UNIVESITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Magistrale in Ingegneria delle Telecomunicazioni

Deep learning techniques applied in computer vision

Relatore:
Prof.
Chiuso Alessandro

Presentata da:
Corsale Federico

Anno Accademico 2016/2017

Abstract

Visual representations are defined in terms of minimal sufficient statistics of visual data, for a class of tasks, that are also invariant to nuisance variability. In previous works analytical expressions for such representations have been derived and they have been related to feature descriptors commonly used in computer vision, as well to convolutional neural networks. In this document we want to verify the relationship between the visual representations and the convolutional neural networks. In order to accomplish this task we adopted a practical approach: we implemented a CNN and we ran several simulations with different configurations. Using the accuracy rating as measurement, we have been able to evaluate whether or not our changes were beneficial.

Failure is an option here. If things are not failing, you are not innovating enough.

- Elon Musk

Contents

Chapter 1

Introduction

1.1 CNN and visual representations

Convolutional neural networks (CNN) are a class of models vastly used in computer vision, speech recognition, data mining, statistics etc. This models have been inspired by the biological neural networks and in particular by the visual cortex. In fact, experiments on animals shown that individual neurons respond to stimuli in a restricted region of space, called receptive field, and that these neurons have an hierarchical structure.

Convolutional neural networks have some analogies with the visual cortex. In particular, if we design a structure that takes an image as input and convolves it with a filter we can obtain another image, usually called feature map, which pixels can be thought as the neurons in the biological networks. In this case the receptive field can be associated to the size of the filters and the stimuli intensities can be associated with the weights of the filters. Using different filters we obtain several representation of the input image, each of which highlights a feature based on the filter used. Stacking up different layers (set of filters) that take as inputs features maps, creates the hierarchical structure known as convolutional neural network.

Finding the weights to make this model useful, is something related to the science of learning. This subject studies methods that let machines learn from data without being explicitly programmed. For neural networks, the back-propagation algorithm has been developed. Without entering too much in detail now, we can tell that the back-propagation algorithm optimize the weights of the CNN by trying to reduce a defined cost function. This fact causes that the weights may assume arbitrary

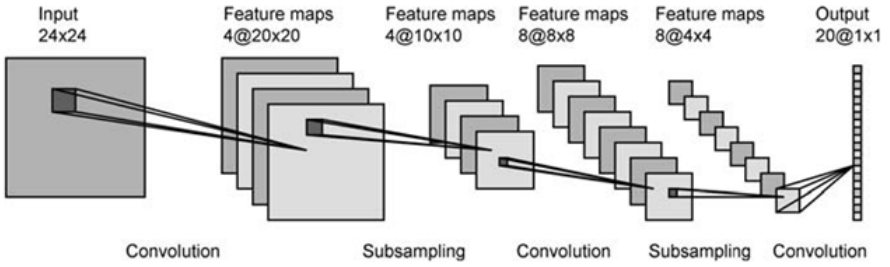


Figure 1.1: Representation of a convolutional neural network

values that in general might have no interpretations. Studies such as [1] tried to explain the meaning of the filters in a CNN, but they haven't derived analytical expressions to give an interpretation to them. So, in general, neural networks are seen as black-boxes that once trained accomplish particular tasks.

In order to optimize the training, several techniques has been proposed over the last few years. The most of the methods proposed in literature aim to counteract some weaknesses of this architecture, but none of them plays around the interpretations of the parameters of the network. Examples of such techniques are Batch Normalization which tries to reduce internal covariance shift, Dropout which tries to reduce the overfitting, gradient clipping which bounds the gradient in the back-propagation algorithm etc.
A recent study made by Soatto and Chiuso [15] derived analytical expression for visual representations which are functions of visual data that are useful to accomplish visual tasks. In their work they also related such representations to feature descriptors commonly used in computer vision, as well to convolutional neural networks. In particular they show that local descriptors can be implemented via linear convolutions and rectified liner units and provided an approximation of the SA likelihood that can be implemented by convolutional neural networks.

