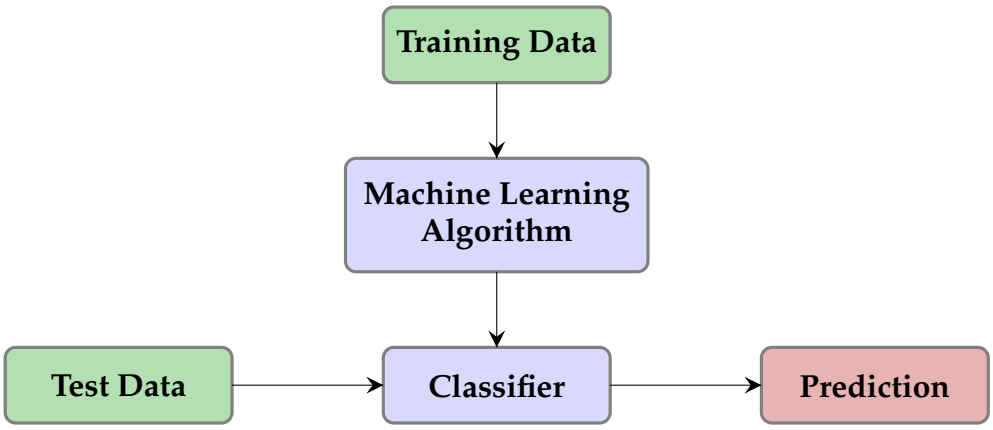


Figure 1.4: This is a scheme about pattern recognition cycle. We can see two kind of data: training and test. The first are used to train the model with a chosen machine learning algorithm. After that the test data are given in input to the classifier created after learning and we get prediction as output.

The factors determining how well a machine learning algorithm will perform are its ability to make training error small and make the gap between training and test error small. These two factors correspond to the two central challenges in machine learning: underfitting and overfitting.

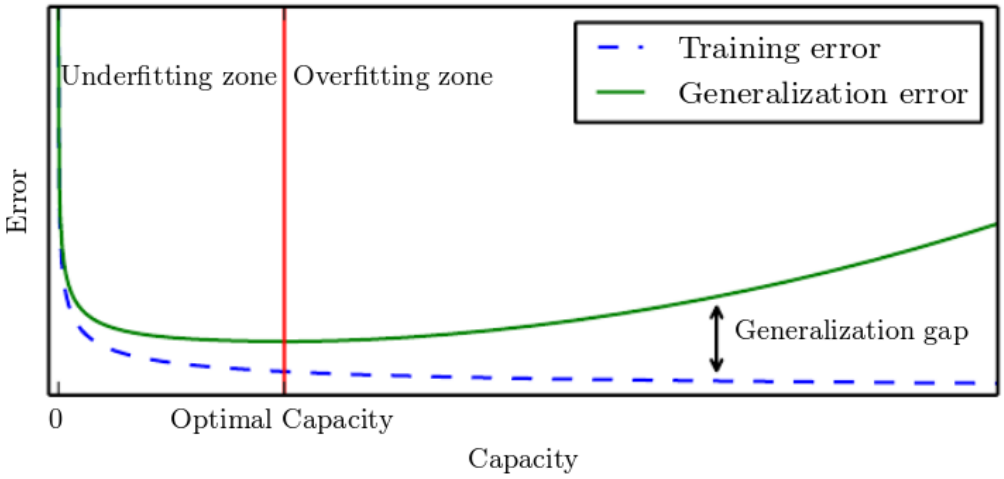


Figure 1.5: Typical relationship between capacity and error. and test error behave differently. At the left end of th graph, training error and generalization error are both high. This is the underfitting regime. As we increase capacity training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs, the decrease in training error, and we enter the overfitting regime, where capacity is too large, above the optimal capacity. [5]

The variables that influence the performance of the models are the architecture and the method used to compute the optimization. In other words the parameters like the number of layers, the number of units per layer, the connections between layers, and the algorithm chosen to find out weight parameters computes by the layers.

1.4 Optimization

The optimization it's the process of finding the set of parameters that minimize the loss function $L(f(\vec{x}))$ by altering $\vec{x}$. Compute the minimum means calculate the derivative, then optimization is the updating of the parameters moving on the opposite sign of the derivative. When this derivative is zero we don't have any information about where to move to minimize the loss, we are not sure that that point is a global minimum because it could be a local minimum or a saddle point. Those kind of points are called critical and they make the optimization difficult. This is

the reason why we try to find the best way to achieve the smallest value for L, that is very low but not necessary minimal. We often use functions that have multiple inputs so we refer at this technique with *gradient descent* because the gradient is the operation that generalize the derivative.

With gradient-based methods we would like to find the direction in which the function decrease faster, moving in the direction of the negative gradient:

$$\vec{x}' = \vec{x} - \epsilon \nabla f(\vec{x}) \tag{1.17}$$

where ϵ is the learning rate, a positive scalar determining the size of the step. All of these concepts are completely general for machine learning, but for neural networks in particular there are some problems. The non linearity of a neural network causes most interesting loss functions to become non convex: convex optimization converges starting from any initial parameters while non convex loss functions have no such convergence guarantee and sensitive to the values of initial parameters. To compute the gradient it's always used the *back-propagation* algorithm that allows the information from the loss to then flow backward through the network (Figure 1.6). This term refers only to the method for computing the gradient,

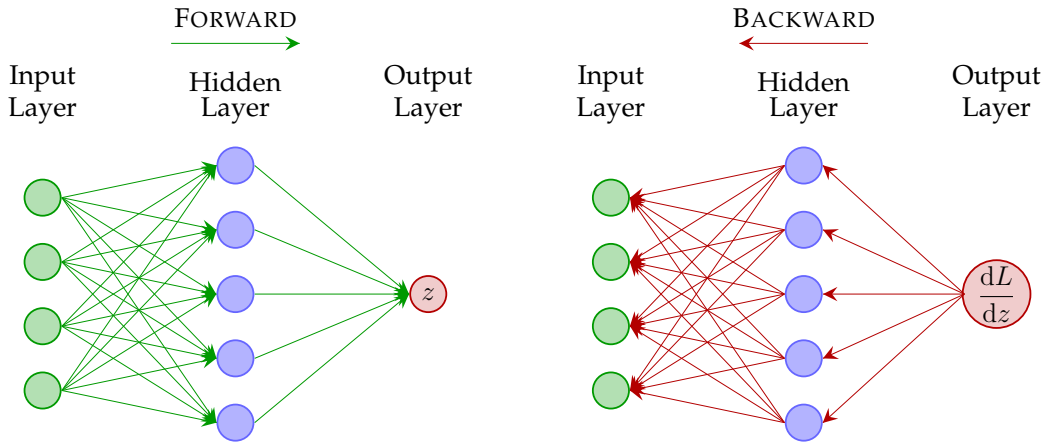


Figure 1.6: The forwardpass on the left in calculates z as a function $f(\vec{x})$ using the input variables x and y. The right side of the figures shows the backwardpass. Receiving $\mathrm{d}L/\mathrm{d}z$, the gradient of the loss function with respect to z from above, the gradients of x and y on the loss function can be calculate by applying the chain rule, as shown in the figure.

while other algorithm are used to perform learning using gradient. The gradient we require is the gradient of the loss function with respect to the parameters. The chain rule of calculus is used to computed the derivatives of functions formed by composing other functions whose derivatives are known. Back-propagation is an algorithm that computes the chain rule with a specific order of operations that is highly efficient. It is straightforward to calculate analytically the gradient but evaluating it in a computer introduces some difficulties. In fact in a net we have that many expressions may be repeated several times, and we don't want to repeat the same calculation more than one time, both for memory and time reason. Any procedure that computes the gradient will need to choose whether to store these expressions or to recompute them several times. The amount of computation required for performing the back-propagation scales linearly with the number partial derivatives as well as performing one multiplication and one addition. The back-propagation algorithm is designed to reduce the number of common expressions without regard to memory, avoiding the exponential explosion in repeated expressions.

To compute the gradient of some scalar z with respect to one of its ancestors x in a graph, we begin by observing that the gradient with respect to z is given by $\frac{\mathrm{d}z}{\mathrm{d}z} = 1$. We can then compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z. We continue multiplying by Jacobians, traveling backward through the graph in this way until we reach x . For any node that may be reached by going backward from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

Hence, it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over batches of the training data. The reason this works well is that the examples in the training data are correlated. The data set would not contain duplicate images, the gradient from a mini-batch is a good approximation of the gradient of the full objective. Therefore, much faster convergence can be achieved in practice by evaluating the mini-batch gradients to per-

form more frequent parameter updates. The extreme case of this is a setting where the mini-batch contains only a single example.

All m samples in the training set could be identical copies of each other. A sampling-based estimate of the gradient could compute the correct gradient with a single sample, using m times less computation than the naive approach. In practice, we are unlikely to encounter this worst-case situation, but we may find large numbers of examples that all make very similar contributions to the gradient.

Optimization algorithms that use the entire training set are called batch or deterministic gradient methods, because they process all the training examples simultaneously in a large batch. This terminology can be somewhat confusing because the word "batch" is also often used to describe the minibatch used by minibatch stochastic gradient descent. Typically the term "batch gradient descent" implies the use of the full training set, while the use of the term "batch" to describe a group of examples does not. For example, it is common to use the term "batch size" to describe the size of a minibatch.

Optimization algorithms that use only a single example at a time are sometimes called stochastic and sometimes online methods. This is usually reserved for when the examples are drawn from a stream of continually created examples rather than from a fixed-size training set over which several passes are made.

Most algorithms used for deep learning fall somewhere in between, using more than one but fewer than all the training examples. These were traditionally called minibatch or minibatch stochastic methods, and it is now common to call them simply stochastic methods.

The training algorithm is almost always based on using the gradient to descend the loss function in one way or another, they usually are improvements and refinements on the ideas of gradient descent, all implemented with back-propagation.

To follow the gradient we can use the following basic algorithms.

- **Stochastic Gradient Descent (SGD)**
  It applied to non convex loss functions has no such convergence guarantee and is sensitive to the values of the initial parameters. it is

possible to obtain an unbiased estimate of the gradient by taking the average gradient on a minibatch of m examples drawn i.i.d from the data-generating distribution. A crucial parameter for the SGD algorithm is the learning rate. It used a learning rate that has to gradually decrease over time, so we denote it at iteration k as ϵ_k. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum. A sufficient condition to guarantee convergence of SGD is that

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty \tag{1.18}$$

- **Momentum**
  The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move. The name momentum derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion. Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector v may also be regarded as the momentum of the particle. A hyperparameter $\alpha \in [0, 1]$ determines how quickly the contributions of previous gradients exponentially decay. The update rule is given by:

$$\vec{v} = \alpha\vec{v}.\epsilon\boldsymbol{\nabla}_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\vec{x}_i, \theta), y_i) \right) \tag{1.19}$$

$$\theta = \theta + \vec{v} \tag{1.20}$$

- **Nesterov momentum**
  It is a variant of the momentum algorithm that was inspired by Nes-

terov's accelerated gradient method:

$$\vec{v} = \alpha\vec{v}.\epsilon\boldsymbol{\nabla}_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\vec{x}_i, \theta + \alpha\vec{v}), y_i) \right) \tag{1.21}$$

$$\theta = \theta + \vec{v} \tag{1.22}$$

The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum, the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard method of momentum. In the stochastic gradient case, Nesterov momentum does not improve the rate of convergence.

Neural network researchers have long realized that the learning rate is reliably one of the most difficult to set hyperparameters because it significantly affects model performance. Other algorithm involves parameters initialization strategies are new techniques of adaptive learning like **AdaGrad**, **RMSProp**, **Adam**. Moreover to achieve optimization we can work at another level with **batch normalization** and the study of model's design. In large-scale applications the training data can have on order of millions of examples. Batch normalization is one of the most exciting recent innovations in optimizing deep neural networks, and it is actually not an optimization algorithm at all. Instead, it is a method of adaptive reparametrization, motivated by the difficulty of training very deep models. Previous approaches had involved adding penalties to the cost function to encourage units to have normalized activation statistics or involved intervening to re-normalize unit statistics after each gradient descent step. In a deep neural network with nonlinear activation functions, the lower layers can perform nonlinear transformations of the data, so they remain useful. Batch normalization acts to standardize only the mean and variance of each unit in order to stabilize learning, but it allows the relationships between units and the nonlinear statistics of a single unit to change.

1.5 Convolutional Neural Networks

Convolutional Neural Networks are inspired from biology and have played an important role in the history of deep learning. Neuroscientific studies say that there is a part of the brain called V1, located at the back of the head also known as the primary visual cortex, that is the first area of the brain that begins to perform significantly advanced processing of visual input. We focus on this area to list the analogies between CNNs and human brain. A convolutional network layer is designed to capture three properties of V1: spatial map as features defined in terms of two-dimensional maps; simple cells as detection units of receptive field; complex cells as pooling units.

They are a key example of a successful application of insights obtained by studying the brain to machine learning applications. They were also some of the first deep models to perform well, they were some of the first neural networks to solve important commercial applications and remain at the forefront of commercial applications of deep learning today. They were also used to win many contests.

Convolutional networks were more computationally efficient than fully connected networks, so it was easier to run multiple experiments with them and tune their implementation and hyperparameters. Larger networks also seem to be easier to train. With modern hardware, large fully connected networks appear to perform reasonably on many tasks. It may be that at the early stage neural networks weren't expected to work, nether by the people who developed them. Convolutional networks provide a way to specialize neural networks to work with data that has a clear structured topology and to scale such models to very large size. This approach has been the most successful on two-dimensional image.

Convolution Networks have provided the rise of important ideas that help to improve machine learning systems: sparse interactions and parameter sharing. Traditional neural network layers has output unit interacting with every input unit while CNNs have sparse interactions: the input image can detect small, meaningful features such as edges with kernels smaller than input. This means that thanks to CNNs we need to store

fewer parameters and we need fewer operations to compute the output. Parameter sharing refers to using the same parameter for more than one function in a model, so rather than learning a separate set of parameters for every location, we learn only one set.

It turns out that we can dramatically reduce the number of parameters by making one reasonable assumption that if one feature is useful to compute at some spatial position (x, y), then it should also be useful to compute at a different position $(x_2, y : 2)$. In other words, denoting a single 2-dimensional slice of depth as a depth slice, we are going to constrain the neurons in each depth slice to use the same weights and bias. In practice during backpropagation, every neuron in the volume will compute the gradient for its weights, but these gradients will be added up across each depth slice and only update a single set of weights per slice.

Notice that if all neurons in a single depth slice are using the same weight vector, then the forward pass of the Conv layer can in each depth slice be computed as a convolution of the neuron's weights with the input volume. This is why it is common to refer to the sets of weights as a filter, or a kernel, that is convolved with the input.

However sometimes the parameter sharing assumption may not make sense. This is especially the case when the input images to a ConvNet have some specific centered structure, where we should expect that completely different features should be learned on one side of the image than another. One practical example is when the input are faces that have been centered in the image. You might expect that different eye-specific or hair-specific features could be learned in different spatial locations. In that case it is common to relax the parameter sharing scheme, and instead simply call the layer a Locally-Connected Layer.

CNNs are trained with the well known back-propagation algorithm but differs from other deep neural networks in the architecture. They are designed to recognize visual patterns directly from pixel images with minimal pre-processing, they are made with the assumption that the inputs are images. They can recognize patterns with extreme variability and with robustness to distortions and simple geometric transformations. The main differences between a standard neural network and a CNN are in

the layers' dimensions, connections, functions. In Figure 1.7 we can see a comparison of the architectures in which it is emphasized the dimensions and the connections.

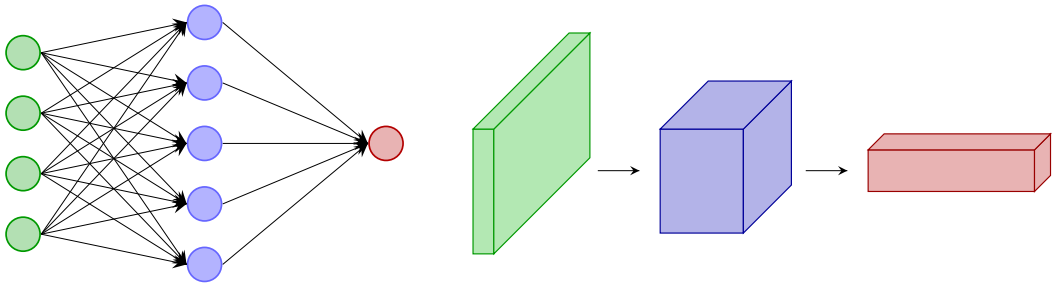


Figure 1.7: A standard 3-layer Neural Network compared with a Convolutional Neural Network that arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activation. In this example, the green input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

The architecture of a CNN is made up of 4 types of layer:

1. Input;
2. Convolutional;
3. Pooling;
4. Fully Connected.

A typical layer of a convolutional network consists of three stages. In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the ReLU function. This stage is sometimes called the detector stage. In the third stage, we use a pooling function to modify the output of the layer further. A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. Below follows a complete description of the 2 characteristic layers: convolutional and pooling.

Convolutional Layer

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting. Conv layer's parameters consist of a set of learnable filters. Every filter is small spatially but extends through the full depth of the input volume. During the forward pass, we *slide*, more precisely *convolve*, each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position.

As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position.

The network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer. Then we will stack all the activation maps along the depth dimension and produce the output volume.

Using the brain/neuron analogies, every entry in the $3D$ output volume can also be interpreted as an output of a neuron that looks at only a small region in the input and shares parameters with all neurons to the left and right spatially.

One of the main characteristics of CNNs is how layers are connected each other. Differently from fully connected layers, Conv layers allow to connect each neuron with only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called *receptive field* of the neuron, or *filter size*. The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize this asymmetry in how we treat the spatial dimensions, width and height and the depth dimension: the connections are local in space, along width and height, but always full along the entire depth of the input volume.

These connections produce a $2D$ activation map, one for each filter, that gives the responses of that filter at every spatial position.

Another important parameter is the number of units arranged in the output volume. Three hyperparameters control the size of the output vol-

ume: the **depth**, **stride** and **zero-padding**.

- **Depth**
  The depth of the output volume corresponds to the number of filters we would like to use, each learning to look for something different in the input, for example the presence of various oriented edges or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a *depth column*.

- **Stride**
  The stride defines how much we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially. Sometimes it will be convenient to pad the input volume with zeros around the border. The size of this zero-padding is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes, commonly we will use it to exactly preserve the spatial size of the input volume so the input and output width and height are the same). We can compute the spatial size of the output volume as a function of the input volume size (W), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border. The correct formula for calculating how many neurons fit is given by

$$N = \frac{(W - F + 2P)}{S + 1} \tag{1.23}$$

- **Zero-Padding**
  Setting zero padding to be $P = (F-1)/2$ when the stride is $S = 1$ ensures that the input volume and output volume will have the same size spatially. It is very common to use zero-padding in this way. Zero padding the input allows us to control the kernel width and the size of the output independently. Without zero padding, we are forced to choose between shrinking the spatial extent of the network rapidly and using small kernels, both scenarios that significantly

limit the expressive power of the network.

There are two special cases of the zero-padding setting: one is the extreme case in which no zero padding is used whatsoever, and the convolution kernel is allowed to visit only positions where the entire kernel is contained entirely within the image. In code terminology, this is called *valid convolution*. In this case, all pixels in the output are a function of the same number of pixels in the input, so the behavior of an output pixel is somewhat more regular.

Another special case of the zero-padding setting is when just enough zero padding is added to keep the size of the output equal to the size of the input. In code one calls this *same convolution*. The input pixels near the border influence fewer output pixels than the input pixels near the center: this can make the border pixels somewhat underrepresented in the model.

Usually the optimal amount of zero padding, in terms of test set classification accuracy, lies somewhere between *valid* and *same* convolution.

Note again that the spatial arrangement hyperparameters have mutual constraints. In some cases it would be impossible to use even stride, since the output dimension becomes not an integer. Therefore, this setting of the hyperparameters is considered to be *invalid*. The use of zero-padding and some design guidelines will significantly alleviate these kind of problems.

Pooling Layer

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and

height, discarding 75% of the activations. Pooling sizes with large receptive fields are too destructive. Every MAX operation would in this case be taking a max over 4 numbers, little 2×2 region in some depth slice. The depth dimension remains unchanged.

Pooling units can also perform other functions, such as average pooling or even L_2-norm pooling. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice. In all cases, pooling helps to make the representation approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is. For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face.

During the forward pass of a pooling layer, it is common to keep track of the index of the max activation, sometimes also called the switches, so that gradient routing is efficient during backpropagation. Many people dislike the pooling operation and think that we can get away without it. Discarding pooling layers has also been found to be important in training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs). It seems likely that future architectures will feature very few to no pooling layers.

For many tasks, pooling is essential for handling inputs of varying size. For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of an offset between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size. Some theoretical work gives guidance as to which kinds of pooling one should use in various situations. It is also possible to dynamically pool features together, for example, by running a clustering algorithm on the locations of interesting features. This approach yields a

different set of pooling regions for each image. Another approach is to learn a single pooling structure that is then applied to all images.

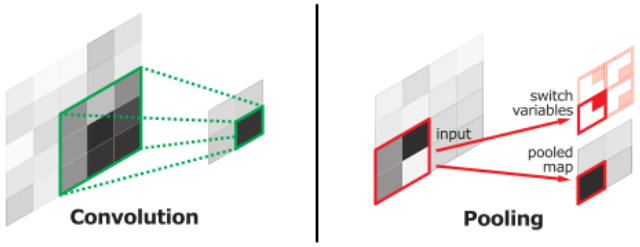


Figure 1.8: Illustration of convolution and pooling operations. [8]

1.6 What is wrong with CNNs?

CNNs are good at speech recognition and object classification, but they are unlike brain. If the aim of deep learning is to emulate and reproduce brain learning we are far from this. From this point of view we can say that today neural networks don't work as well as they could. The main difference is that they have very few level of structure, like neurons, layers and the whole net. There in no *entity* into this kind of architecture.

G. Hinton believes that we need to group neurons in each layer into "capsules" which are able to do a lot of internal computation and to output a compact result. This capsules represents the entities which we are looking for to make deep learning more similar to brain learning, they are like brain *cortical minicolumn*[2].

A capsule take predictions from low-level capsules about what their generalize pose should be, so about multi-dimensional vector, and they look for predictions that agree tightly. They don't care if there's a lot of predictions that outliers or they concerned with or if there is a small subset of predictions that agree well.

[2]A cortical minicolumn is a vertical column through the cortical layers of the brain. Minicolumns comprise perhaps $80 - 120$ neurons, except in the primate primary visual cortex where there are typically more than twice the number. All of them have the same receptive field; adjacent minicolumns may have different fields. There are about 2×108 minicolumns in humans.

This entity has two kind of association parameters: a value for its own presence and some properties, like orientations, size, velocity, color, in a hierarchical order, from low-level to high-level. Capsule output is then the probability that the entity is present and the generalize pose of this entity, which in vision in going to be an object or part of an object. The brain need to do this compute, it really has to take predictions from low-level to high-level in a similar way.

CNNs on the other hand work with multiple layers of learned feature detectors, interconnected with max-pooling layers or average pooling layers that take only the most active neuron. These feature detectors are local and their spatial domains get bigger in higher layers, but they are interleaved with subsampling layers that pool the outputs of nearby feature detectors of same type.

Pooling process gives a small amount of translational invariance at each level and reduces the number of input to the next layer of feature extraction.

The activations in the last hidden layer of a deep ConvNet are a precept that contains information about many of the objects in the image but without any spatial relationship between those objects. This spatial information is loosed after pooling. Internal data representation of a convolutional neural network does not take into account important spatial hierarchies between simple and complex objects.

This is the reason why Hinton talked about four arguments against pooling and here we try to understand each of one in order to point out the necessity to introduce Capsule Networks.

Intrinsic coordinate

The first argument against pooling is that it is a bad fit to the psychology of shape perception: it does not explain *why* we assign intrinsic coordinate frames to objects and *why* they have such huge effects. When people do shape perception they do it by imposing rectangular coordinate frames on things and if they take the same object and impose a different rectangular coordinate frame they don't even realize it's the same object.

It's a huge effect: we can't say how the same pixels can be processed completely differently depending on the coordinate frame because they have no notion of imposing coordinate frame. Human vision system imposes rectangular coordinate frames on objects in order to represent them and it would perform mental rotation on the object to a point of reference which it's familiar to before making the comparison. Computer graphics has to say what the relation is between a part in the whole, so it put a frame on the part and tells you the matrix that will map point in the whole relative frame. The relationship between an object and the viewer is represented by a whole bunch of active neurons that capture different aspects of the relationship not by a single neuron or a coarse-coded set of neurons.

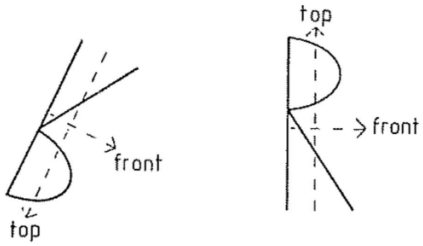


Figure 1.9: An example of intrinsic coordinates. Everyone can experience mental rotation to proof that the first image is not representing a letter R but it's a mirror of it. [2]

So the conclusion is that human vision uses coordinate frames embedded in object and embedding parts of objects and represents those coordinate frames but, if it represents the pose of the object, the relationship between its embedded coordinate frame and the viewer spreads over a bunch of numbers, not just in one.

Invariance

The second argument against pooling is that the neural network it's solving the wrong problem because we don't want the neural activities to be invariant to viewpoint, we want the knowledge to be invariant to viewpoint. So Hinton talk about the *equivariance*: changes in viewpoint lead to corresponding changes in neural activities. From one hand we want

that label doesn't change with viewpoint, so the final label needs to be viewpoint-invariant, but we want a representation where as you change viewpoint then it change just like the viewpoint does (Figure 1.10).

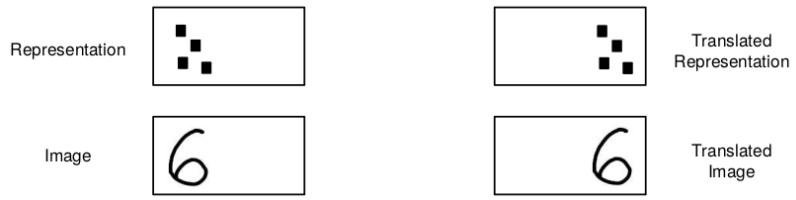


Figure 1.10: CNNs can give 'place-coded' equivariance for discrete translations like this one in which change in pixel positions leads to change in representation. [2]

There are two types of equivariance: place-coded and rate-coded. The first is about translating by a whole number of pixels that leads representation change as well. The second it's verified when an object is moved around and the same neurons are encoding it but with a change in activities. The interpretation is that at low level we have very small domains where tiny changes could change the rates; if these changes are more and more big the change is in neurons and then in another bunch of neurons, running from a domain to another. If we moved at high levels we can't go a long way without changing neurons are coding it but the activities of neurons changes to tell you where it is. So the idea is that higher-level capsules have bigger domains so low-level 'place-coded' equivariance gets converted into high-level 'rate-coded' equivariance.

So when detected feature moves around the image or its state somehow changes, the probability still stays the same (length of vector does not change), but its orientation changes.

Absence of linear structure

The worse property of CNNs is they fail to use and underlying linear structure, they don't make use of the natural linear manifold that perfectly handles the largest source of variance in images. Spatial structure is modeled by matrices, viewpoint invariant, that represent the transfor-

mation from a coordinate frame embedded in the whole to a coordinate frame embedded in each part. In fact a CNN, to learn the invariance, has to train all along different viewpoints so it requires a lot of training data and consequently a lot of training time. What is missing is a built-in bias that could generalize in the right way across viewpoint.

A better approach would be to use the pose information to get everything to be linear to better extrapolate.

For many years people would be saying you could think of vision as inverse graphics, but didn't mean it literally. Anyway in a computer system one can do graphics backwards literally. Graphics programs use hierarchical models in which spatial structure is modeled by matrices that represent the transformation from a coordinate frame embedded in the whole to a coordinate frame embedded in each part. These matrices are totally viewpoint invariant and this representation makes it easy to compute the relationship between a whole and the viewer starting from the relationship between a part and the viewer. We can represent the relationship between a whole and a part as a matrix of weights, which is completely viewpoint invariant. The pose is the same of this matrix, to take suppose of the whole and gives you the pose of the part. We have a capsule, a bunch of neurons, and the activities of neurons represent different properties. With this architecture we have that a higher level visual entity is present if several lower visual entities agree on their predictions for its pose.

As an example we can see the prediction of a face in the known Picasso's Problem (Figure 1.11).

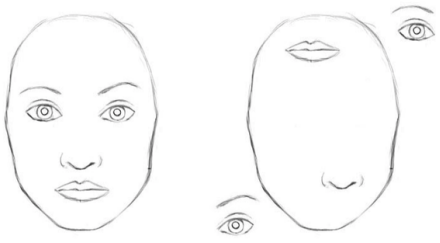


Figure 1.11: A representation of the so-called Picasso's Problem. A CapsNet would be able to distinguish between the two images while a CNN would say that both of them are facec. [9]

From one hand it is an old process in which we have fairly low dimensional features trying to predict the high dimensional poses and if the dimensionality of the features is lower than dimension of the object you have to predict a subspace and it need an array to accumulate results. In this new way proposed the features have as many dimensions as the high dimensional things and to get features you can reliably extract from pixels: you can finally make point prediction.

Routing

The last thing is that pooling is a very primitive way to do dynamic routing. Hinton proposed a routing principle where you route the information to the capsule that knows how to deal with it. So the idea is that we assume that each part that we discover has one parent, that's a single parent constraint, or possibly no parents; so we want to model the process as a parse tree and we want to find what is a part of the tree and what is not.

When you discover a pilot (like a circle ...) and you don't know from this of what is a part, then you take the poses it sends to all the possible places. To send is a kind of weighting and with it then you can choose your *bet*. Each higher-level capsule receives different predictions for what the pose of the input vector should be. Each of these predictions has a voting-strength, a weight, between 0 and 1 which is called a *bet*. So the capsule looks at all these incoming weak bets and find a bunch that agree and, when it finds it, then the low level capsule sends its pose to several high level capsules weighting by the prior. So there's going to be a prior that the pilot is might be a part of an object and this prior influenced the first sending process.

A *bet* is treated as a fraction of an observation. The higher-level capsule tries to find a subset of the predictions that agree well and it throws out the remaining predictions. It gets a score which is big if many predictions agree well. In this case we should be able to model the predictions much better using mixture model. We are going to get a score for how good a cluster is. In the capsules you find the clusters: you send your prediction and verify if it belongs to the cluster or not.

With this routing you send top-down feedback and that's very different from backpropagation. It's based on the agreement between all the possible prediction so it is called *routing-by-agreement*.

1.7 Capsules: simulating human vision

The substantial novelty of the proposed Capsule Networks is in the conceptual changing introduced with the definition of capsule as entity in the net. Basically the novelty is in the structure is the *building block* of the net: it is no more a neuron, as in all the others neural networks, but it's a capsule. The most important difference between a neuron and a capsule is that the first works with scalar input-output while the other with vector input-output. Another important novelty is not pretty structural but about computational complexity, with the introduction of the *routing-by-agreement* algorithm in stead of the pooling and the scalar product. These novelties lead to the solution of the Picasso's problem and give importance to the spatial relationship between the objects.

The main characteristic of the CapsNet, which distinguish it from all the other nets, is the use of equivariance in stead of invariance. So the vector that represent the presence's probability of the entity and all the properties of this entity will change with spatial transformation (like translation and rotation). This change will be not in magnitude but in direction, in a linear way with the transformation. For example if at a first low-level basic shapes (like nose, eye, mouth ...) are recognized, in a second high-level the agreement between all the prediction for the whole object (like human face) are computed, at which all the parts previously analyzed belongs to. It's considering the spatial relationship between all the parts that the CapsNet will be able to label the whole object (to say if it's a face or not).

We can summarize the main differences between CNNs and CapsNets in the following table:

CNN	CapsNet
neuron building block	capsule building block
scalar-to-scalar	vector-to-vector
ReLU	squashing
pooling	routing-by-agreement
invariance	equivariance
Picasso's problem	spatial relationship
far from human vision	closer to human vision
needs a lot of training data	needs less training data
few training time	longer training time

So in stead of the invariance in neurons activities that used scalar input-output Hinton proposed equivariance in capsules' *instantation parameters* which used vector input-output with a dynamic routing that requires less data and more time to do the prediction.

1.8 The implementations

The idea behind Capsule Networks is really simple and Hinton has been thinking about this for decades. He provided a formal definition for *capsule* and a basic idea of how to implement this new architecture in [10] in 2011.

The reason why there were no publications before last year is simply because there was no technical way to make it work before [9]. One of the reasons is that computers were just not powerful enough in the pre-GPU-based [3] era before around 2012. Another reason is that there was no algorithm that allowed to implement and successfully learn a capsule network. In the same fashion the idea of artificial neurons was around since

[3] A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. Their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms that process large blocks of data in parallel.

1940-s, but it was not until mid 1980-s when backpropagation algorithm showed up and allowed to successfully train deep networks.

So the idea of capsules itself is not that new but there was no algorithm up for 6 years to make it work. This algorithm is called "dynamic routing between capsules" and was published from Hinton in 2017 in [11]. This algorithm allows capsules to communicate with each other and create representations similar to scene graphs in computer graphics.

Then other progresses were made until now, in fact another implementation was proposed in 2018 based on matrices and then people starts to use CapsNets approach also to segmentation task.

Here we go through the first three implementation proposed from Hinton: transforming auto-encoders ([10]), vector capsules ([11]), matrix capsules ([12]).

Transforming auto-encoders implementation

The first architecture that used capsules as its fundamental building block was the transforming auto-encoder in [10]. In this early implementation based on Capsule Theory, Hinton proposed to train the primary capsules not by backpropagating the errors made in digit classification but by doing unsupervised learning. The idea is to extract pose information using a domain specific decoder producing an image by adding together contributions from each capsule. Each capsule learns a fixed template that gets intensity-scaled and translated differently for reconstructing each image. The encoder must learn to extract the appropriate intensity and translation for each capsule from the input.

In this early design a capsule use one of the values in its output vector to represent the probability that the entity exists, while the other values in the vector represent the instantiation parameters. The transforming auto-encoder is the part of the network that generates the instantiation parameters to be used by the capsules. It was not a network built to recognize objects in images, but rather to take an input image of an object and a pose for the object, and output an image of the same object in the given pose. So it's more properly a reconstruction of the image from the capsules.

Using capsules instead of neurons enabled an artificial network to easier understand the pose of the objects it tries to identify. One drawback of this architecture however is that the transforming auto-encoder needs the pose of the objects to be supplied externally.

A first representation of the capsule is given in Figure 1.12.

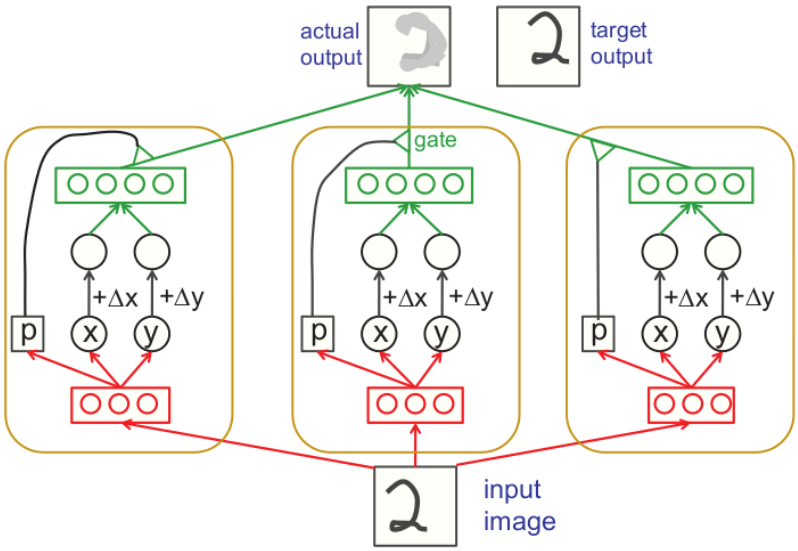


Figure 1.12: Graphical representation of three capsules of a transforming auto-encoder that models translations. Each capsule in the figure has 3 recognition units (red) and 4 generation units (green). The weights on the connections are learned by backpropagating the discrepancy between the actual and target outputs. [10]. This specific capsule design's only instantiation parameters are its x and y position

.

Here one can see that each capsule has its own logistic "recognition units" and its own "generation units". The first act as a hidden layer for computing three numbers: x, y, and p. These numbers are the outputs that the capsule will send to higher levels of the vision system. Here p is the probability that the capsule's visual entity is present in the input image. The second kind of units are used for computing the capsule's contribution to the transformed image.

The graphic system, after extracting the probability for an entity and its pose, translates the template according to the features and then adds it to the image. Each template learned by the auto-encoder is multiplied by a case-specific intensity and translate by a case-specific Δx, Δy, then it is added to the output image. After having done that unsupervised with learning the pose, the code can do supervised on top of this, modeling the outputs of the primary capsules with a factor analyzer. It concatenate all the parameters that have just been extracted in order to get a big vector. Doing factor analysis on this factor leads to find underlying factors, for the affine tranformation and for the deformation. The model is nicely because as you change your viewpoint all of the factors are changing, providing the equivariance.

Vector implementation and routing-by-agreement algorithm

The main characteristic of a Capsule Network is that both the inputs and the outputs are vectors. After the convolutional layer, data are re-scaled and taken as input vectors into capsules.

Each capsule has a vector output and each vector represents the entity's presence probability in module and its instantation parameters in direction. We want the length of the output vector of a capsule to represent the probability that the entity is present in the current input. This probability of presence is locally invariant. The unit vector that gives the direction is a generalize *pose*; these instantation parameters that defined the direction of the vector are equivariants.

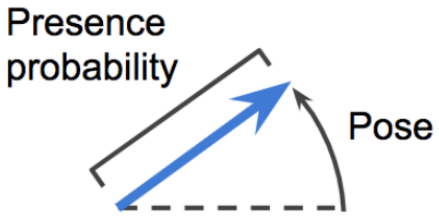


Figure 1.13: A capsule's output vector. [13]

All of these information stored by the output vector go as input in the digit caps, a layer made up of capsules in which the number of capsules represent the number of class, so its aim is to do the final classification.

It's a classification that goes from a *low-level* to an *high-level*, from the recognition of simple objects to the recognition of objects more and more complexes.

Taking into account the example of a layer with two capsules (Figure 1.14) we can understand how it works in details.

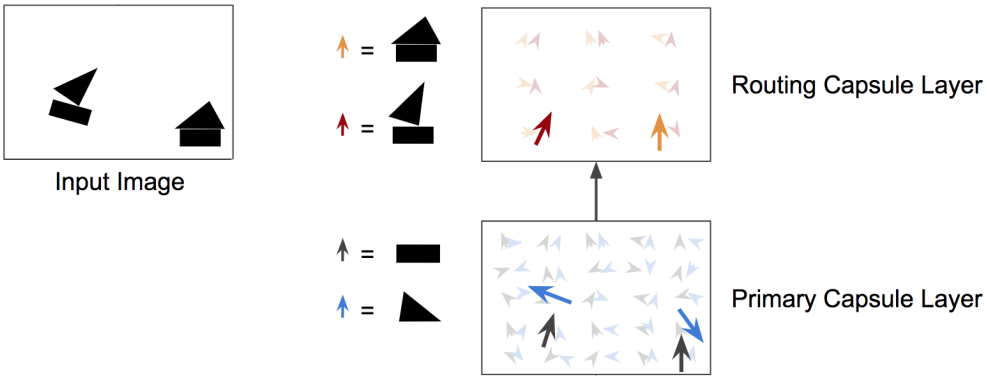


Figure 1.14: Example of a layer with two capsules. [13]

In this example the primary capsule layer is made up of two coupled capsules 5×5 : one represents the entity "triangle" the other the entity "rectangle". The second layer is made up of two capsules 3×3 representing one the entity "boat" and the other the entity "house".

The behavior that characterizes the capsules is the algorithm called *routing- by- agreement*. Its aim is to predict the presence's probability for the object's *pose*, taking into account the parts that composed the object. So it tries to found an *agreement* between all the predictions for the presence's probability of the parts. Following the previous example, we can see in Figure 1.15 how this routing works.

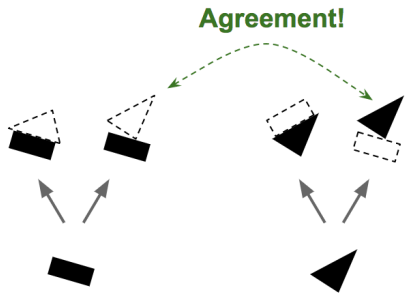


Figure 1.15: In this step all the predictions are compared to see if there is agreement between them. [13]

Once the rectangle and the triangle have been recognized with their respective *pose*, the algorithm goes on recognizing as parts of another bigger object, like a boat or a house. So it considers the two possibilities independently and then the option that gives a better agreement between the possibilities is chosen.

Weights associates at each prediction are updated based on the result obtained, so they increase if there is *agreement* and decrease if there is *disagreement* (Figure 1.16).