1.2 Our contribution

Our contribution has been experimenting whether or not the aforementioned approximation can be useful for training convolutional neural networks. In particular we want to exploit the knowledge about visual representations in order to constrain some parameters of the network. While in the one and we risk to lose learning capabilities (because the network

has less freedom for its weights), in the other hand we may "guide" the network in the right direction during the training.
In order to evaluate if these changes were beneficial we adopted a practical approach: we implemented a convolutional neural network, we computed the accuracy rating with different configurations and we compared the results with the accuracy rating of a traditional CNN.

1.3 Structure

In the next chapter we will introduce machine learning, neural networks and convolutional neural networks. In chapter 3 we will introduce visual representation and illustrate the setup used to perform the simulations. In chapter 4 we will present the results obtained from our simulations. In the last chapter we will draw conclusions about the experiment.

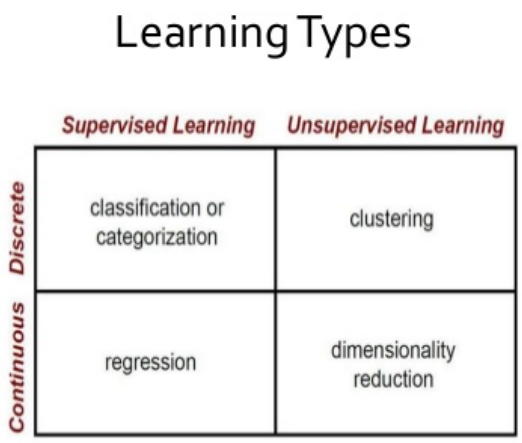
Chapter 2

Neural networks

2.1 Machine learning

The science of learning plays a key role in several fields such as statistics, data mining, computer vision, adaptive filtering, speech recognition etc. This science studies methods that let the machines to learn from data without being explicitly programmed. Examples of such tasks may be predicting whether a patient will have an heart attack or identify the risk factors for prostate cancer based on clinical and demographic variables.

In a typical scenario we have an outcome measurement, usually quantitative (such as a stock price) or categorical (such as heart attack/no heart attack), that we wish to predict based on a set of features (such as diet and clinical measurements).
In general, in machine learning, we use the letter x to indicate the features and the letter y to indicate the outcome. For example, if we want to estimate the price of an house, we expect x to be the set of features (such as the number of bedroom, the number of bathroom, the number of floors, having garage or not etc.) and y to be the price.
Since machine learning means learning from data, we need a *training set* (which is a collection of available data) to build a prediction model. A training set may be constituted by a collection of (x_i, y_i) or by a collection of x_i only. In the first case we speak about *supervised learning* while in the second case we speak about *unsupervised learning*. We denote with X_{tr} and Y_{tr} the collection of x_i and y_i respectively in the training set.
There are different types of learning. A common classification is based on what we know and the type of the data we want to predict. Here's a table that shows the different types of learning.



In this document we will see a realization of a classifier, i.e. a model that learning from a training set X_{tr} will be able to make predictions on new data by associating to a set of features x_{new} a discrete value y_{new} often referred as label.

When we build a model, usually we need different datasets: the training, the validation and the test one.

Training dataset

This dataset is required in order to train a given model i.e. optimizing its parameters in order to fit the training data well. Usually we train different models with this dataset and later we chose the one that works best.

Validation dataset

This dataset is required in order to evaluate the goodness of the models. After training them, we make predictions on this data and we chose the model that perform best. However, since the data in the validation set has been used to select the model, we can expect that the error computed is lower than the error on new data. For this reason we need to evaluate the performance of the model using another dataset.

Test dataset

This set is used to evaluate the performance of the model. Differently from the validation set this isn't used to chose the model, but just as a

tool to measure its goodness.

In practice we have just the "available data". In order to train the model we usually split them in these three datasets and subsequently we perform the training. There are no rules that defines how the available data should be split between training, validation and test sets, but an example might be 50%, 25%, 25%. Even the division of the available data may have impacts on the performance of the model. For this reason techniques such as cross-validation or bootstrap are commonly used [21].

In real world applications usually we have at disposal a large amount of data. Among the features we collected in general there might be some that are useless and we would like that our model won't account for them. For example, if our classifier wants to predict whether a patient will have an heart attack, the feature "patient's favourite color" is probably useless and we would like our model to not use it.
The process that aims to discard poor features is called regularization. Beside the computational aspect (using a large amount of features in our model increases the computational complexity), selecting the right features is also beneficial when we will use the model to predict new data. In fact, having a model that is overparametrized, lead to the phenomena called overfitting. The concept is that if we have too many parameters compared to the number of data available, we are able to fit the training data "too well" without being able to capture the real trend of the data.