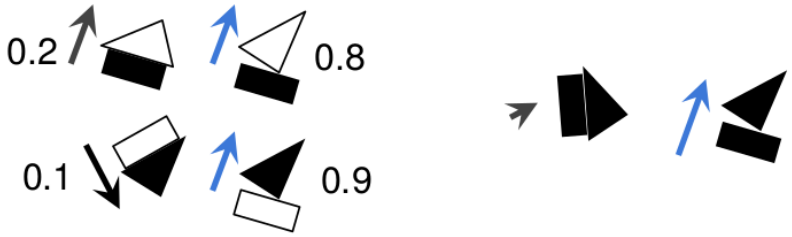


Figure 1.16: This is the step of the algorithm in which the weights are update. They increase or decrease according to the agreement. [13]

This model is described mathematically with vectors and the *routing-by-agreement* algorithm is represented in Figure 1.17.

So we can defined the output vector from the capsule i, that becomes input vector for the capsule j, as:

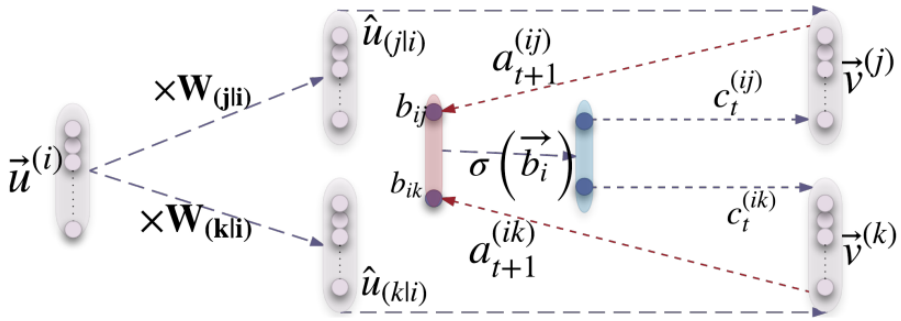


Figure 1.17: A scheme of the dynamic routing. [14]

$$\vec{s_j} = \sum_{i=1}^{N} c_{ij} \widehat{u}_{ji} = \sum_{i=1}^{N} c_{ij} W_{ij} \vec{u}_i \tag{1.24}$$

where sum is over all the prediction vectors $\hat{u}$ produced by multiplying the output $\vec{u}_i$ of a capsule in the layer below by a weight matrix W_{ij}.

The coupling coefficient determined by the iterative dynamic routing process measures the agreement between the correct output and the prediction, indeed it's determined by a "routing softmax" whose initial b_{ij} are the log probabilities that capsules are coupled:

$$c_{ij} = \frac{e^{b_{ij}}}{\sum_k e^{b_{ik}}} \quad \text{with} \quad \sum_i c_{ij} = 1 \tag{1.25}$$

where the initialization is $b_{ij} = 0$ so that at the beginning each vector is compared with all the capsules with the same probability, $c_{ij} = 0.5$. Recall that the length a capsule's output vector is interpreted as probability of existence of the feature that this capsule has been trained to detect and orientation of the output vector is the parameterized state of the feature, we can say that for each lower level capsule i, its weights c_{ij} define a probability distribution of its output belonging to each higher level capsule j. Then each vector obtains a unit form and doesn't change in direction, scaling as follows:

$$\vec{v_j} \ll 1 \rightarrow 0 \quad \text{e} \quad \vec{v_j} \approx 1 \rightarrow 1 \tag{1.26}$$

To ensure it a non-linear "squashing" function is applied to the input vector:

$$\vec{v_j} = \frac{\|\vec{s_j}\|^2}{1 + \|\vec{s_j}\|^2} \frac{\vec{s_j}}{\|\vec{s_j}\|} \tag{1.27}$$

So the aim is to compute the squash operation for all the vectors and then update the coefficients c_{ij} increasing it when the agreement is high.

The agreement is simply the scalar product $a_{ij} = \vec{v_j} \cdot \hat{u}_{ji}$. After the routing-by-agreement the coefficients are updated as follows:

$$b_{ij} = b_{ij} + a_{ij} \tag{1.28}$$

Here there is one of the most critical point, that characterize this approach from CNN. In fact in the classical approach we have the max-pooling technique that ignore all but the most active neuron. Here we want to preserve all the information, all the vectors, but with a change in the corresponding weight.

The update mechanism reminds the Hebbian learning rule, that specifies how much the weight of the connection between two units should be increased or decreased in proportion to the product of their activation. This rule was built starting from Hebbs's 1949 learning rule, which states that the connections between two neurons might be strengthened if the neurons fire simultaneously.

The Capsule Layer computes this routing procedure shown in Figure 1.17 that's written in details in Table 1.1.

The first line says that this procedure takes all capsules in a lower level l and their outputs $\hat{u}_{ji}$, as well as the number of routing iterations r. The very last line tells you that the algorithm will produce the output of a higher level capsule $\vec{v_j}$.

In the second line there is the coefficient b_{ij} that is simply a temporary value that will be iteratively updated: at start of training the value of b_{ij} is initialized at zero and, after the procedure is over, its value will be stored in c_{ij}.

Line 3 says that the steps in 4–7 will be repeated r times, where r is the

Routing

1. **procedure** Routing $(\hat{u}_{ji}, r, l)$
2. for all capsule *i* in layer *l* and capsule *j* in layer $(l+1)$: $b_{ij} = 0$
3. for *r* iterations do
4. for all capsule *i* in layer *l*: $c_i = \mathsf{softmax}(b_i)$
5. for all capsule *j* in layer $(l+1)$: $\vec{s_j} = \sum_{i=1}^{N} c_{ij}\hat{u}_{ji}$
6. for all capsule *j* in layer $(l+1)$: $\vec{v_j} = \mathsf{squash}(\vec{s_j})$
7. for all capsule *i* in layer *l* and capsule *j* in layer $(l+1)$: $b_{ij} = b_{ij} + \hat{u}_{ji} \cdot \vec{v_j}$

return $\vec{v_j}$

Table 1.1: Routing algorithm procedure as described in [11] .

chosen number of routing iterations.

Step in line 4 calculates the value of vector c_i which is all routing weights for a lower level capsule i. This is done for all lower level capsules. Then softmax will make sure that each weight c_{ij} is a non-negative number and their sum equals to one, providing probabilistic nature of the coupling coefficients.

At the first iteration, the value of all coefficients c_{ij} will be equal, because on line two all b_{ij} are set to zero, this represents the state of maximum confusion and uncertainty: lower level capsules have no idea which higher level capsules will best fit their output. Of course, as the process is repeated these uniform distributions will change.

After all weights c_{ij} were calculated for all lower level capsules, we look, in line 5, at higher level capsules. This step calculates a linear combination of input vectors, weighted by routing coefficients c_{ij}, determined in the previous step. Intuitively, this means scaling down input vectors and adding them together, which produces output vector $\vec{s_j}$. This is done for all higher level capsules.

Next, in line 6, vectors from last step are passed through the squash

non-linearity, that makes sure the direction of the vector is preserved, but its length is enforced to be no more than 1. This step produces the output vector $\vec{v_j}$ for all higher level capsules.

So the first steps simply calculate the output of higher level capsules while step on line 7 is where the weight update happens. This step captures the essence of the routing algorithm looking at each higher level capsule j and then examines each input and updates the corresponding weight b_{ij} according to the formula. The formula says that the new weight value equals to the old value plus the dot product of current output of capsule j and the input to this capsule from a lower level capsule i. The dot product looks at similarity between input to the capsule and output from the capsule. After this step, the algorithm starts over from step 3 and repeats the process r times. After r times, all outputs for higher level capsules were calculated and routing weights have been established. The forward pass can continue to the next level of network. Then, after the classification of an object k, we can calculate the *margin loss* as a variation of the Squared Hinge Loss:

$$L_k = \begin{cases} \max\{0, m^+ - \|\vec{v}_k\|\}^2 & \text{if there is } k \\ \lambda \max\{0, \|\vec{v}_k\| - m^-\}^2 & \text{if there isn't any } k \end{cases} \tag{1.29}$$

where $m^+ = 0.9$, $m^- = 0.1$ and $\lambda = 0.5$.

A Capsule Network has the following architecture:

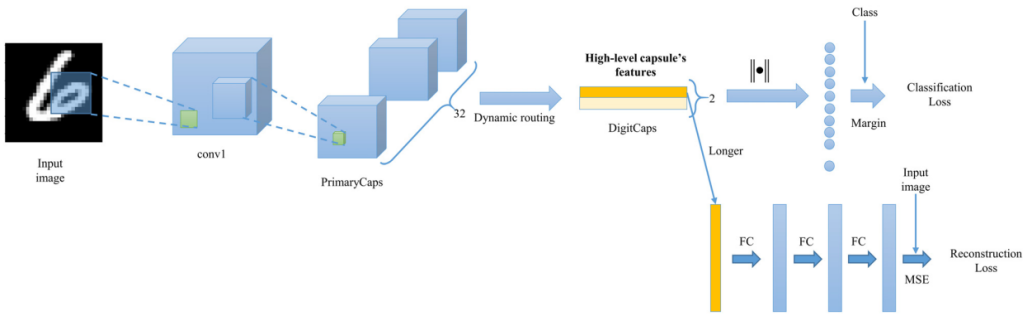


Figure 1.18: Representation of a CapsNet architecture. From DigitCaps, eventually, there is the possibility to have a decoder of fully connected layers with the aim of reconstruct the image [15].

where the output of the standard convolutional layer it's the input of the primary capsule layer. The convolutional layer converts pixel intensities to the activities of local feature detectors that are then used as inputs to the primary capsules The aim of this layer is to transform the 1D data representation in a multidimensional representation, from scalar to vector. So the entire volume of the output after convolution it's divided in a certain number of capsules, one for each entity to be recognize and represent.

The Primary Capsules are the lowest level of multi- dimensional entities and, from an inverse graphics perspective, activating the primary capsules corresponds to inverting the rendering process. This is a very different type of computation than piecing instantiated parts together to make familiar wholes, which is what capsules are designed to be good at.

Then there is a Convolutional Capsule layer: each primary capsule contains convolutional units and its outputs sees the outputs of all of them. Conv1 units whose receptive fields overlap with the location of the center of the capsule. Each output is a vector with the same dimension of the capsule and each capsule in the grid is sharing their weights with each other. One can see PrimaryCaps as a Convolution layer with its block non-linearity given by the squash operation.

The final layer called DigitCaps has one capsule per digit class and each of these capsules receives input from all capsules in layer below.

Matrix implementation and EM algorithm

In 2018 Hinton and the other researches presented in [12] a refinement of the architecture. This version of capsules consists in capsules with logistic units to represent the presence of an entity and a 4×4 matrices which could learn to represent the relationship between that entity and the viewer, the pose.

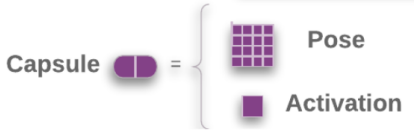


Figure 1.19: A representation of capsule in [12].

The new architecture uses capsules whose inputs and outputs are both a matrix M and an activation probability a, rather than vectors, and the previous dynamic *routing-by-agreement* algorithm is exchanged with a form of expectation–maximization algorithm called *EM routing*. The objective of this routing is to group capsules to form a part-whole relationship using the EM clustering technique usually used to cluster data points into Gaussian distributions.

A capsule in one layer votes for the pose matrix of many different capsules in the layer above by multiplying its own pose matrix by trainable viewpoint-invariant transformation matrices that could learn to represent part-whole relationships. The reasoning for exchanging the vectors to matrices is in order to make the transformation matrices in between capsules smaller.

Each of the votes is weighted by an assignment coefficient. These coefficients are iteratively updated for each image using the EM algorithm between each pair of adjacent capsule layers, such that the output of each capsule is routed to a capsule in the layer above that receives a cluster of similar votes.

By using matrices for an output of size n, the transformation matrices can be made with n elements instead of n^2. The probability of the entity represented by a capsule being present is no longer the length of its vector but a separate parameter a. This in order to avoid the squashing function which was not considered objective and sensible. In the dynamic routing of [11] the agreement was measured as the cosine of the angle between two pose vectors. Although this is an implementation that works, it is not good at distinguishing between a quite good agreement and a very good one.

In order to correct this the EM routing algorithm was introduced, described below. When the capsules i in a lower layer l has calculated their output matrices M_i and activation probabilities a_i these are used to cast a vote on the pose of each capsule j in the layer $l + 1$ above. Each capsule i in layer l has a weight matrix W_{ij} to each capsule j in layer $l + 1$ that is iteratively learned during the training of the network. The output matrices M_i are multiplied with the corresponding weight matrix W_{ij} in

order to retrieve vote. A vote v_{ij} for the parent capsule j from capsule i is computed by multiplying the pose matrix of capsule i with a viewpoint invariant transformation matrix:

$$v_{ij} = M_i W_{ij} \tag{1.30}$$

The probability that a capsule i is grouped into capsule j as a part-whole relationship is based on the proximity of the vote v_{ij} to all the other votes $(v_{1j}, \ldots, v_{kj})$ from other capsules. The weight matrix W_{ij} is learned through a cost function and the backpropagation. It learns not only what an object is composed of, it also makes sure the pose information of the parent capsule matched with its sub-components after some transformation.

Even the viewpoint may change, the pose matrices and the votes change in a co-ordinate way: EM routing is based on proximity and therefore it can still cluster the same children capsules together even if they are transformed. Hence, the transformation matrices are the same for any viewpoints of the objects, so they are viewpoint invariant. We just need one set of the transformation matrices and one parent capsule for different object orientations to classify all the capsules.

EM routing clusters capsules to form a higher level capsule in runtime. It also calculates the assignment probabilities r_{ij} to quantify the runtime connection between a capsule and its parents, that is related with the activation of the capsule.

In EM routing, we model the pose matrix of the parent capsule with a Gaussian so each component of the 4×4 matrix represent both a the mean (μ) and the standard deviation (σ) of the distribution.

The probability that the $h-$th component of the vote v_{ij} belongs to the capsule j's Gaussian model can be calculated as follow:

$$P_{i|j}^{h} = \frac{1}{2\pi(\sigma_j^h)^2} exp\left(-\frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right) \tag{1.31}$$

We now want to decide which capsules to activate in the layer above and how to assign each active lower-level capsule to one active higher-level capsule. Each capsule in the higher-layer corresponds to a Gaussian and

the pose of each active capsule in the lower-layer corresponds to a data-point. Using the minimum description length principle we have a choice when deciding whether or not to activate a higher-level capsule. If we do activate the higher-level capsule we must pay a fixed cost for coding its mean and variance and the fact that it is active and then pay additional costs, pro-rated by the assignment probabilities, for describing the discrepancies between the lower-level means and the values predicted for them when the mean of the higher-level capsule is used to predict them via the inverse of the transformation matrix. We can compute the cost of describing a datapoint is to use the negative log probability density of that datapoint's vote under the Gaussian distribution (1.31), as an approximation. The incremental cost of explaining a whole data-point i by using an active capsule j that has an axis-aligned covariance matrix is simply the sum over all dimensions of the cost of explaining each dimension, h, of the vote v_{ij}:

$$cost_{ij}^h = -ln(P_{i|j}^h) \tag{1.32}$$

$$= -ln\left(\frac{1}{2\pi(\sigma_j^h)^2} exp\left(-\frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right)\right) \tag{1.33}$$

$$= \frac{ln(2\pi)}{2} + ln(\sigma_j^h) + \frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2} \tag{1.34}$$

Defining $\sum_i r_{ij}$ the amount of data assigned to j and summing over all lower-level capsules for a single dimension, h, of j we get:

$$cost_j^h = \sum_i r_{ij}\left(\frac{ln(2\pi)}{2} + ln(\sigma_j^h) + \frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right) \tag{1.35}$$

$$= \left(\frac{ln(2\pi)}{2} + ln(\sigma_j^h)\right)\sum_i r_{ij} + \sum_i r_{ij}\frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2} \tag{1.36}$$

The cost has to consider both the situation in which we do not activate a capsule that should be active and, vice versa situation for false positive activation. So two fixed costs have to be defined:

- β_u for describing the poses of all the lower-level capsules that are as-

signed to the higher-level capsule, that is the negative log probability density of the data-point under an improper uniform prior;

- β_a for describing the discrepancies between the lower-level means and the values predicted for them when the mean of the higher-level capsule is used to predict them via the inverse of the transformation matrix.

The difference in cost between these two, is then put through the logistic function on each iteration to determine the higher-level capsule's activation probability. So the activation function of capsule j in then defined as the logistic function of the total cost:

$$a_j = logistic\left(\lambda\left(\beta_a - \beta_u \sum_i r_{ij} - \sum_h cost_j^h\right)\right) \tag{1.37}$$

where β_a is the same for all capsules and λ is an inverse temperature parameter. We learn β_a and β_u in training using backpropagation and set fixed λ as a hyper-parameter, as inverse temperature parameter.

Routing algorithm returns activation and pose of the capsules in layer $l+1$ given the activation and votes of capsules in layer l. The EM method fits datapoints into a a mixture of Gaussian models with alternative calls between an E-STEP and a M-STEP.

The M-STEP re-calculate the Gaussian models' values and parent activation a_j from a, v and r_{ij}. The E-STEP determines the assignment probability r_{ij} of each datapoint to a parent capsule, based on the new Gaussian model and the new a_j. At the end of the last iteration the last a_j will be the parent capsule's output. The EM procedure is described in the Table 1.2.

EM Routing

1. **procedure** EM Routing (a, v)
2. for all capsule *i* in layer *l* and capsule *j* in layer $(l+1)$: $r_{ij} = 1/|N_{l+1}|$
3. for *r* iterations do
4. for all capsule *j* in layer *l+1*: M-STEP (a, r, v, j)
5. for all capsule *i* in layer *l*: E-STEP (μ, σ, a, v, i)

return a, M

1. **procedure** M-STEP (a, r, v, j)
2. for all capsule *i* in layer *l*: $r_{ij} = r_{ij} * a_i$
3. for all h: $\mu_j^h = \frac{\sum_i r_{ij} v_{ij}^h}{r_{ij}}$
4. for all h: $(\sigma_j^h)^2 = \frac{\sum_i r_{ij} (v_{ij}^h - \mu_j^h)^2}{r_{ij}}$
5. $cost^h = \left(\beta_u + log(\sigma_j^h)\right) \sum_i r_{ij}$
6. $a_j = logistic\left(\lambda\left(\beta_a \sum_h cost^h\right)\right)$

1. **procedure** E-STEP (μ, σ, a, v, i)
2. for all capsule j in layer $(l+1)$: $P_j = \frac{1}{\sqrt{\prod_h^H 2\pi(\sigma_j^h)^2}} exp\left(-\sum_h^H \frac{(v_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right)$
3. for all capsule j in layer $l+1$: $r_{ij} = \frac{a_j P_j}{\sum_{k \in N_{l+1}} a_k P_k}$

Table 1.2: EM algorithm procedure as described in [12] .

The loss proposed to learn the parameters during backpropagation is the *margin loss* as in (1.29). In the article [12] it is called "spread loss " and

is defined as follow:

$$L_i = \max\{0, m - (a_t - a_i)\}^2 \tag{1.38}$$

where m is the margin, while a_t is the activation of the target class and a_i is the activation of the wrong class. This loss maximize the gap between the activation of the target class and the activation of other classes.

It's supposed to start with a small margin, $m = 0.2$ and linearly increasing it during training to $m = 0.9$. This approach avoids dead capsules in the earlier layers.

Finally the total loss is calculated as the summation of all the contributions of wrong activation: $\sum_{i \neq t} L_i$.

The architecture for this model is represented in Figure 1.20:

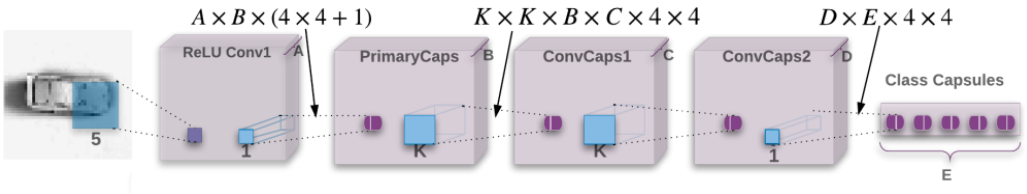


Figure 1.20: Architecture for matrix implementation of CapsNet as proposed in [12].

The model starts with a convolutional layer with with a ReLU nonlinearity. All the other layers are capsule layers starting with the primary capsule layer.

The 4×4 pose of each of the primary capsule types is a learned linear transformation of the output of all the lower-layer ReLUs centered at that location.

The activation of the primary capsules are produced by applying the sigmoid function to the weighted sums of the same set of lower-layer ReLUs.

Primary capsules are followed by two convolutional capsule layers of which the last one is connected to the final capsule layer that has one capsule per output class. The transformation matrices are then shared be-

tween different positions of the same capsule type and the scaled coordinate (row, column) of the center of the receptive field of each capsule are added to the first two elements of the right-hand column of its vote matrix, in a technique called *Coordinate Addition*.