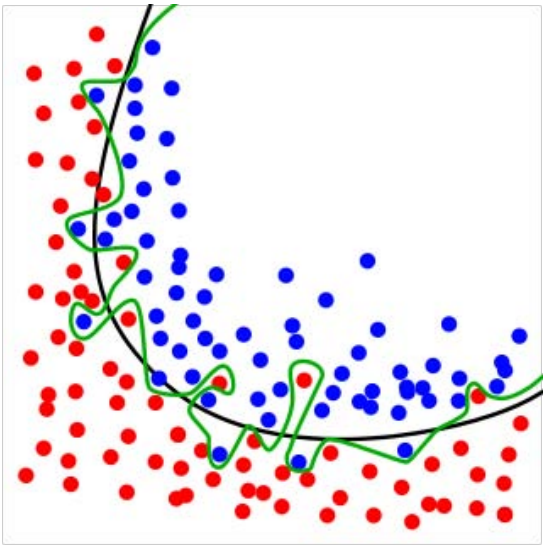


Figure 2.1: Overfitting.

For example, in Figure 2.1 it is possible to see that the green line fit

the training data perfectly, but we can expect that it will perform worse than the black line on new data. In order to counteract overfitting, regularization techniques such as Ridge regression, the LASSO, Dropout etc [10] has been developed.

In literature there exists several models that can be used as classifiers, the one we will use is a neural network.

2.2 Neural networks

Neural networks are a class of models that have been subject of several studies. Their importance is caused by their large usage is different fields and applications. Nowadays neural networks are used for function approximation, classification tasks, data processing, robotics etc [2].

But what Neural Networks actually are? The term *Neural Network* has evolved over time and encompass a large class of models and learning methods. The term itself has an historical reason. In fact, initially, they were created to model the human brian. The different units of a neural network should represents the neurons, while the connections between them should represent the synapses.
In order to understand what they are and how they works, here we describe the simplest of the models i.e. the one used for regression or classification.

A Neural Network is a two-stage regression or classification model. For regression, typically $K = 1$ and there is only one output unit Y_1 at the top. For K-class classification, there are K units at the top, with the k-th unit modeling the probability of class k. There are K target measurements Y_k, k=1, ..., K, each being coded as a 0-1 variable for the k-th class.
Derived features Z_M are created from linear combinations of the inputs, and then the target Y_k is modeled as a function of linear combinations of the Z_m,

$$\begin{aligned} Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X) \qquad m = 1, ..., M, \\ T_k &= \beta_{0k} + \beta_k^T Z \qquad k = 1, ..., K, \\ f_k(X) &= g_k(T) \qquad k = 1, ..., K, \end{aligned} \tag{2.1}$$

where $Z = (Z_1, Z_2, ..., Z_M)$, and T=$(T_1, T_2, ..., T_K)$.

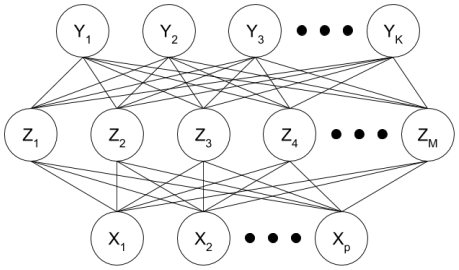
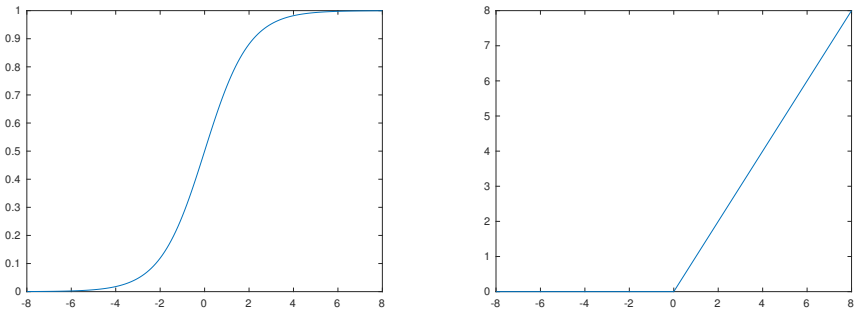


Figure 2.2: Representation of Neural Network

The activation function $\sigma(v)$ is usually chosen to be the *sigmoid* defined as:

$$\sigma(v) = \frac{1}{1 + e^{-v}} \tag{2.2}$$

even if other functions such as the Rectified Linear Unit ($ReLU$) has been also adopted recently for both performance and computational reasons. The activation function $\sigma(v)$ is necessary because it is the element that let the neural network to learn non-linear function. In fact, should the $\sigma(v)$ be equal to the identity function, the entire model would collapse to a linear model in the inputs.



(a) Sigmoid.

(b) ReLU.

Figure 2.3: Activation functions.

The output function $g_k(T)$ allows a final transformation of the vector of outputs T. For regression we typically choose the identity function $g_k(T) = T_k$. Early work in K-class classification also used the identity function, but this was later abandoned in favour of the $softmax$ function defined as:

$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}} \tag{2.3}$$

This is the same transformation used in multilogit model and produces positive estimates that sum to one.

The units in the middle of the network are called *hidden units* because the values Z_m are not directly observed. In general there can be more than one hidden layer and actually the most recent neural networks stacks several of them. The Z_m can be interpreted as a basis expansion of the original input X. The neural network is then a standard linear model using these transformation as inputs [3].

Neural networks have been proved to be universal approximators. In particular the universal approximation theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of R^n, under mild assumptions on the activation functions. This implies that simple neural networks can represent a wide variety of functions when given appropriate parameters.
The first version of the theorem has been proven by George Cybenko in 1989 for sigmoid function, but in 1991 Kurt Hornik has proven that it is not the specific choice of the activation function, but rather the multilayered feed-forward architecture itself which gives neural networks the potential of being universal approximators [22].

2.3 Training neural networks

The neural network model has unknown parameters, usually called *weights*. The purpose of the training is to find values for them such that the model fit the training data well. We denote the set of weights as θ, which consists of

$$\begin{aligned} &\{\alpha_{0m}, \alpha_m; m = 1, 2, ..., M\} \qquad M(p+1)\ weights, \\ &\{\beta_{0k}, \beta_k; k = 1, 2, ..., K\} \qquad K(M+1)\ weights. \end{aligned} \tag{2.4}$$

In order to measure how well the model fit the data, we need an error function. For classification task, both sum-of-squared errors or cross entropy are used. In our model we will use the cross-entropy defined as:

$$R(\theta) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\, log\, f_k(x_i; \theta) \tag{2.5}$$

Where $y_{ik} \in \{0, 1\}$ since y_i has been coded as an one hot vector.
The classifier is $G(x) = argmax_k\, f_k(x)$.
The generic approach to minimize $R(\theta)$ is by gradient descent, called $back - propagation$ in this settings.

2.3.1 Back-propagation algorithm

The back-propagation algorithm consists of two phases: propagation and weights update. In the first phase, an input is presented to the network and it is propagated, layer by layer, until it reaches the output layer. The output of the network is then compared with the desired output. Using the cost function we are able to evaluate a loss and use it in order to improve the weights of our network. In particular the computed errors in the output layer are propagated backward until each neuron has an associated error which represents its contribution to the original output. Using these errors, we are able to calculate the gradient of the loss function with respect to the weights of the network.
In the second phase, this gradient is fed to the optimization method, which uses it to update the weights, in attempt to minimize the loss function.

Here is an example of the back-propagation algorithm used when the activation function σ is the *sigmoid* and the loss function is the squared error function defined as

$$E = \frac{1}{2}(t - y)^2 \tag{2.6}$$

where t is the target output for a training sample and y is the actual output of the output neuron. The term $\frac{1}{2}$ is used to cancel the exponent when differentiating.
For each neuron j, its output o_j is defined as

$$o_j = \sigma(net_j) = \sigma(\sum_{k=1}^{n} w_{kj} o_k) \tag{2.7}$$

where n is the number of input units to the neuron j, w_{ij} denotes the weight between neurons i and j and lastly net_j is the weighted sum of outputs o_k of the previous neurons. If the neuron is in the first layer after the input layer, the o_k of the input layer are simply the inputs x_k to the network.