The routing procedure is used between each adjacent pair of capsule layers. For convolutional capsules, each capsule in layer $l + 1$ sends feedback only to capsules within its receptive field in layer l. The instances closer to the border of the image receive fewer feedback with corner ones receiving only one feedback per capsule type in layer $l + 1$.

This new type of capsule system proposed in [12] introduces a logistic unit for each capsule, to represent the presence of an entity, and a pose matrix, to represent the pose of that entity. It introduced also a new iterative routing procedure between capsule layers, based on the EM algorithm, which allows the output of each lower-level capsule to be routed to a capsule in the layer above in such a way that active capsules receive a cluster of similar pose votes.

Matrix implementation with the EM algorithm achieves better accuracy than the state-of-the-art CNN, reducing significantly the number of errors.

CHAPTER 2

MACHINE LEARNING FOR BRAIN IMAGING

"What I cannot create, I do not understand."
R. Feynman

Computer scientists use biological models as inspiration to improve deep-learning methods, so these two fields of research are very close to each other. In fact, on the other side, computer performances in analysis and recognition are applied on brain images to unlock deeper biological insights into the functionality of our brains.

So one of the most studied and fascinating applications of deep learning techniques is in brain imaging. Theoretical neuroscientists are working to develop a multi-scale theory of the brain that synthesizes top-down and data-driven bottom-up approaches. In particular cognitive neuroscientists are looking at the nature of visual perception.

The crucial question is: how does the brain create a representation of an object from multi-sensory information? Much work in the last decades focused on object recognition as a framing problem for the study of high-level visual cortex [16]. A deeper problem is that object recognition and categorization are only a small slice of our visual systems can and must

do. Computer vision field is increasingly moving onto study 'scene understanding': computer science and visual neuroscience holds the promise to advance the state of understanding in both fields.

In this framework of research the best approach to brain imaging concerns segmentation methods. So here we make a review of the state-of-the-art neural networks used for segmentation and then we try to explain how a CapsNets approach could be possible and useful for brain imaging, studying both the construction of the architecture and the problem of brain images in all the steps from the data set acquisition to the analysis of it.

2.1 CNN for object segmentation

Object segmentation in computer vision communities has remained an interesting and challenging problem over the past several decades. The communities came to favor supervised techniques, instead of unsupervised (MRF and CRF [17]), where algorithms were developed using training data to teach systems the optimal decision boundaries in a constructed high-dimensional feature space. In computer vision fields, various sets of feature extractors were used to construct these spaces.

Deep learning techniques are used to achieve object recognition and achieve a lot of success in classification problems. Usually CNNs were applied to address computer vision and researchers tries to apply them to structured prediction problems like semantic segmentation.

There are different levels of "scene understanding" that we can reproduce with machine learning algorithm [18]: image classification, object localization, semantic segmentation and intance segmentation.

So the very first step is to classify the object then to provide additional information about its spatial location. Semantic segmentation is the next natural step in the progression from coarse to fine inference, making dense prediction for every pixel. Further improvements is instance segmentation which separates labels for different instances of the same class.

The typical use of convolutional networks is on classification tasks, where the output to an image is a single class label. However, in many

visual tasks, especially in biomedical image processing, the desired output should include localization and usually a class label is supposed to be assigned to each pixel.

The problem of label each pixel can be modeled as the problem to assign a state from the label space $\mathcal{L} = \{l_1, l_2, \ldots, l_k\}$ to each one of the elements of a set of random variables $\mathcal{X} = \{x_1, x_2, \ldots, x_N\}$.

Deep learning techniques, CNNs in particular, can be used for pixel-level labeling problems like semantic segmentation thanks to their ability to learn feature representations in an end-to-end fashion in stead of using features that require domain expertise. The artificial neural networks that are used to recognize shapes typically use one or more layers of learned feature detectors that produce scalar outputs. The computer vision community uses complicated, hand-engineered features that produce a whole vector of outputs including an explicit representation of the *pose*[1] of the feature.

Autoencoder neural network is an unsupervised learning algorithm useful for shape recognition and object segmentation. Suppose we have only unlabeled training example, we set $\{x^{(1)}, \ldots, x^{(N)}\}$ where $x^{(i)} \in \mathbb{R}^{\kappa}$, the target values have to be equal to the inputs: $y^{(i)} = x^{(i)}$.

The autoencoder is made up of an encoder network and a decoder network. The first takes the input and outputs a feature map (vector or tensor) that hold the information that represent the input. The other takes feature vector from the encoder and gives the best closest match to the actual input.

This architecture tries to learn a function that approximates the identity function [19]. Each hidden unit i computes a function of the input that represents the activation of the neuron. So one can computes the maximum of this function finding the input that maximize the activation. The visualization of the output image helps to understand what feature hidden unit i is looking for.

By examine all the images, one for each hidden unit, we can understand what the ensemble of hidden units is learning. So the autoencoder

[1] In computer vision it refers to the combination of position and orientation of an object with the term *pose*.

learning algorithm is an approach to automatically learn features from unlabeled data.

For semantic segmentation tasks Deep Convolutional Neural Networks methods mainly utilize the architecture of Fully Convolutional Networks (FCN). The trend is to convert CNN architecture constructed for classification to a Fully Convolutional Network. This classification network with downsampling operations sacrifices the spatial resolution of feature maps to obtain the invariance to image transformations, so the results are coarse.

Many approaches have been proposed to solve the above problems. For example some researchers applied dilated receptive fields and capture larger contextual information without losing resolution. Others explore multi-scale or global features for performance improvement. Another approach is to recover the spatial resolution by an upsampling or deconvolutional path. For example SegNet and U-net generate high-resolution feature maps for dense prediction. In these upsampling solutions, the deconvolutional and unpooling layers are appended with symmetric structure of the corresponding convolutional and pooling layers.

A most recent alternative technique is the DeconvNet, which implements the deconvolutional idea also in learning process.

Newer deep architectures particularly designed for segmentation have advanced the state-of-the-art by learning to decode or map low resolution image representations to pixel-wise predictions. These segmentation architectures usually share the same encoder network and they only vary in the form of their decoder network.

This encoder network weights are typically pre-trained on a large object classification data set.

The decoder network varies between these architectures and is the part which is responsible for producing multi-dimensional features for each pixel for classification. Each decoder in the Fully Convolutional Network architecture learns to *upsample* its input feature map(s) and combines them with the corresponding encoder feature map to produce the input to the next decoder.

In the following sections we go through all the relevant methods used for segmentation, both CNN based and not.

2.1.1 Fully Convolutional Networks

Fully Convolutional Network (FCN), can transform a classification-purpose CNN to produce spatial heatmaps by replacing fully connected layers with convolutional ones (Figure 2.1).

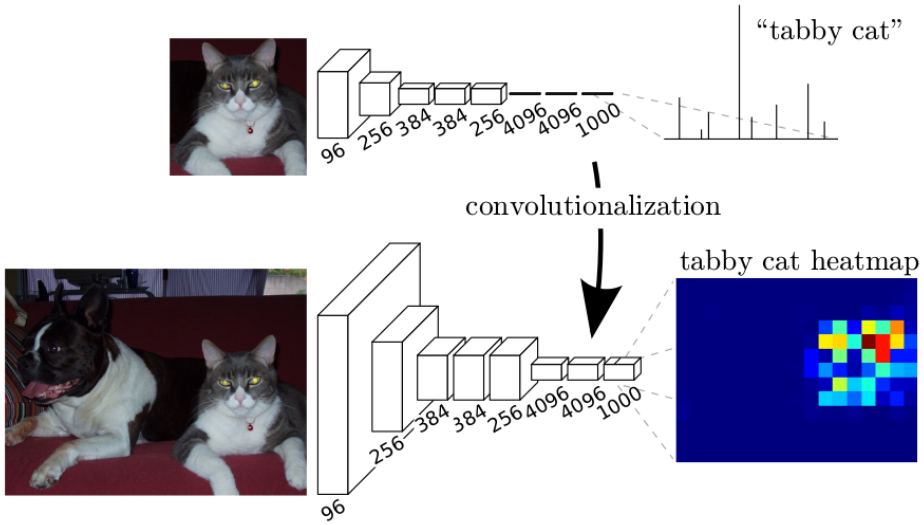


Figure 2.1: Transforming fully connected layers into convolution layers enables a classification net to output a heatmap. Adding layers and a spatial loss produces an efficient machine for end-to-end dense learning [20].

Every classification networks could be adapted into FCN to transfer their learned representations by fine-tuning to the segmentation task [20], achieving state-of-the-art performances. While a general deep network computes a general nonlinear function, a net with only convolutional layers computes a nonlinear filter operating on an input of any size and producing an output of corresponding re-sampled spatial dimension.

FCN models hinder their application to certain problems and situations, for example their spatial invariance does not take into account useful global context information and they are not completely suited for unstructured data such as 3D point clouds.

To achieve the task of semantic segmentation the classifier is usually followed by a blob detection [21]. The blob detection is a procedure used

to answer at the following questions: how should regions be selected automatically? And how to detect appropriate scales and regions at interest when there is no a *priori* information available? How to determine the scale of an object and where to search for it before knowing what kind of object we are studying and before knowing where it is located?

This problem is intractable as a pure mathematical problem. The basic tools to address this problem will be scale-space theory and a heuristic principle stating that stable in space blob-like regions could correspond to significant structure in the image. Blobs are regions that are brighter or darker than the background and stand out from their surrounding. In other words it's a region associated with one local extreme.

The goal is to extract significant image features considering the appearance and stability of these objects over scales. So it's important to the define the spatial extent of the region around the blob, in [21] they proposed to extent it until it would merge with another blob.

FCN approach gives a coarse label map, performing a simple deconvolution implemented as bilinear interpolation. If on one hand this architecture accepts a whole image as an input and performs fast and accurate inference, on the other hand works with fixed-size receptive field. So label prediction is done with only local information for large objects while small ones are often ignored.

For these reasons a lot of variants are done to try to optimize FCN and solve its problems.

2.1.2 U-Net

U-Net is a variation of the FCN described above. It consists on a contracting path and an expansive path, showed in Figure 2.2.

The first follows the typical architecture of a convolutional network while the last consists in halving the number of feature map with upsampling and upconvolution.

The network has 23 convolutional layers and the architecture looks u-shaped thanks thanks to the symmetry between the two paths. U-net was

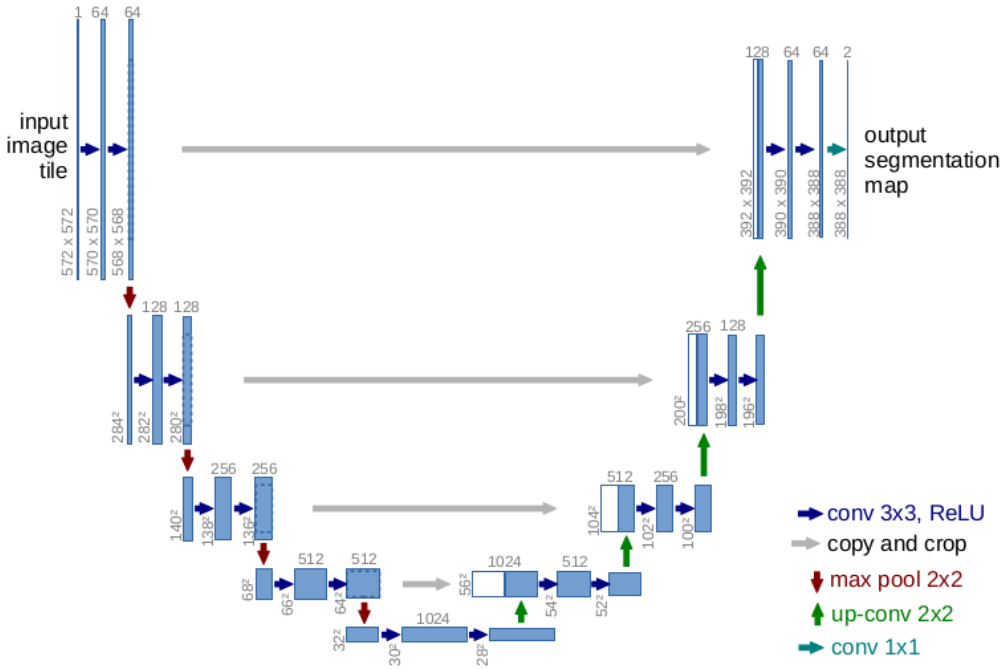


Figure 2.2: U-net architecture (example for 32 × 32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.[22]

applied to three different segmentation tasks in [22] for neuronal structured and other cells segmentation. They found that this approach achieves very good performance especially training in small data set with data augmentation.

The most important characteristic of this architecture is that it doesn't have any fully convolutional layers and and only uses the valid part of each convolutional, so with no padding. The overlap-tile strategy is used to predict pixels in the border region, mirroring the input image in missing context (Figure 2.3).

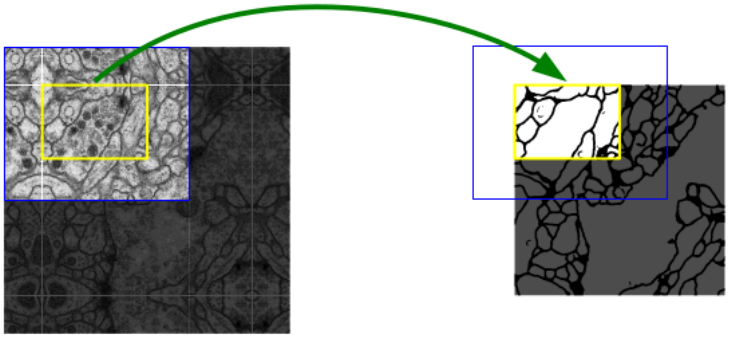


Figure 2.3: Overlap-tile strategy in segmentation of neuronal structures. Prediction of the segmentation in the yellow area, requires image data within the blue area as input and missing input data is extrapolated by mirroring.[22]

2.1.3 SegNet

SegNet is a deep FCN for semantic pixel-wise segmentation [23]. It has the structure of an autoencoder but it's used for supervised learning tasks and the decoders are integral part of the network (Figure 2.4).

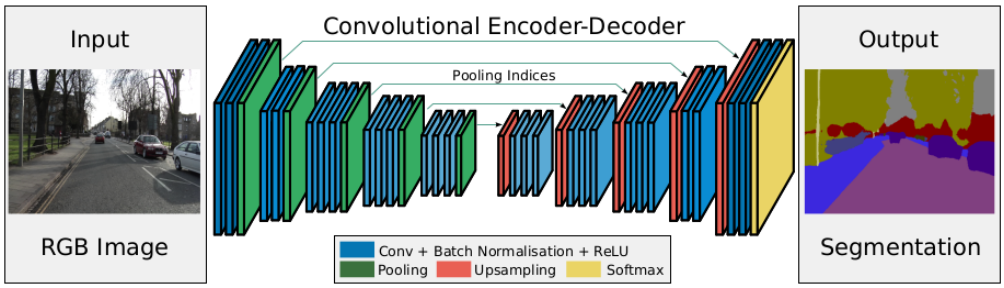


Figure 2.4: SegNet architecture [23].

The encoder network consists of 13 convolutional layers, each of which has a corresponding decoder layer. Each encoder performs convolution with element-wise ReLU and 2×2 max-pooling indices with strides 2. The resulting output it's sub-sampled after the capture and store of boundary information.

A more efficient way to store these it to consider only maximum indices, for example the location of the maximum feature value in each pool-

ing spatial window is memorized for each encoder feature map. These indices are used from each decoder to upsample the input feature map.

So the role of the decoder network is to map the low resolution encoder feature maps to full input resolution feature maps for pixel-wise classification, using pooling indices computed in the max-poooling step of the corresponding encoder to perform non-linear upsampling.

The final decoder output is fed to a multi-class soft-max classifier to produce class probabilities for each pixel independently.

The output of the soft-max classifier is a K channel image of probabilities where K is the number of classes. The predicted segmentation corresponds to the class with maximum probability at each pixel.

The SegNet architecture was introduced to design an efficient method for road and indoor scene understanding. It only stores max-pooling indices of the feature maps and uses them in its decoder network to achieve good performance so it results efficient both in terms of memory and computational time.

SegNet is smaller and faster than other competing architectures and achieves the segmentation task with great results.

2.1.4 DeconvNet

A critical limitation of FCN approach is that smooths detailed structures of an object, because the label map is too coarse and deconvolution is over simple. Semantic segmentation involves deconvolution conceptually but learning deconvolution network is not very common.

A different strategy was proposed in [8] where they learn a multi-layers deconvolution network composed of deconvolution, unpooling and ReLU layers.

The architecture is composed of two parts: convolution and deconvolution (Figure 2.5).

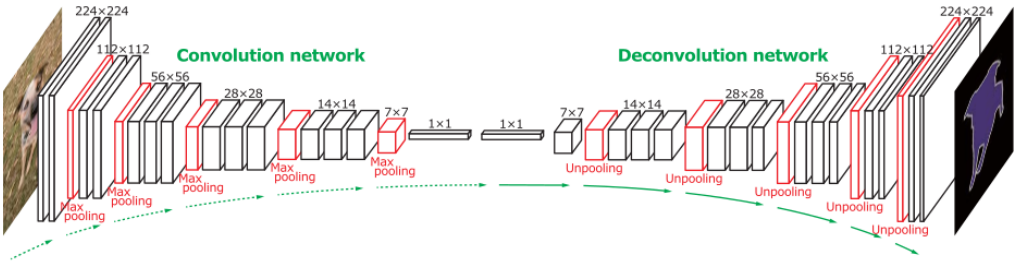


Figure 2.5: Overall architecture of the DeconvNet. Given a feature representation obtained from the convolution network, dense pixel-wise class prediction map is constructed through multiple series of unpooling, deconvolution and rectification operations. [8]

The first corresponds to feature extractor, it transforms the input image to multidimensional feature representation. The other is a shape generator, it produces object segmentation from the feature extracted to convolutional part. The output is a probability map in the same size of input image. It indicates probability of each pixel that belongs to one of the predefined classes and could be visualize thanks to an heat-map. The deconvolutional part is a mirror of the convolutional part and they have an opposite scope: the first part has to reduces the size of activation while the second has to enlarges the activation. With common pooling the network retains only robust activation so spatial information within receptive field is lost. Otherwise unpooling reconstruct the original size of activations. It records the locations of maximum activations selected during pooling operation in switch variables.

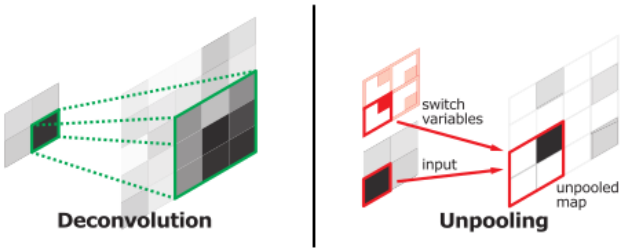


Figure 2.6: Illustration of deconvolution and unpooling operations. [8]

A hierarchical structure of deconvolutional layer are used to capture different level of shape details: filters in lower layers tend to capture overall shape while filters in higher layers encodes class-specific fine-details. So coarse-to-fine object structures are reconstructed through the propagation in the deconvolutional layers. By the combination of unpooling and deconvolution, the network generates accurate segmentation maps. Researchers in [8] proposed to combine FCN with DeconvNet using both the outputs and computing the mean of them and finally applied CRF (Conditional Random Fields [17]) to obtain the final semantic segmentation. The performance of all the deconvolutional solutions described till now are limited by their difficulties on model optimization or simple network design.

Stacked Deconvolutional Network (SDN) is a deeper deconvolutional network easier to optimize compared with most of the previous solutions. In [24] they proposed a shallow deconvolutional network called SDN unit and then they stacked multiple SDN units one by one with dense connections. Each SDN unit is an encoder-decoder network, the first operated as a downsampling process and the second as an upsampling process. The first encoder's part employs full convolutional DenseNet-161 to obtain high-semantic features, while the others are implemented as downsampling block. In particular they consists of a max-pooling layer, 2 convolutional layers and a compression layer. They used 2 downsampling block to enlarge the receptive fields of the network. In the decoder module they apply upsampling blocks to upsample feature maps to larger resolution. They consist of a deconvolutional layer, several convolutional layers and a compression layer. So for each SDN unit we have a classification network to encode images and deconvolutional layers to generate more refined recovery of the spatial resolution. These SDN units are piled up from end to end. Intra-unit connections are performed between convolutional layers directly linking the inputs of previous convolutional layers to the ones of back convolutional layers. Inter-unit connections are performed between certain two SDN units in two different ways: linking encoder to decoder of the adjacent SDN unit, connecting the multi-scale feature map from the encoder of the first SDN unit to the decoder modules of each SDN unit.

This techniques of stack multiple shallow deconvolutional networks with random initialization leads to additional optimization difficulty but hierarchical supervision is applied during the upsampling process, it ensures that early layers of the network can obtain more gradient feedback.

2.1.5 Previous approaches to segmentation topic

In 1992 a method for the creation of Machine Learning came into the limelight: Support Vector Machine. It was born as a direct implementation of a learning theory based on Statistical Learning Theory (SLT) from researchers that have been looking for a new principle capable of overcoming a few drawbacks that the moment theory involved. With this theory they proposed a new learning principle which could lead machines throughout their learning process: the Structural Risk Minimization (SRM), by which the machine is forced not only to try to learn an experience at disposal, the Empirical Risk Minimization (ERM), but it must be able to generalize.

From that moment on SVM gained an enormous popularity performing supervised learning task in both classification and segmentation. The roots of this approach is the Support Vectors method of constructing the optimal separating hyperplane introducing decision rules used to binary classification.

In [25] Campanini et al. have proposed a variant for a SVM approach. This new method is based on the detection of the region of interest without the extraction of any feature by exploiting all the information available on the image with a single pixel approach.

They used this method for digital mammograms, where it's difficult to identify morphological, directional or structural quantities that can characterize the lesions at any scales and any modalities of occurrence because the visual manifestation in the mammogram of the shape and edge of a lesion depends upon the physical properties of the lesion, the image acquisition technique and the projection considered. Clinical trials and retrospective studies indicate that the detection rate can be increased with Computer Aided Detection (CAD) systems, without any significant de-

crease of specificity. The automatic detection of masses can be made difficult by the wide diversity of their shape, size and subtlety.

Detection methods often rely on a feature extraction step: here, the masses are isolated by means of a set of characteristics which describe the opacities. Due to the great variety of the masses, it is extremely difficult to get a common set of features effective for every kind of masses. This is why they proposed a mass detection system which does not rely on any feature extraction step.

The algorithm automatically learns to detect the masses by the examples presented to it, with the same idea developed with Representative Learning. In this way, there is no a priori knowledge provided by the trainer: the only thing the system needs is a set of positive examples (masses) and a set of negative examples (non-masses). Then they considered mass detection as a two-class pattern recognition problem and the great amount of information handled by the algorithm is classified by means of a Support Vector Machine classifier.

The advantages of SVM over other classifiers are that its setting is easier, it usually performed better on novel data and it was able to compress the useful information of high-dimensional spaces into a small number of elements named support vectors. SVMs are therefore capable of learning in sparse, high-dimensional spaces, by using very few training examples.

The algorithm encodes all the regions of the image in the form of vectors, these vectors being then classified as suspect or not by means of an SVM classifier. The system is virtually able to detect lesions whatever position these may occupy and at different scales in the input mammographic image; this is realized by scanning and classifying all the possible locations of the image with the passage of a window called *crop*. By combining the scanning pass with an iterated resizing of the window, multi-scale detection is so achieved. Each crop classified as positive identifies an area judged as suspect by the CAD system.

The solution implemented is that of using scanning masks of different dimensions and subsampling the crops of the image extracted from that mask to a prefixed size of pixels, this is why we can call it a *single pixel approach*.

For each crop, the vector of coefficients is used as input for the **first SVM classifier**. Once trained, the SVM classifies each crop. For each crop, SVM gives the distance from the separating hyperplane for positive (suspect) regions. This distance is an index of confidence on the correctness of the classification: a vector classified as positive with a large distance form the hyperplane will have a higher likelihood of being a true positive as compared to a vector very close to the hyperplane, and hence close to the boundary area between the edges of the two classes.So there is a list of suspect candidates, where each candidate consisting of a crop with a distance from the hyperplane greater than a prefixed threshold.

All the candidates are then passed to a **second SVM classifier** to eliminate the false candidates selected by the first classifier, usually survived because of theirs high distance from the hyperplane. The task of the two classifiers are quite different. The first SVM must have a very small error, the second SVM could have a worse error, compared to the first one.

The last step of the detection scheme consists of the merging of the multi-scale information. The output of the second SVM classifier is a set of candidates detected at either one of the scales.

The scanning step at one particular scale is diverse from the others. They fuse all the candidates within a specified neighborhood into a single candidate. Therefore, the output of the detection method (called *expert*) is a list of suspect regions, each one detected at least at one scale.

An ensemble of experts improves the overall performance of individual experts, if the individual experts are independent, or negatively dependent. Each expert differs from the others for the training sets and/or for the kernel used in the SVM classifiers.

A region is considered suspect only if at least two (of three) experts detect that region.

The idea is to provide the classifier with a complete representation of the image, without guiding the generalization of the class with assumptions deriving from our modeling of the pattern. To this aim, in [25] they used an over-complete dictionary of Haar wavelets [2]. A redundant encod-

[2]The Haar wavelet is a sequence of rescaled "square-shaped" functions which together form a wavelet family or basis. Wavelet analysis is similar to Fourier analysis in that it

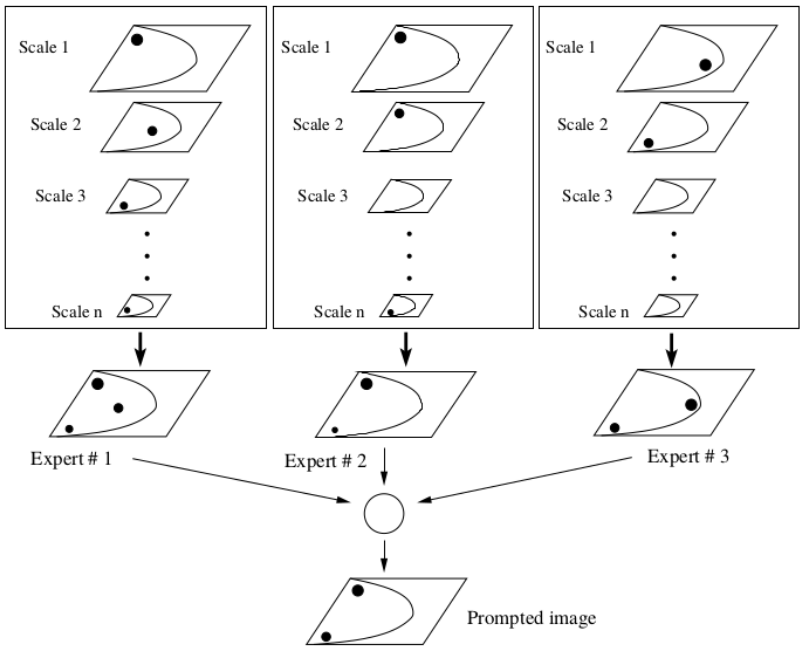


Figure 2.7: Three experts: the prompted image consists of any overlapped suspect regions "voted" by at least two of three experts. Each expert corresponds to a detection system with the merging of multi-scale information.[25]

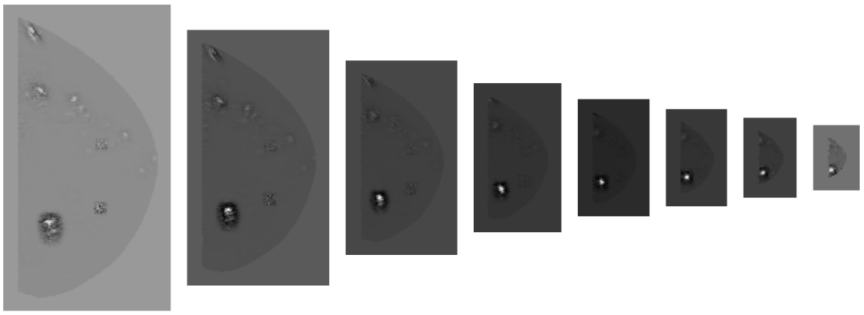
ing of the data with spatially superposed scale has a greater number of coefficients. The wavelet tranform is calculated for each of the crops produced by scanning at the various scales.For each level of decomposition, three types of coefficients are obtained, namely horizontal, vertical, and diagonal.

If no restriction is placed on the class of functions when choosing the estimate the rule function with SVMs, it could happen that even a function that performs well with training data may not generalize well to unseen examples. The minimization of the training error it is necessary to restrict the class of functions. Restricting the complexity of the chosen function class means avoid the overfitting problem.

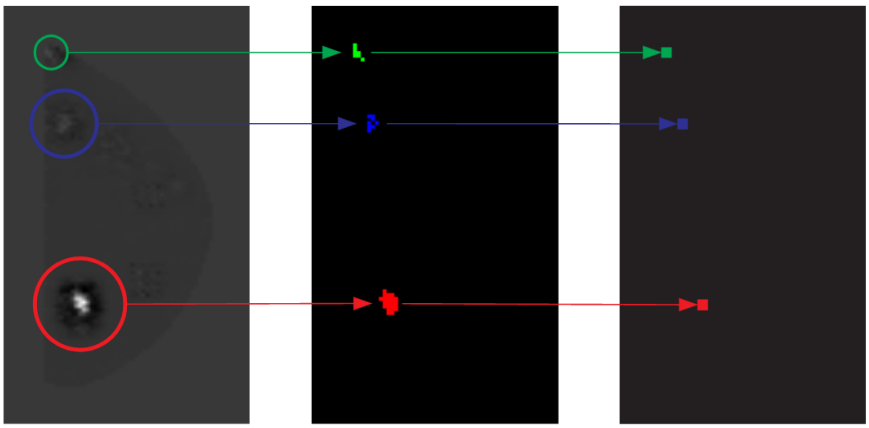
The Maximal Margin Hyperplane (MMH) is finally computed as a decision surface.

allows a target function over an interval to be represented in terms of an orthonormal basis.

Note that the shifting procedure of the detection step is very similar to the process of subsampling an image with a convolution mask. As a result of the convolution, we obtain a new image of reduced dimension where the pixel value is set as the output of SVM for the given mask. In this view, SVM behaves like a filter which transforms the original image in a likelihood image according to a prefixed model. In this new representation pixels with high value correspond to an area of the mammogram with high probability to contain a mass. In Figure 2.7 they shows these likelihood images for different scale of search.



(a) The likelihood images at multiple scan levels.



(b) Peak identification at one level scan. Likelihood image (left), identified blobs (center) andpeak (strong) elements (right).

Figure 2.7: Results from [26].

2.2 CapsNet for object segmentation

In the last few years, deep learning methods, in particular convolutional neural networks, have become the state-of-the-art for various image analysis tasks. Specifically related to the object segmentation problem, U-Net, Fully Convolutional Networks and other encoder-decoder style CNNs have become the desired models for various medical image segmentation tasks and constitute a popular class of solutions for segmentation, producing state-of-the-art results in a variety of applications. They are commonly constructed with an encoder-decoder architecture and their success depends on finding an architecture to fit the task. So researchers work on designing new and more complex deep networks to improve the expected outcome. This naturally brings high number of hyperparameters to be configured, making the overall network too complex to be optimized.

Moreover convolutional neural networks have shown remarkable results over the last several years but do come with their own set of flaws. Originated from and constructed upon convolutional neural networks, FCNs' encoders inherit some common drawbacks of CNNs, one of which is the lack of an internal mechanism in achieving viewpoint-invariant recognition. As a result, more data samples or additional network setups would be required for objects from different viewpoints to be correctly recognized. The absence of explicit part-whole relationships among objects imposes another limitation for FCNs – without such a mechanism, the rich semantic information residing in the higher layers and the precise boundary information in the lower layers can only be integrated in an implicit manner.

Otherwise the new architecture of Capsule Networks, shown great initial results and formed meaningful part-to-whole relationships not found in standard CNNs that could be very useful for segmentation tasks. Such part-whole hierarchy equips capsule nets with a solid foundation for viewpoint-invariant recognition, which can be implemented through dynamic routing or EM routing. The same hierarchy, if properly embedded into a segmentation network, would provide a well-grounded platform to specify contextual constraints and enforce label consistency.

The task of segmenting objects from images can be formulated as a joint object recognition and delineation problem. The goal in object recognition is to locate an object's presence in an image, whereas delineation attempts to draw the object's spatial extent and composition so apart from recognizing the object, we also have to label that object at the pixel level, which is an ill-posed problem.

Recent studies have hypothesized that capsules can be used effectively for object segmentation with high accuracy and heightened efficiency compared to the state-of-the-art segmentation methods.

The simple three-layer capsule network showed remarkable initial results producing state-of-the-art classification results on the MNIST dataset and since then, researchers have begun extending the idea of capsule networks to other applications. No work existed in literature for a method of capsule-based object segmentation since last year, because performing object segmentation with a capsule-based network is difficult for a number of reasons: is extremely computationally expensive, both in terms of memory and run-time and the number of parameters required quickly swells beyond control.

2.2.1 SegCaps

In [27] LaLonde et al. studied the possibility to use Capsule Networks for object segmentation. This work is the first in literature where a convolutional-deconvolutional capsule network is proposed. They used their so called SegCaps to segment pathological lungs from low dose CT scans and they compared the results with other U-net based architecture.

They solved memory burden and parameter explosion given by the original CapsNet architecture extending the idea of convolutional capsules and rewriting the dynamic routing algorithm. Changes from the original article are made in order to adapt CapsNet to segmentation tasks. The original dynamic routing tales place between every parent and every possible child, the modification proposed is to route children capsule only to parents within a defined spatially-local kernel. The other modification is to not shared the transformation matrices with all the capsules but only

within capsules of the same type. Considering these changes the algorithm proposed take the name of "locally-constraint dynamic routing".

Besides these novelties the SegCaps architecture has also "deconvolutional" capsule, to achieve segmentation tasks Figure 2.8.

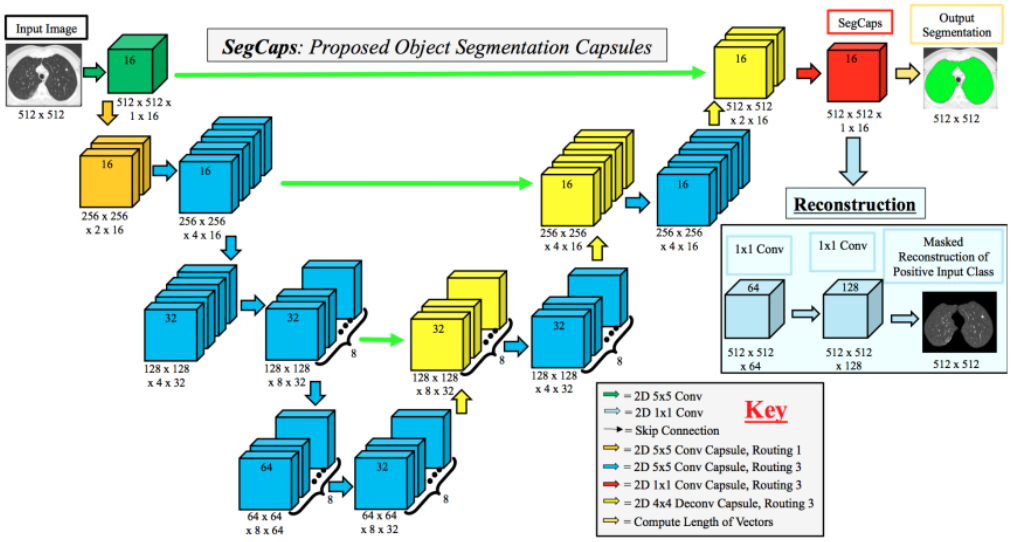


Figure 2.8: The architecture of SegCaps proposed in [27] for object segmentation.

They operate using transposed convolutions routing by the routing proposed above. This convolutiona-deconvolutional capsule architecture is far deeper than the original three-layers capsule network and extend the masked reconstruction of the target class as a method for regularization to the problem of segmentation.

These modifications allowed the authors to operate on large images, a slice of a CT scan of 512×512 pixel.

Each image is passed through a 2D convolutional layer which produces 16 feature maps of the same spatial dimension. This is the first set of capsules with a single capsule type: a grid of 512×512 each of which is a 16 dimensional vector. It's then followed by the first convolutional capsule layer. Generalizing mathematically, at a layer l we have a set of capsule types $T^l = \{t_1^l, \ldots, t_n^l \mid n \in \mathbb{N}\}$ and for every $t_i^l \in T \exists$ an $h^l \times w^l$ grid composed by chidl capsules z^l-dimensional. Here $h^l \times w^l$ is the spatial dimension of the output of previous layer $l-1$ and the capsules could be

written as:

$$C = \{c_{11}, \ldots, c_{1w^l}, c_{h^l 1}, \ldots, c_{h^l w^l}\} \tag{2.1}$$

where each c_{jk} is a vector z^l-dimensional. At the next layer $l+1$ we have T^{l+1} capsule types and a grid $h^{l+1} \times w^{l+1}$ of parent capsule each of which is $z^{l+1}-$dimensional. We could write these parents capsules as:

$$P = \{p_{11}, \ldots, p_{1w^{l+1}}, p_{h^{l+1}1}, \ldots, p_{h^{l+1}w^{l+1}}\} \tag{2.2}$$

This is the so called primary capsule layer, then we have the convolutional capsule layer in which every parent capsule $p_{xy} \in P$ receives a set of prediction vectors $\{\hat{u}_{xy|t_1^l}.\hat{u}_{xy|t_2^l}, \ldots, \hat{u}_{xy|t_n^l}\}$ one for each capsule type $t_i^l \in T^l$. Each of the prediction vectors is calculated as follow:

$$\hat{u}_{xy|t_i^l} = M_{t_i^l} \times U_{xy|t_i^l} \quad \forall t_i^l \in T^l \tag{2.3}$$

where $M_{t_i^l}$ is a transformation matrix learned via backpropagation and $U_{xy|t_i^l}$ is a sub-grid of child capsules with a defined kernel center at (x, y). The sub-grid has shape $k_h \times k_w \times z^l$ where $k_h \times k_w$ are the dimensions of the user-defined kernel. The transformation matrix has shape $k_h \times k_w \times z^l \times |T^{l+1} \times z^{l+1}$ and this doesn't depend on the spatial location (x, y): the same $M_{t_i^l}$ is shared across all spatial locations within a given capsule type t_i^l. The final input to each parent capsule $p_{xy} \in P$ is computed as follow:

$$p_{xy} = \sum_n r_{t_i^l|xu} \cdot |t_i^l \hat{u}_{xy|t_i^l} \tag{2.4}$$

where $r_{t_i^l|xu}$ are the routing coefficient computed by a "routing softmax ":

$$r_{t_i^l|xu} = \frac{exp(b_{t_i^l|xy})}{\sum_k exp(b_{t_i^l k})} \tag{2.5}$$

determined by dynamic routing with the update of the log prior probabilities $b_{t_i^l|xy}$ that $\hat{u}_{xy|t_i^l}$ should be routed to p_{xy}.

Using the kernel the creation of prediction vector in (2.3) is locally constraint and route only of child and parents belonged to the same space is allowed. After this, following the original procedure, the output capsule

is compute using non-linear squashing function:

$$v_{xy} = \frac{\|p_{xy}\|^2}{l + \|p_{xy}^2\|} \frac{p_{xy}}{\|p_{xy}\|} \tag{2.6}$$

where v_{xy} is the output of the capsule at (x, y) and p_{xy} is its final input calculated in (2.4).

2.2.2 Tr-CapsNet

Following the idea to expand the CapsNet applications, has recently been introduced a capsule-based neural network model to solve semantic segmentation problem [28]. The goal of image segmentation is to compute the probability of each pixel belonging to certain class type and the new solution proposed is specifically designed for this purpose. The new architecture is called Tr-CapsNet and is made up of three modules: feature extraction, capsule and traceback, upsampling (Figure 2.9).