In order to find the derivative of the error with respect to a weight w_{ij}, we apply the chain rule twice obtaining the following relation:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}} \tag{2.8}$$

In the last factor of the right-hand side of the above, only one term in the sum net_j depends on w_{ij}, so that

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}(\sum_{k=1}^{n} w_{kj} o_k) = o_i \tag{2.9}$$

If the neuron is in the first layer after the input layer, o_i is just x_i. The derivative of the output of neuron j with respect to its input is simply the partial derivative of the activation function. Since we use the *sigmoid*, which has the derivative

$$\frac{\partial \sigma}{\partial v}(v) = \sigma(v)(1 - \sigma(v)) \tag{2.10}$$

we obtain

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial}{\partial net_j}\sigma(net_j) = \sigma(net_j)(1 - \sigma(net_j)) \tag{2.11}$$

The first factor is straightforward to evaluate if the neuron is the output layer, because then $o_j = y$ and

$$\frac{\partial E}{\partial o_j} = \frac{\partial E}{\partial y} = \frac{\partial}{\partial y}\frac{1}{2}(y - y)^2 = y - t \tag{2.12}$$

However, if j is in an arbitrary inner layer of the network, finding the derivative E with respect to o_j is less obvious.
Considering E as a function of the inputs of all neurons $L = u, v, ..., w$ receiving input from neuron j

$$\frac{\partial E(o_j)}{\partial o_j} = \frac{\partial E(net_u, net_v, ..., net_w)}{\partial o_j} \tag{2.13}$$

and taking the total derivative with respect to o_j, a recursive expression for the derivative is obtained:

$$\frac{\partial E}{\partial o_j} = \sum_{l \in L}(\frac{\partial E}{\partial net_l}\frac{\partial net_l}{\partial o_j}) = \sum_{l \in L}(\frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial net_l} w_{jl}) \tag{2.14}$$

Therefore, the derivative with respect to o_j can be calculated if all the derivatives with respect to the outputs o_l of the next layer (the one closer to the output neuron) are known.
Summarizing we have

$$\frac{\partial E}{\partial w_{ij}} = \delta_j o_i \tag{2.15}$$

with

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j} = \begin{cases} (o_j - t_j) o_j (1 - o_j) & \text{if } j \text{ is an output neuron} \\ (\sum_{l \in L} \delta_l w_{jl}) o_j (1 - o_j) & \text{if } j \text{ is an inner neuron} \end{cases} \tag{2.16}$$

To update the weight w_{ij} using gradient descent, one must choose a learning rate α. The change in weight, which is added to the old weight, is equal to the product of the learning rate and the gradient, multiplied by -1:

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} = \begin{cases} -\alpha o_i (o_j - t_j) o_j (1 - o_j) & \text{if } j \text{ is an output neuron} \\ -\alpha o_i (\sum_{l \in L} \delta_l w_{jl}) o_j (1 - o_j) & \text{if } j \text{ is an inner neuron} \end{cases} \tag{2.17}$$

The term -1 is needed in order to update in the direction of a minimum of the error function.

2.3.2 Modes of learning

We have just explained how back-propagation algorithm works, and in particular that the propagation and the update phases are repeated cyclically. However we haven't told anything about how often this cycles repeat themselves. There are mainly two different ways in which we can train our network: stochastic and batch. The first method foresees to repeat the propagation-update cycle every time we provide a new input to the network. This method has the advantage to reduce the probability of the network getting stuck in a local minima, but has the disadvantage that the network becomes very slow to be trained. On the other hand, batch learning foresees to propagate several inputs and accumulate errors before updating the weights. This second method yields a faster and more stable descent to a local minima since the update is performed in the direction of the average error of the batch samples. In the most recent applications an hybrid mode is used, and the so called $mini - batch$ are used to train the network [4]. The mini-batches are constituted by m samples chosen randomly among the data in the training set.