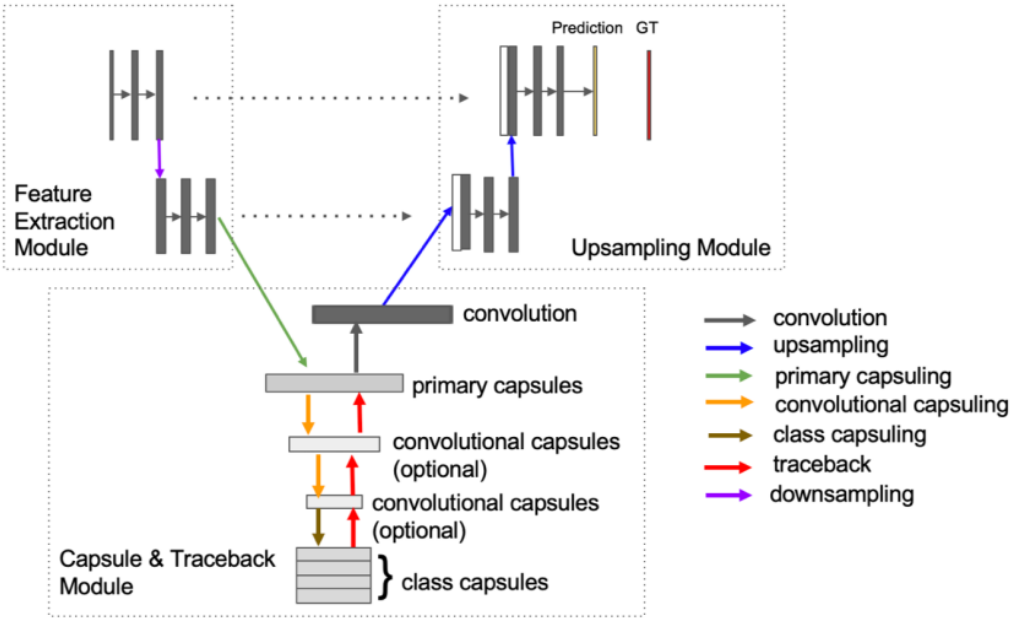


Figure 2.9: Overall architecture of our Tr-CapsNet. The traceback pipeline, shown as red arrows, is the major innovation of this paper. GT stands for ground-truth. [28]

The first has the scope to capture discriminative features of the input data and it consists in a sequence of convolutional layers. The second consists in primary capsule layer and class capsule layer then, in the same module, there is the *traceback pipeline*. This is the innovation proposed in the article and its aim is to produce class maps of the same size as the primary capsule which are taken as input in convolution layer. The third module is based on the deconvolution scheme, so it has to restore the original resolution by upsampling the label map computed in the previous layers. The traceback procedure is characterized by the possibility to infer the probability $P(\mathcal{C}_k)$ of a class label through repeated applications of the product rule and the sum rule, tracing the class label in the class capsule layer with backpropagation layer-by-layer.

The probability of a position belonging to certain class $P(\mathcal{C}_k)$ in layer l could be calculated as:

$$P(\mathcal{C}_k) = \sum_{i \in T^l} P(\mathcal{C}_k, i) = \sum_{i \in T^l} P(i) P(\mathcal{C}_k | i) \tag{2.7}$$

and its the same also for the original capsule network with i capsule type. The novelty is in let the likelihood of certain position taking $\mathcal{C}_k$ as its class label be available after inference reaches the next layer. So $P(\mathcal{C}_k|i)$ can be estimate as follow:

$$\begin{aligned} P(\mathcal{C}_k|i) &= \sum_{j \in T^{l+1}} P(\mathcal{C}_k, j|i) && (2.8) \\ &= \sum_{j \in T^{l+1}} P(j|i) P(\mathcal{C}_k|j, i) && (2.9) \\ &= \sum_{j \in T^{l+1}} c_{ji} P(\mathcal{C}_k|j) && (2.10) \end{aligned}$$

where in (2.8) we assign i to the parent j and in (2.10) we found i to belong to the same class of j. Here the coefficients c_{ij} are inferred with backpropagation. This mathematical derivations means that $P(\mathcal{C}_k|i)$ can be estimated with a layer-by-layer backward propagation procedure, starting at the layer $l - 1$, and repeatedly applying (2.10) to compute the conditional

probabilities for the lower layers. It could be written into a recursive equation with respect to the upper layer, if we assume that each lower-layer capsule only takes capsules in one particular position of the higher-layer as its possible parents.

This represents the simple case where each lower-layer capsule only take same-position capsules of the higher-layer as its possible parents. This is strictly connected with the assumption at the base of the CapsNet approach that "at each location in the image there is be at most one instance of the type of entity that a capsule represents"([11]). So it works when there is at most one instance of a category in the image but for the image data that have multiple instances of same classes, capsule approach has no guarantee to outperform CNNs in recognition accuracy. The traceback pipeline in Tr-CapsNet does not rely on the this one-instance assumption, in fact it provides that capsules in two or more positions might be the parents of a lower-layer capsule and a number of capsules take capsules at different positions in next layer as their parents. For these cases traceback procedure remains effective and the (2.10) should be modified as follow:

$$P(\mathcal{C}_k|i) = \frac{\sum_n P_n(\mathcal{C}_k|i)}{N} \tag{2.11}$$

where n is the $n-$th location for possible parents of the capsule i and N is the total location number. In the article [28] they obtained successful results for Tr-CapsNet, it outperform U-Net model both in MNIST data set both in its original and occluded version.

2.3 Motivations and data acquisition of brain images

Characterizing the cytoarchitecture of mammalian central nervous system on a brain-wide scale is becoming a compelling need in neuroscience.

For example, realistic modeling of brain activity requires the definition of quantitative features of large neuronal populations in the whole brain. Quantitative anatomical maps will also be crucial to classify the cytoarchtitectonic abnormalities associated with neuronal pathologies in a high reproducible manner[29].

Cellular localization and projections throughout the whole brain is another important step to understand brain functions.

These tasks are challenging both from a technological and computational point of view, in data acquisition and analysis. Techniques like Computer Tomography (CT) or Magnetic Resonance Imaging (MRI) do not yield cellular resolution, and mechanical slicing procedures are insufficient to achieve high-resolution reconstructions in three dimensions. The common used techniques in this research area are: Two-Photons Tomography, Two-Photons Fluorescence Microscopy (TPFM), Light Sheet Microscopy (LSM), Confocal Light Sheet Microscopy (CLSM). The LSM technique permits reconstruction of the whole brain with micron-scale resolution in a timescale ranging from hours to few days [30]. Contrast, resolution and timescale are the most important parameters to be optimized in order to achieve the best data acquisition. All of these techniques need also a preventive special procedure to clear tissue.

The possibility to use these advanced procedures for whole brain mapping opens a new challenge: handle big data and extract quantitative information from them.

For these reasons the Human Brain Project (HBP) has the aim to put in place a cutting-edge, ICT-based scientific research infrastructure, that will permit scientific and industrial researchers to advance our knowledge in the fields of neuroscience, computing and brain-related medicine. They make several specific grants to create initial versions of six separate ICT

Platforms and to make them available to external users. They will extend the initial capabilities of these Platforms and transform them into an integrated scientific research infrastructure. The Neuroscience Subprojects will extend their research in brain organization and theory to support the building of increasingly sophisticated models and simulations, as well as related work in brain-like computing and robotics, working up to replication of the whole mouse brain, while also laying the foundations for simulation of the much larger and more complex human brain.

In Firenze there is a group involved in one of the HBP Subprojects, the Biophotonics Group of LENS led by Francesco Saverio Pavone. The development and the application of new optical methodologies and the consequent acquisitions provide fundamental insights in the knowledge of the brain and his diseases and represent a completely new approach for the investigation of the physiology of neuronal network. LENS applies this approach in both mouse and human Subprojects in which it's involved.

Here we describe the workflow of LENS experimental lab [34]. It is mostly focused in high resolution imaging in mouse brain and also in human brain cortex, using advance techniques such as light spectroscopy and two photons florescence microscopy. These instruments acquired high-resolution images which produced huge data sets as big as 10^{12} voxels per single tomography, or several 70TB in terms of storage, so they need to be processed to limit memory occupancy.

The first thing they have to do in a processing pipeline is image stitching: these instruments produced images in the form of a grid of overlapping tiles then they need to fused together in order to reconstruct the global volume. To do this they used a stitching software developed in their laboratory called ZetaStitcher. After image stitching they are able to extract information out of their raw images using a deep learning approach. First there is a notation phase in which human experts create ground truth used to train the network and after prediction they are able to extract the spatial distribution of cells in the whole mouse brain or to do automatic cells segmentation classification.

The data set is pretty big so recently they have started to experiment with video compression algorithms to reduce data set size. A tomography

really like moving in a sense, consecutive tiles in a tomography look like consecutive frames in a video in a sense. This approach is very efficient to reduce the data set size. They are able to reduce from an uncompressed data set of 2.5TB to a 1.5GiB size for the same data set, maintaining really good images quality. Retaining very good quality suitable for visualization.

The whole mouse brain tomography was obtained with light sheet microscope, with sub-micron resolution. For this reason they are able to see individual neurons pretty well. This particular data set showed here is made up of 15×12 stacks, 23 GiB each, for a total of 4.2TiB. In order to obtain the global volume seen here they use the stitching software as said before.

An important phase of the processing pipeline is the manual annotation phase in which they generate markers to pin points the centroids of individual neurons in the whole mouse brain tomography, these markers are used to train a neural network to generate synthetic images using these markers.

Basically what the neural network does is enhanced neuronal bodies while discarded everything else in the image that is not neuronal bodies. It acts adding a non-linear filter so it's easier to determine the coordinate of every neuron.

Totally different data set a portion of the human brain cortex imaged at the two photons microscope. Again here they did manual segmentation drawing the contours in individual neurons to classify them in their shape and size. After training a neural network they were able to do automatic segmentation of the cells shape. This works using a software frame named ALIQUIS developed from Bioretics researchers [35], specializing in machine learning for computer vision. With this technique it is possible to see the 3D rendering, a reconstruction of cellular bodies in shape in order to study how they are distributed in the space.

The reconstruction of the imaged volume is allowed thanks to a stitching tool that overlap all the tiles acquired in a proper way. After this step the final image results made up of 10^9 pixels, with a file size on the order of GiB [31].

The progress on the analysis of these kind of data-set achieves the result of automatic identification of neurons first in mice and then in humans brain. The algorithm used goes from clustering to segmentation and we can enumerate and classify brain cells in real time allowing a deep understanding of brain functionality.

In the sections below hints of the analysis from mice and human brain are presented.

2.4 Mouse brain data analysis

There are a lot of studies about the analysis of mouse brain images that try to identify Purkinje cells in order to classify their *soma* [3]. Purkinje cells are a central part of the cerebellum, the part of the brain that plays an important role in motor learning, fine motor control of the muscle, equilibrium and posture but also influences emotions, perception, memory and language.

These cells are some of the largest neurons with an intricately elaborate dendritic arbor, characterized by a large number of dendritic spines which form nearly two dimensional layers through which parallel fibers from the deeper-layers pass. A mouse brain has a volume of the order of $1cm^3$, TB scale at the micron-resolution.

There are studies that propose automatic localization or segmentation of cell bodies in 2D and 3D microscopy. Frasconi et al. introduced [30] an algorithm for fully automated cell identification, addressing the problems of handle large data-set and a contrast variability. The algorithm is based on mean shift clustering to detect soma centers, supervised semantic deconvolution and manifold learning to filter false positive and false negative. They produced the first complete map of a selected neuronal population, Purkinjie cells, in a large area of the mouse brain, cerbellum cortex. The most important step of their method is the mean shift clustering, which is made up of three major part: substacking, cell identification and thresholding.

So the very first step is to partition the 3D images into a set of substacks

[3] Soma, or cell body, is the bulbous, non-process portion of a neuron or other brain cell type, containing the cell nucleus.

of size $W \times H \times D$ and then overlap each other of a length M to ensure that every cell with a center detected inside the substack of size $(W-M)\times(H-M)\times(D-M)$ falls entirely within the very first substack (Figure 2.10). This procedure allows to avoid border effects.

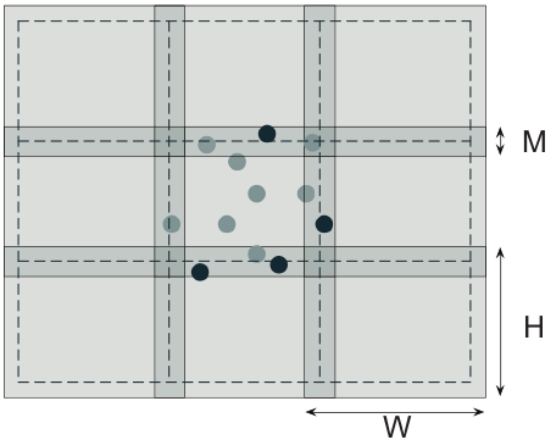


Figure 2.10: Overlapping of substacks (depicted in 2D for simplicity). Sample accepted and rejected after processing the central substack are shown as light and dark circles, respectively [30].

After that the goal is to group together voxels belonging to the same soma, in other words it's to group together pixels sharing similar features or colors. The number of cells is unknown and so is the number of clusters.

The approach used to clustering is non-parametric and it's a variant of the mean shift. In stead of place a kernel on each available data point they proposed to use a chosen set of so called 'seed' S, determined by the extraction of the local maxima after a convolution with a normalized spherical filter applied on all local maxima.

Then the algorithm goes on with mean shift using a spherical kernel:

$$K(\vec{a}) = \begin{cases} 1 & \text{if} \quad \|\vec{a}\| < R \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

All of the points analyzed get assigned to the "center of mass"of points falling within the sphere defined by the kernel function. Finally thresholding is applied to the results in order to limit the number of false positive detection. They set 3 ranges of voxels intensities and compute by maximum entropy the two delimiting thresholds Θ_1 and Θ_2, identifying background in the range $[0, \Theta_1]$ and foreground in the range $[\Theta_1, \Theta_2]$ and $[\Theta_2, \infty)$. Regions with a non uniform intensity need to be filtered. This is the whole point of semantic deconvolution. This step is carried out in a supervised fashion: biologists annotated 10 substacks making the location of the true centers in order to have 10 labeled training data.

Then a neural network is used as non-linear convolutional filters, trained on cubic patchs of 2197 voxels. The neural network is made up of 2 fully connected hidden layers, one with 500 and the other with 200 units, and a sigmoidal output layer.

For backpropagation they trained the network for ≈ 100 epochs of stochastic gradient descent with momentum and with a minibatch size of 10.

The goal is to predict, for each voxel, the conditional probability that it falls in a white area of the original image.

They observed that the performance of the mean shift algorithm increases when applied to the image cleaned by the semantic deconvolution technique (Figure 2.11).

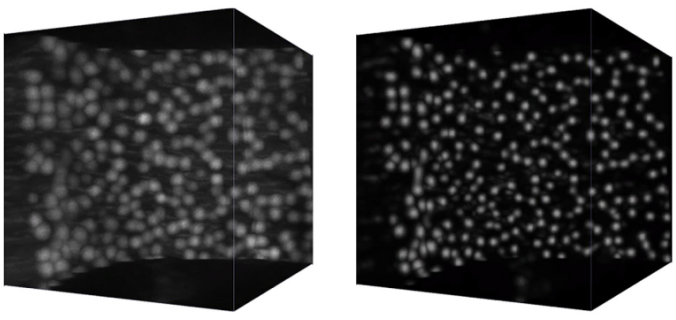


Figure 2.11: A small volume from the mouse cerebellum before (left) and after (right) semantic deconvolution. On the final image is much easier to run a reliable automatic localization algorithm [29].

The last procedure described in the article take into account the architecture of the cerebellum cortex. It folds into folia so it's naturally modeled as manifolds.

So apply this model means consider the cell's organization not as a random distribution in 3D space but as a pattern with fixed distances. This step contributed to remove false positive.

So with this technique the provide an estimate of the number of cells and a map of their spatial distribution (Figure 2.12).

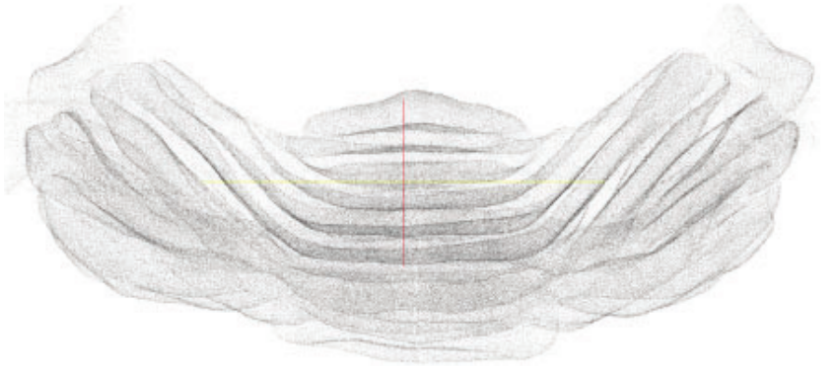


Figure 2.12: With the application of the manifold filter, the algorithm detected 224222 Purkinje cells in the whole cerebellum image. This is the final set of predicted cell centers as a point cloud [30].

Human supervision is needed only for the initial training of a neural network, but it's able to generalize so this semantic deconvolution can perform well for the same cell type of other brain.

Another important work in this research area was done by Silvestri et al. [29].

They used the previous techniques and, starting from the cloud of points representing all the Purkinjie neurons, performed a further analysis (Figure 2.13).

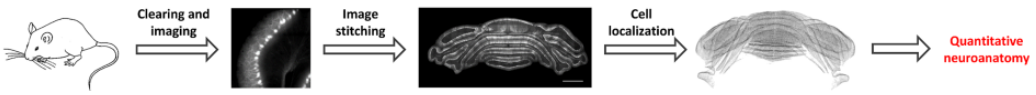


Figure 2.13: Experimental pipeline used in [29] for large-volumes quantitative neuroanatomy. After acquisition and processing, the final raw image stacks are stitched together and a software for automatic cell localization is applied. The resulting cloud of points representing the position of labeled cells can be the starting point for many different quantitative neuroanatomical analysis.

They highlight both the clusterization properties of the point cloud and their distribution in the layer localizing the gap between them.

These measurements can provide robust insights into the distribution of Purkinjie cells under different physiological or pathological conditions.

2.5 Human brain data analysis

To study human brain samples a new technique was recently developed by Italian researchers in Firenze at LENS. The used Two-Photons fluorescence microscopy on tissues previously cleared, cutted and incubated.

This method, as like as LSM, produces a mosaic of overlapping 3D stacks so to recreate the imaged volume stitching procedure was used. To analyze these data in [31] they proposed to use image segmentation. Segmentation methods try to label each pixel of the image and then extract a semantic picture of the scene, splitting background from foreground. In this work they proposed a segmentation of neurons in 3D images of human brain cortex working at the level of local visual pattern, as texture, rather than of single pixels. To address this problem they search for the solution in deep convolutional neural network architectures. The procedure starts considering each slice of the z-stack as independent image and a standard CNN with 3 layers is adopted to classify each single pixel of each slice. The CNN used was made up of $(32, 64, 64)$ kernels of $(5\times5, 3\times3, 3\times3)$ size with ReLu activation function followed by 2×2 max pooling and by 2 fully connected layers of 128 neurons with ReLu and a softmax output layer. This classification returns the probability of that label predicted for

that pixel is the correct one and these resulting probabilistic values are plotted into the heatmap [4].

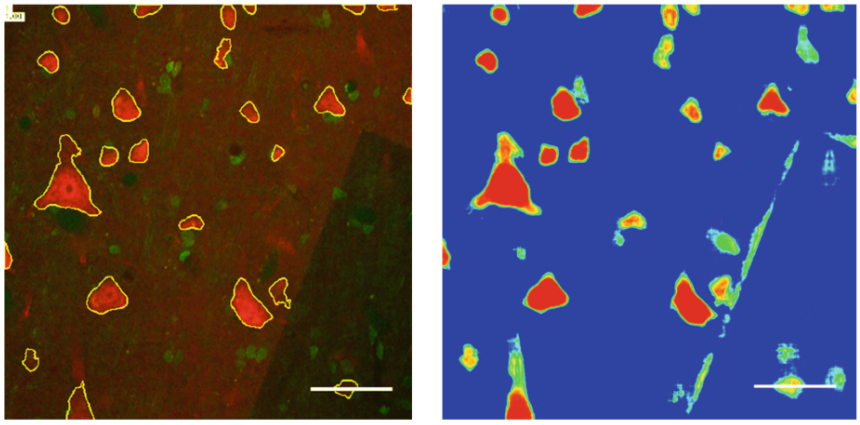


Figure 2.14: Heatmap from [31].

This step is followed by a probabilistic blob detection in which a contour finding algorithm localized objects in the heatmap. Here it's possible to set the desired confidence value for accepting or rejecting objects in the contour finder.

Subsequent reconstruction of the 2.5D volume is performed via computer graphics algorithms.

The segmentation procedure described above is a supervised classification: as like as for the mouse's brain, biologists annotated manually 29 images finding 104 kinds of neuron. With the analysis they found 88 neurons correctly segmented with 12 false positive blobs, afforded a sensitivity of about 85%.

So the classification of human brain cells was done in a binary way identifying only if the object is a neuron or not, but the same method could be applied in multiple classification to discriminate different neuronal classes (Figure 2.15).

[4] A heatmap is a graphical representation of data where the individual values contained in a matrix are represented as colors.

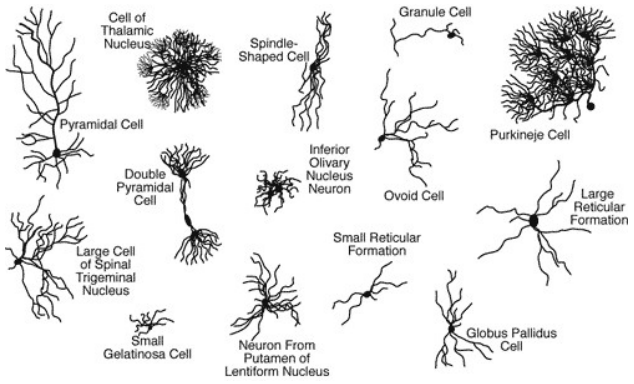


Figure 2.15: There are as many as 10,000 specific types of neurons in the human brain, this image identifies how neurons come in various shapes and sizes, called neurons morphological types [32]

In fact, considering the shape of the neurons, the algorithm could distinguish, for example, pyramidal to non-pyramidal cells. This result is very useful to researchers that work on quantitative histological analysis of biological tissue.

Other studies was done on the analysis of human brain cells, to count them and classify [33], but the most important and exciting feature of recent studies is that they can use acquisition techniques in vivo. That allows more possibilities both in research and diagnostic.

CHAPTER 3

MNIST ANALYSIS

"Se tu sapessi quel che stai facendo probabilmente ti annoieresti.
Solo perchè ti annoi non vuol dire che sai cosa stai facendo."

In this chapter we propose the very first attempt to implement networks for image classification task. The experimental application was made using a public online data set and the aim is to compare the structure and the results of CNN and CapsNet. The use of a very known data set is always the first experimental step to study and understand an innovative approach, like CapsNet, so the focus is not on the data result itself but more on how the net works.

MNIST is the most famous database used for machine learning and its name means *Modified National Institute of Standards and Technology*: NIST is a physical science laboratory, an agency of the United States Department of Commerce; "modified" because it's a combination of two of NIST's databases, Special Database 1 and Special Database 3, consisting respectively of digits written by high school students and employees of the United States Census Bureau. So MNIST is created by "re-mixing" the samples from NIST's original data sets. The whole data set contains handwritten digit images [36]. It's divided in $60K$ examples for the training set and

$10K$ examples for testing. In many papers the official training set is divided into an actual training set of $50K$ examples and $10K$ validation examples, when it is necessary for selecting hyper-parameters like learning rate and size of the model.

All of the images have been size-normalized and centered, by subtracting the mean from every feature, in a fixed image of 28×28 pixels. Each pixel is represented by a value in $[0; 255]$, from black to white with different shades of gray. The data set also contains a label for each image.

An image is represented as a 1-dimensional array of 784 (28×28) float values between 0 and 1, from black to white. The labels are numbers between 0 and 9 indicating which digit the image represents.

This is an example of the output to visualize the data:

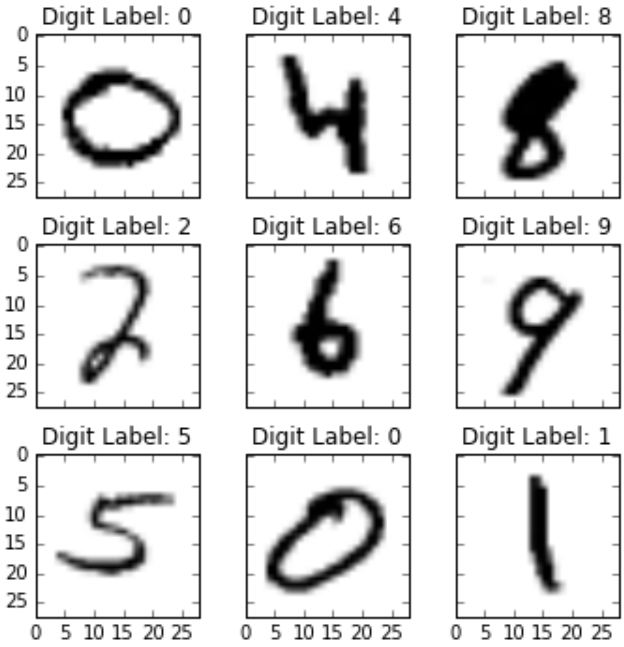


Figure 3.1: An example of 9 data expressed as image in black and white, with the respective true label assigned.

As we can see in the following sections, the results on this data set lead us to consider the CapsNet approach equal to CNN one. In fact for the classification task for which we tested both the architectures we obtained good comparable performance in terms of accuracy.

3.1 A CNN approach

For this analysis the data set is organized with shape [samples][width] [height][pixels] and the one hot encoder is eventually used:

```
x_train=
x_train.reshape(x_train.shape[0],28,28,1).astype('float32')
x_test=
x_test.reshape(x_test.shape[0],28,28,1).astype('float32')
y_train=tf.keras.utils.to_categorical(y_train)
y_test=tf.keras.utils.to_categorical(y_test)
```

The model

We implemented a Convolutional Neural Network in TensorFlow to analize MNIST Data. In particular we used Keras, a high-level API to build and train models that includes first-class support for TensorFlow-specific functionality [37]. The model is made up of:

- Convolutional Layer with 6 filters (5,5) and no padding;
- Max Pooling Layer with pool size (2,2);
- Convolutional Layer with 16 filters (5,5) and no padding;
- Max Pooling Layer with pool size (2,2);
- Fully Connected Layer with 100 neurons and ReLU activation function;
- Output Layer with SoftMax activation function.

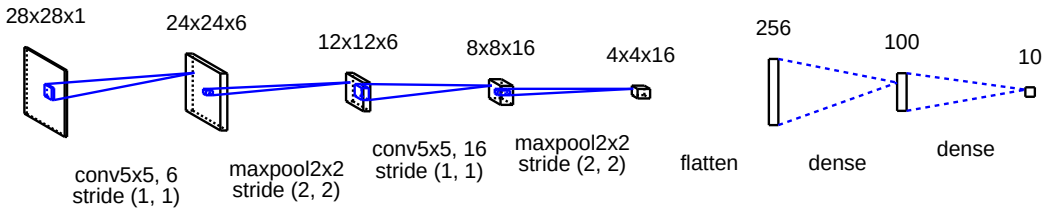


Figure 3.2: Representation of the CNN architecture used in this project.

The code for this model is:

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(filters = 6,
                                  kernel_size = (5, 5),
     padding = 'valid', data_format='channels_last',
     input_shape= (28, 28, 1)),
  tf.keras.layers.MaxPool2D(2, 2),
  tf.keras.layers.Conv2D(filters = 16,
             kernel_size = (5, 5), padding = 'valid'),
  tf.keras.layers.MaxPool2D(2, 2),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(100, activation=tf.nn.relu),
  tf.keras.layers.Dense(num_classes,
                        activation=tf.nn.softmax)])
```

We can summarize the model in the following table:

Layer Type	Output Shape	Parameters
conv1	(None, 24, 24, 6)	156
maxpool1	(None, 12, 12, 6)	0
conv2	(None, 8, 8, 16)	2416
maxpool2	(None, 4, 4, 16)	0
flatten	(None, 256)	0
fc	(None, 100)	25700
output	(None, 10)	1010

Training and testing

The code for the model configuration is:

```
model.compile(optimizer='adam',
  loss='categorical_crossentropy', metrics=['accuracy'])
```

where we choose the Adam algorithm as training procedure and cross-entropy loss. Adam is an adaptive moment estimator that computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods [38].

Using the default setting for parameters we have:

- learning rate = 0.001
- beta_1=0.9
- beta_2=0.999
- decay = 0.0 .

The cross-entropy loss is useful when the classes are mutually exclusive. It's defined as:

$$L_i = -f_{y_i} + \log\left(\sum_j e^{f_j}\right) \tag{3.1}$$

where f_j is the j-th element of the vector of class scores f while f_{y_i} is the true label [6].
To train the model we use the code:

```
train = model.fit(x_train, y_train, epochs=100,
        validation_data=(x_test, y_test))
```

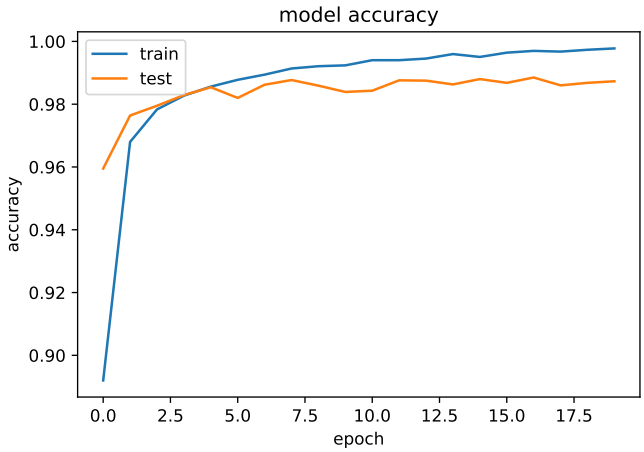
where we used test set as validation test. We tried two different configuration, changing the batch_size and the epochs.

So I've done two simulation with the following parameters:

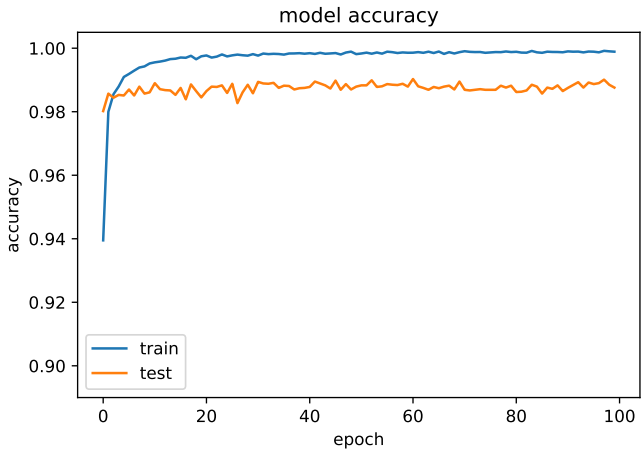
- Batch_size=128, Epochs=20;
- Batch_size=32, Epochs=100.

Results

The results are:



(a) Batch_size=128, Epochs=20, Test accuracy: 0.9873



(b) Batch_size=32, Epochs=100 Test accuracy: 0.9876

Figure 3.3: These are the accuracies of the CNN model in function of the number of epoch.

3.2 A CapsNet approach

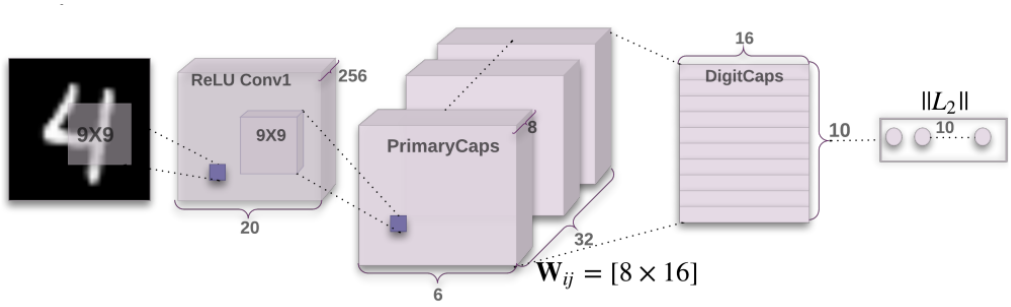
For this analysis the data set is organized with shape [samples][width][height] [pixels] and the one hot encoder is eventually used, as well as in previous CNN implementation.

The model

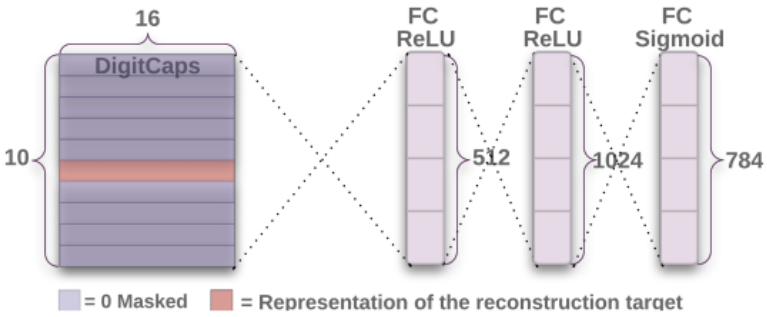
An implementation [39] of the Capsule Network was used to analize MNIST Data, based on the paper "Dynamic Routing Between Capsules" [11] in which how to use *capsule* with an algorithm named *routing-by-agreement* is pointed out. The model is made up of (Figure 3.4):

- Convolutional Layer with 256 filters $(9, 9)$, no padding and ReLU activation function;
- Primary Capsule Layer with 32 channels of convolutional 8D capsules $(9, 9)$;
- Digit Capsule Layer with 10 capsules 16D;
- Decoder Network made up of:
  - Fully Connected Layer with 512 neurons, ReLU activation function;
  - Fully Connected Layer with 1024 neurons, ReLU activation function;
  - Output Layer with sigmoid activation function.

The dimensions of a digit capsule represent the variations: one always represents the width of the digit, others are digit-specific variations, combinations of global variations or variations in a localized part of the digit. We can manipulate the images to see the properties of the capsules.



(a) The model.



(b) The Decoder Network.

Figure 3.4: A rapresentation of the Capsule Network [11].

The code for this model follows.

```
# Layer 1: Just a conventional Conv2D layer
conv1 = layers.Conv2D(filters=256, kernel_size=9,
       strides=1, padding='valid', activation='relu',
                                        name='conv1')(x)
```

This layer converts pixel intensities to the activities of local feature detectors that are then used as inputs to the primary capsules.

```
# Layer 2: Conv2D layer with 'squash' activation
# then reshape to [None, num_capsule, dim_capsule]
primarycaps = PrimaryCap(conv1, dim_capsule=8,
n_channels=32, kernel_size=9, strides=2, padding='valid')
```

Each primary capsule output sees the outputs of all 256×81 Conv1 units whose receptive fields overlap with the location of the center of the capsule.

```
# Layer 3: Capsule layer. Routing algorithm works here.
digitcaps = CapsuleLayer(num_capsule=n_class,
                      dim_capsule=16, routings=routings,
                          name='digitcaps')(primarycaps)
```

each of these capsules receives input from all the capsules in the layer below.

```
# Decoder network.
y = layers.Input(shape=(n_class,))
masked_by_y = Mask()([digitcaps, y])
masked = Mask()(digitcaps)

# Shared Decoder model in training and prediction
decoder = models.Sequential(name='decoder')
decoder.add(layers.Dense(512, activation='relu',
```

```
input_dim=16*n_class))
decoder.add(layers.Dense(1024, activation='relu'))
decoder.add(layers.Dense(np.prod(input_shape),
activation='sigmoid'))
decoder.add(layers.Reshape(target_shape=input_shape,
name='out_recon'))
```

Decoder structure is used to reconstruct a digit from the DigitCaps layer representation. The output of the digit capsule is fed into a decoder consisting of 3 fully connected layers that model the pixel intensities.

Each layer is implemented following the procedure in the article [11].

We can summarize the model as in Figure 3.5 and in the following table:

Layer Type	Output Shape	Parameters
input	(None, 28, 28, 1)	0
conv1	(None, 20, 20, 256)	20992
primarycaps conv2	(None, 6, 6, 256)	5308672
primarycaps reshape	(None, 1152, 8)	0
primarycaps squash	(None, 1152, 8)	0
capsule layer	(None, 10, 16)	1474560
input2	(None, 10)	0
mask	(None, 160)	0
capsnet	(None, 10)	0
decoder	(None, 28, 28, 1)	1411344

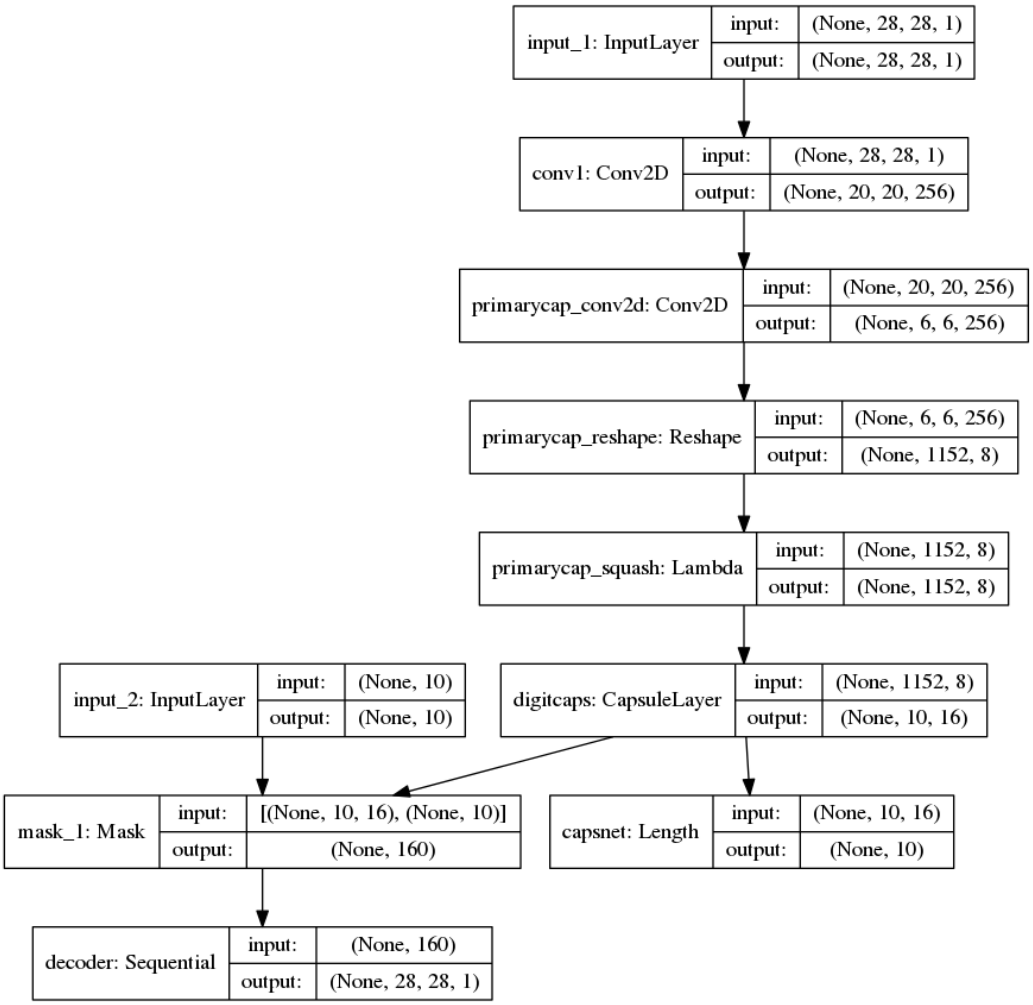


Figure 3.5: A scheme for the model used.

Training and testing

The code for the model configuration is:

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

The margin loss is defined as (1.29) it's written in the code as follows:

```
def margin_loss(y_true, y_pred):
 L = y_true * K.square(K.maximum(0., 0.9 - y_pred)) +
     0.5 * (1 - y_true) * K.square(K.maximum(0.,
                                             y_pred - 0.1))
 return K.mean(K.sum(L, 1))
```

To train the model we used the code:

```
model.fit_generator(generator=train_generator(x_train,
   y_train, args.batch_size, args.shift_fraction),
   steps_per_epoch=int(y_train.shape[0] / args.batch_size),
   epochs=args.epochs,
   validation_data=[[x_test, y_test], [y_test, x_test]],
   callbacks=[log, tb, checkpoint, lr_decay])
```

Reconstruction

During training, all but the activity vector of the correct digit capsule are masked out. Then this activity vector is used to reconstruct.

Mean squared error is used as the reconstruction loss and the coefficient for the loss is lam_recon $= 0.0005 \times 784 = 0.392$. This should be equivalent with using sum squared error and lam_recon $= 0.0005$ as in the paper. This different digits are reconstructed from different feature vectors (digit capsules). These vectors are mutually independent during reconstruction.

The code used to reconstruct the images is:

```
img=combine_images(np.concatenate([x_test[:50],
                                   x_recon[:50]]))
image = img * 255
```

Manipulation

The trained CapsNet is moderately robust to small affine transformations of the training data. After computing the activity vector for the correct digit capsule, this activity vector can be perturbated using the decoder network. The aim is to see how the perturbation affects the reconstruction, learning what the individual capsule dimensions represent.

The code used to perturbate with affine transformations is:

```
index = np.argmax(y_test, 1) == args.digit
number = np.random.randint(low=0, high=sum(index) - 1)
x, y = x_test[index][number], y_test[index][number]
x, y = np.expand_dims(x, 0), np.expand_dims(y, 0)
noise = np.zeros([1, 10, 16])
x_recons = []
for dim in range(16):
for r in [-0.25, -0.2, -0.15, -0.1, -0.05, 0, 0.05, 0.1,
        0.15, 0.2, 0.25]:
tmp = np.copy(noise)
tmp[:,:,dim] = r
x_recon = model.predict([x, y, tmp])
x_recons.append(x_recon)
x_recons = np.concatenate(x_recons)
img = combine_images(x_recons, height=16)
image = img*255
```

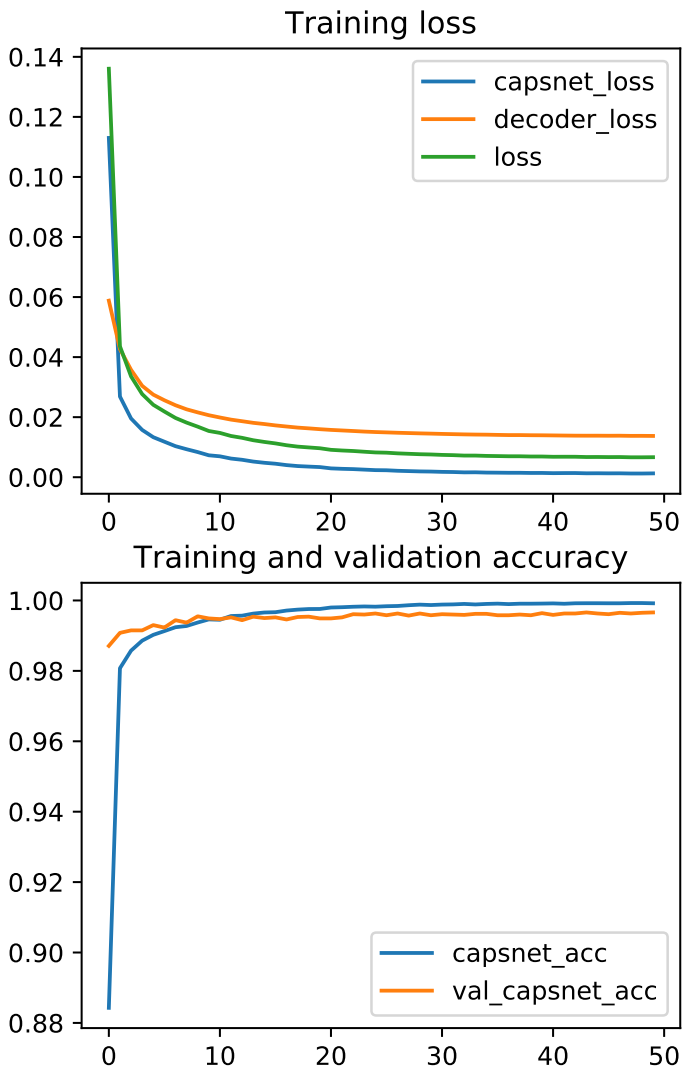
where each capsule dimension is tweaked by intervals of 0.05 in the range $[-0.25, 0.25]$.

Results

With the setting parameters:

- decay factor $= 0.9$;
- routing $= 3$;

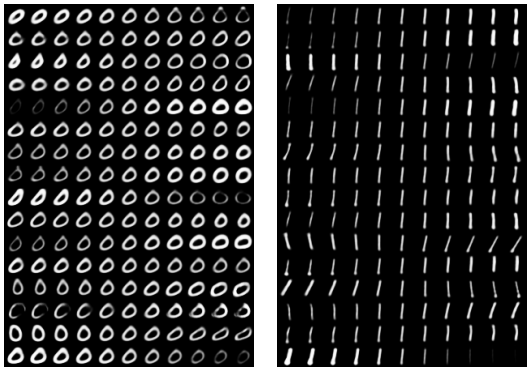
the results are:



The test accuracy doesn't improved from 0.9966 so the test error is 0.34%.

The reconstruction gives these results:

Figure 3.6: Digits at top 5 rows are real images from MNIST and digits at bottom are corresponding reconstructed images.



(a) (b)

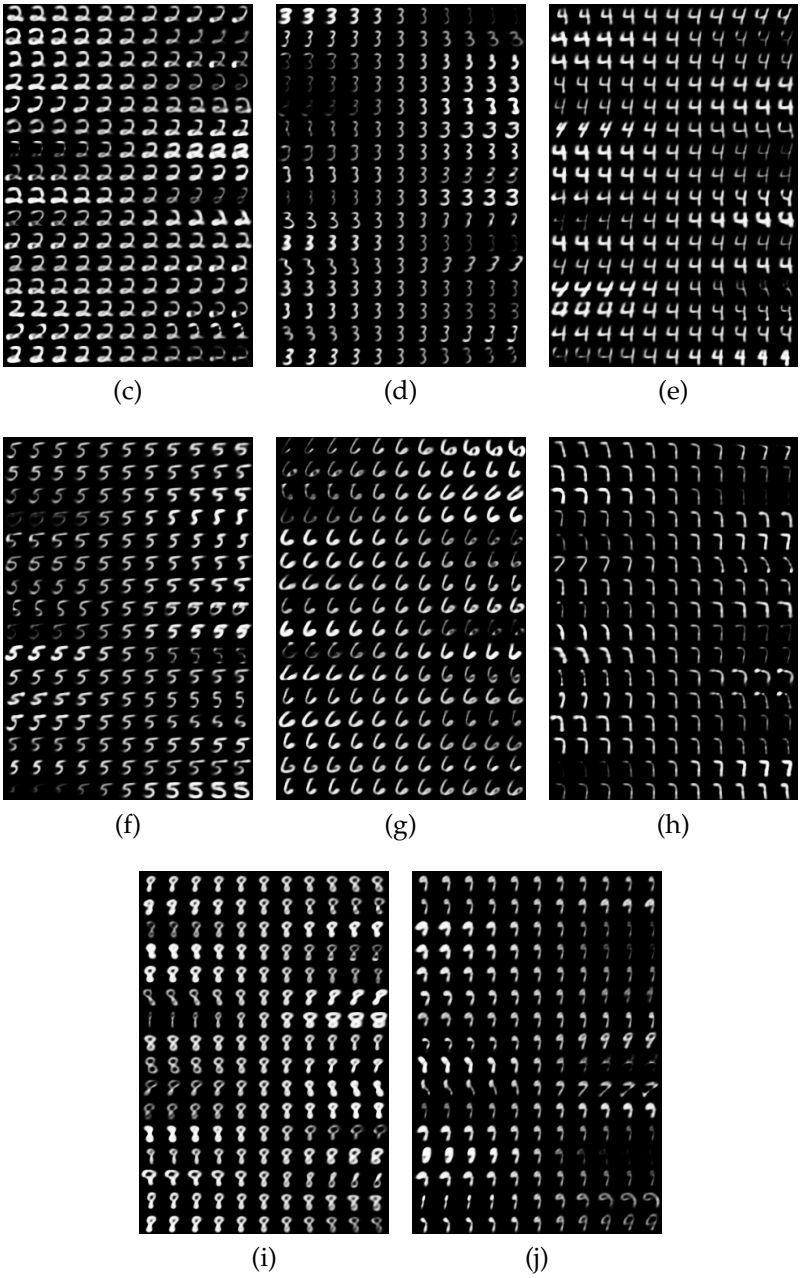


Figure 3.6: For each digit, the i-th row corresponds to the i-th dimension of the capsule, and columns from left to right correspond to adding $[-0.25, -0.2, -0.15, -0.1, -0.05, 0, 0.05, 0.1, 0.15, 0.2, 0.25]$ *to the value of one dimension of the capsule.*

CONCLUSION

The qualities of the CapsNet's approach come surely from the idea of reproducing more faithfully the human brain mechanism of vision. The introduction of spatial relationship between parts in the object leads to better performance in recognition tasks because in this way the object representation doesn't depend on the point of view of the observer. This is important because this net doesn't require a lot of training data, one for each viewpoint, to learn how to classify that object. Using CapsNet we can train better models with less training data which is quite beneficial for medical image analysis where the annotations are limited and expensive to get. CNN is also vulnerable to adversaries by simply moving, rotating or resizing individual features while CapsNet is more robust under these transformations, it also has better performance in noisy or occulted images. Moreover CapsNet doesn't lose as much information as CNN with pooling procedure. Another important quality of this new approach concerns the number of parameters that has to be inferred in training: it was been verified that compared with U-net for a Capsule Network it decreases by 95%.

CapsNet flaws are more practical than theoretical: the way in which this innovative *idea* is put into code and achieves practical results is the real problem from the very beginning. Although if GPU helps a lot in the realization of CapsNet experiment, the are many issues that haven't

been solved yet. First of all the main algorithm in capsule layer has a very huge training time, also to train on not very big images. CapsNet is slower than its CNN counterpart because of the routing update iterations and has not proven yet its effectiveness in large-scale visual recognition problems. Moreover each element in the activity vector does not always represent meaningful properties of the input image.

Capsule Networks are nowadays used in image classification projects and also in image segmentation for both research and diagnostic tasks.

Moreover they are used as auto-encoders to generate images, as it was proposed in the very first article where Capsule Theory was introduced. The original aim of capsules was not to recognize images but to extract pose information from input images and to create an image of the same object with a different chosen pose.

Working with this new architecture introduces many challenges. Current implementations are much slower than other modern deep learning models. Time will show if capsule networks can be trained quickly and efficiently. In addition, it remains to be verified if they work well on difficult data sets and in different domains [9].

In any case, Capsule Network is a very interesting and already working model which will definitely get developed further over time and contribute to further expansion of deep learning application domain.

Future perspectives in these studies have to test many different typologies of data sets and also to test the architecture for other imaging tasks. One of the next possible achievements for deep learning researchers is to optimize the CapsNets implementation in order to reduce training time. Once these improvements are realized, CapsNets will swiftly replace CNN in all imaging tasks.

APPENDIX

Data analysis with TensorFlow

TensorFlow (TF) is a framework to define and run computations involving tensors [37]. A tensor is a generalization of vectors and matrices to potentially higher dimensions. Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes.

TF is based on multiple API (Application Program Interface) layers, classified in different levels:

- High-Level: like Estimators, Keras; they are methods to train the model;
- Mid-Level: the whole structures like layers, data set, metrics; all the objects that can be used to train a model;
- Low-Level: is basically Python language.

One can use TF choosing the level in order to take advantage of the potentiality of this framework.

Starting from the high-level API we can describe how Keras works. Keras is a high-level API to build and train deep learning models. It's used for fast prototyping, advanced research, and production, with three key advantages: user friendly, modular and easy to extend.

Keras has a simple, consistent interface optimized for common use cases. It provides clear and actionable feedback for user errors. Keras models are made by connecting configurable building blocks together, with few restrictions.

From a low-level point of view TensorFlow programs work by first building a graph of tf.Tensor objects, a partially defined computation that will eventually produce a value.

Then there is the session that has to evaluate the graph. A session encapsulates the state of the TensorFlow runtime, and runs TensorFlow operations.

The characteristic of the computational graph is that it is *static*, it means that first the whole graph is created and then it is all executed by the session. The tensors used are called *place holder* because, when they are defined into graph, they have the only aim of keeping memory that will be used only in run section. So they keep only the information about the operations that they will do and about the shape of the output.

Let's see how graph and session work in details.

Graph

A tf.Graph contains two relevant kinds of information: graph structure and graph collections.

Graph structure are the ensamble of nodes and edges of the graph, indicating how individual operations are composed together, but not prescribing how they should be used. The graph structure is like assembly code: inspecting it can convey some useful information, but it does not contain all of the useful context that source code conveys.

So a computational graph is composed of two types of objects: tf.Operation, the nodes of the graph, operations describe calculations that consume and produce tensors; tf.Tensor, the edges in the graph. These represent the values that will flow through the graph. Most TensorFlow functions return tf.Tensors.

In a dataflow graph, the nodes represent units of computation, the operations, while the edges represent the data consumed or produced

by a computation, the tensors. Most TensorFlow programs start with a dataflow graph construction phase. In this phase, you invoke TensorFlow API functions that construct new tf.Operation (node) and tf.Tensor (edge) objects and add them to a tf.Graph instance. TensorFlow provides a default graph that is an implicit argument to all API functions in the same context.

High-level APIs such as the tf.estimator.Estimator API manage the default graph on your behalf, and for example may create different graphs for training and evaluation.

A tf.Graph object defines a namespace for the tf.Operation objects it contains. TensorFlow automatically chooses a unique name for each operation in your graph, but giving operations descriptive names can make your program easier to read and debug.

Each API function that creates a new tf.Operation or returns a new tf.Tensor accepts an optional name argument.

The graph visualizer uses name scopes to group operations and reduce the visual complexity of a graph. See Visualizing your graph for more information.

Note that tf.Tensor objects are implicitly named after the tf.Operation that produces the tensor as output.

Many TensorFlow operations take one or more tf.Tensor objects as arguments.

Tensor objects will accept a tensor-like object in place of a tf.Tensor, and implicitly convert it to a tf.Tensor. Tensor-like objects include elements of the following types: tf.Tensor, tf.Variable, numpy.ndarray, list, Scalar Python types: bool, float, int, str.

TensorFlow, as the name indicates, is a framework to define and run computations involving tensors. A tensor is a generalization of vectors and matrices to potentially higher dimensions. Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes.

When writing a TensorFlow program, the main object you manipulate and pass around is the tf.Tensor. A tf.Tensor object represents a partially defined computation that will eventually produce a value. TensorFlow programs work by first building a graph of tf.Tensor objects, detailing how

each tensor is computed based on the other available tensors and then by running parts of this graph to achieve the desired results.

A tf.Tensor has the following properties:

- **Type**
  Each element in the Tensor has the same data type, and the data type is always known. The shape (that is, the number of dimensions it has and the size of each dimension) might be only partially known. Most operations produce tensors of fully-known shapes if the shapes of their inputs are also fully known, but in some cases it's only possible to find the shape of a tensor at graph execution time.

  Some types of tensors are special, and these will be covered in other units of the TensorFlow guide. The main ones are: tf.Variable, tf.constant, tf.placeholder, tf.SparseTensor.

  With the exception of tf.Variable, the value of a tensor is immutable, which means that in the context of a single execution tensors only have a single value. However, evaluating the same tensor twice can return different values; for example that tensor can be the result of reading data from disk, or generating a random number.

  It is not possible to have a tf.Tensor with more than one data type. It is possible, however, to serialize arbitrary data structures as strings and store those in tf.Tensors.

  A tf.Tensor has the following properties:

- **Shape**
  The shape of a tensor is the number of elements in each dimension. TensorFlow automatically infers shapes during graph construction. These inferred shapes might have known or unknown rank. If the rank is known, the sizes of each dimension might be known or unknown.

- **Rank**
  The rank of a tf.Tensor object is its number of dimensions. Synonyms for rank include order or degree or n-dimension. Note that rank in

TensorFlow is not the same as matrix rank in mathematics.Each rank in TensorFlow corresponds to a different mathematical entity:

Rank	Math Entity
0	Scalar
1	Vector
2	Matrix
3	3-Tensor
n	n-Tensor

Session

TensorFlow uses a dataflow graph to represent your computation in terms of the dependencies between individual operations. This leads to a low-level programming model in which you first define the dataflow graph, then create a TensorFlow session to run parts of the graph across a set of local and remote devices. TensorFlow uses the tf.Session class to represent a connection between the program, although a similar interface is available in other languages, and the C++ runtime. A tf.Session object provides access to devices in the local machine, and remote devices using the distributed TensorFlow runtime. It also caches information about your tf.Graph so that you can efficiently run the same computation multiple times.

The command tf.Session.init accepts three optional arguments: target; the address of a TensorFlow server, which gives the session access to all devices on machines that this server controls; graph, you can specify an explicit tf.Graph to run different from default corrent one; config, to control the behavior of the session.

The tf.Session.run method is the main mechanism for running a tf.Operation or evaluating a tf.Tensor. You can pass one or more tf.Operation or tf.Tensor objects to tf.Session.run, and TensorFlow will execute the operations that are needed to compute the result.

REFERENCES

[1] A. Esteva and alt. "A guide to deep learning in healthcare". In: *Nature Medicine* (2019).

[2] *What is wrong with CNN?* 2014. URL: https://www.youtube.com/watch?v=rTawFwUvnLE.

[3] *Can digital computers think?* URL: http://www.turingarchive.org/browse.php/B/5.

[4] D. H. Ballard and C. M. Brown. *Computer Vision*. 1982.

[5] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deep-learningbook.org. MIT Press, 2016.

[6] *CS231n: Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.stanford.edu/.

[7] *Loss Functions in Neural Networks*. URL: https://isaacchanghau.github.io/post/loss_functions/.

[8] H. Noh, S. Hong, and B. Han. "Learning Deconvolution Network for Semantic Segmentation". In: *arXiv e-prints* (May 2015).

[9] *Understanding Hinton's Capsule Networks. Part I: Intuition.* 2017. URL: https://medium.com/ai%C2%B3-theory-practice-business/understanding-hintons-capsule-networks-part-i-intuition-b4b559d1159b.

[10] G. E. Hinton, A. Krizhevsky, and S. D. Wang. "Transforming Auto-Encoders". In: *Artificial Neural Networks and Machine Learning – ICANN 2011*. Ed. by T. Honkela et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 44–51.

[11] G. E. Hinton, S. Sabour, and N. Frosst. "Dynamic Routing Between Capsules". In: 2017.

[12] G. E. Hinton, S. Sabour, and N. Frosst. "Matrix capsules with EM routing". In: 2018.

[13] A. Géron, ed. *Introduction to Capsule Networks (CapsNets)*.

[14] N. F. S. Sabour and G. E. Hinton, eds. *Dynamic Routing Between Capsules*.

[15] K. Qiao et al. "Accurate Reconstruction of Image Stimuli From Human Functional Magnetic Resonance Imaging Based on the Decoding Model With Capsule Network Architecture". In: *Frontiers in Neuroinformatics* 12 (2018), p. 62.

[16] D. D. Cox. "Do we understand high-level vision?" In: *Current Opinion in Neurobiology* 25 (2014). Theoretical and computational neuroscience, pp. 187–193.

[17] J. Lafferty, A. Mccallum, and F. Pereira. "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data". In: *Proc ICML* (Jan. 2002).

[18] A. Garcia-Garcia et al. "A Review on Deep Learning Techniques Applied to Semantic Segmentation". In: *TPAMI* (2017).

[19] A. Ng. *Deep Learning and Unsupervised feature learning*. Lecture Notes. Stanford University, 2011.

[20] J. Long, E. Shelhamer, and T. Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: *CoRR* abs/1411.4038 ().

[21] T. Lindeberg. "Detecting salient blob-like image structures and their scales with a scale-space primal sketch: A method for focus-of-attention". In: *International Journal of Computer Vision* 11.3 (Dec. 1993), pp. 283–318.

[22] O. Ronneberger, P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *arXiv e-prints*, arXiv:1505.04597 (May 2015), arXiv:1505.04597. arXiv: 1505.04597 [cs.CV].

[23] V. Badrinarayanan, A. Kendall, and R. Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation". In: *arXiv e-prints* (Nov. 2015).

[24] J. Fu et al. "Stacked Deconvolutional Network for Semantic Segmentation". In: *arXiv e-prints* (Aug. 2017).

[25] R. Campanini et al. "A novel featureless approach to mass detection in digital mammograms based on support vector machines". In: *Physics in medicine and biology* 49 (Apr. 2004), pp. 961–75. DOI: 10.1088/0031-9155/49/6/007.

[26] M. Roffilli. "Advanced Machine Learning Techniques for Digital Mammography". PhD thesis. Tech. Rep. UBLCS-2006-12, University of Bologna, Mar. 2006. URL: http://www.cs.unibo.it/people/phd-students/roffilli/.

[27] R. LaLonde and U. Bagci. "Capsules for Object Segmentation". In: *arXiv e-prints*, arXiv:1804.04241 (Apr. 2018), arXiv:1804.04241. arXiv: 1804.04241 [stat.ML].

[28] T. Sun et al. "Trace-back Along Capsules and Its Application on Semantic Segmentation". In: *arXiv e-prints* (Jan. 2019). arXiv: 1901.02920 [cs.CV].

[29] L. Silvestri et al. "Quantitative neuroanatomy of all Purkinje cells with light sheet microscopy and high-throughput image analysis". In: *Frontiers in Neuroanatomy* (2015).

[30] P. Frasconi et al. "Large-scale automated identification of mouse brain cells in confocal light sheet microscopy images". In: *Bioinformatics* 30.17 (2014), pp. i587–i593.

[31] G. Mazzamuto et al. "Automatic Segmentation of Neurons in 3D Samples of Human Brain Cortex". In: Mar. 2018, pp. 78–85.

[32] *Neurons, Synapses, Action Potentials, and Neurotransmission*. 2008. URL: http://www.mind.ilstu.edu/curriculum/neurons_intro/neurons_intro.php.

[33] M. Alegro et al. "Automating cell detection and classification in human brain fluorescent microscopy images using dictionary learning and sparse coding". In: *Journal of Neuroscience Methods* 282 (2017), pp. 20–33.

[34] *Human Brain Project - Specific Grant Agreement 2*. URL: http://lens.unifi.it/bio/research-projects/.

[35] *Aliquis*. 2016. URL: http://www.bioretics.com/aliquis.

[36] *The MNIST database of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/.

[37] *TensorFlow Guide*. URL: https://www.tensorflow.org/guide/.

[38] P. Kingma and J. L. Ba. "Adam: a method for stochastic optimization". In: ed. by ICLR. 2015.

[39] *CapsNet-Keras*. URL: https://github.com/XifengGuo/CapsNet-Keras.