One of the major strengths of the back-propagation algorithm is its simple and local nature. In fact, each hidden unit passes and receive information only to and from units that share a connection. So, from a computational point of view, we have the advantage that this algorithm can be implemented efficiently on a parallel architecture computer.
On the other hand, back-propagation presents also some disadvantages. In particular the update performed by the gradient descent depends on the learning rate α. Such parameter must be chosen carefully because it dictates how much the weights are changed at every iteration. Using a value of α to big may cause too strong variations causing the minimum

to be missed [5]. On the other side, a value of α that is too small slows the training unnecessarily and may lead to a local minimum that is not optimal. In order to counteract this kind problems, adaptive algorithms such as the Adam optimizer has been deployed.
Another drawback related to the back-propagation algorithm is the so called *vanishing gradient problem* [6]. This phenomena is caused by the fact that weights receives updates proportional to the gradient of the error function with respect to the current weight in each iteration of training. Traditional activation functions such as the sigmoid or the hyperbolic tangent have gradients in the range (-1, 1), and back-propagation computes gradients by the chain rule. This has the effect of multiplying n of those small numbers to compute gradients of the lower layers in an n-layer network, causing that the gradient decreases exponentially with n and the lower layers train very slowly [9]. To overcome this problem some solution has been proposed depending on the network trained: for standard neural networks, we can train one level at a time through unsupervised learning and later tune it with back-propagation, for Recurrent Neural Networks (RNN) the method called *long short term memory* (LSTM) has been vastly adopted etc.
For the same reason vanishing gradient problem exists, also the *exploding gradient problem* exists when activation functions with larger derivatives are adopted. A way to counteract this problem is using the method called *gradient clipping* which clips the gradients to prevent them from getting to large.

Like others models, also neural networks uses different dataset for training. In fact, beside the training set used to tune the parameters of the model, the validation set is used to evaluate the optimal number of layers, the filter size etc.

2.4 CNN

In the previous sections we have introduced neural networks and we have explained how they are trained. From what we have seen it is clear that the amount of weights to manage is pretty huge because each neuron is connected to every other in the previous and following layer.
Such a structure has several downsides. In particular, once we start to increase the number of layers of the network, the number of weights becomes larger and larger to the point that it is very inefficient to be managed. Another problem related to this huge amount of weights is

the overfitting. In fact, having such a number of parameters that can be learned, means that if our training set isn't big enough the network may learn to perfectly discriminate data of our training set, but may have poor performance on new data. Lastly, having such an interconnected structure means that we lose the capability of extract local features.

Such kind of problems has been made easier to tackle in *Convolutional Neural Networks* (*CNN*). Differently from regular neural networks, here the neurons are connected just to a subset of the outputs of the previous layer.

Historically this kind of structure has been inspired by observing the animal visual cortex. In fact, individual cortical neurons respond to stimuli in a restricted region of space called *receptive field*. Receptive fields of different neurons partially overlap such that they tile the visual field. In the CNN we approximate the response of an individual neuron to stimuli within its receptive field with a convolution operation [7].
The convolution is made between the input of a given layer and one or more *filters* (also called *kernels*). The output of the convolution operation is called *feature map* (or *activation map*). There are as many feature maps as the number of filters we use to process the input data. In Figure 2.4 it is shown an example of a convolutional layer with two filters.

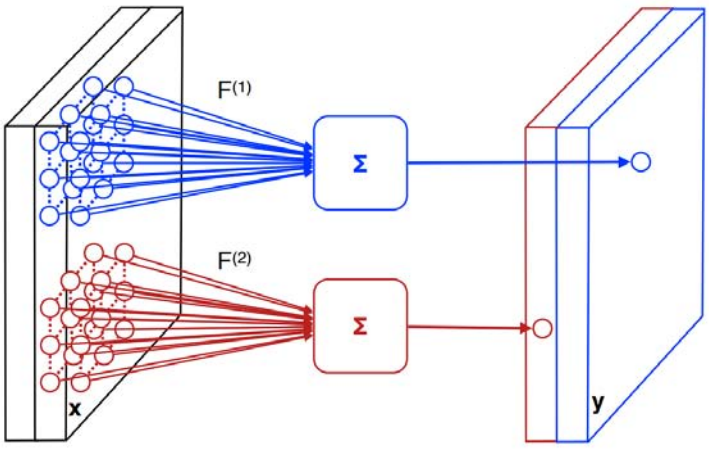
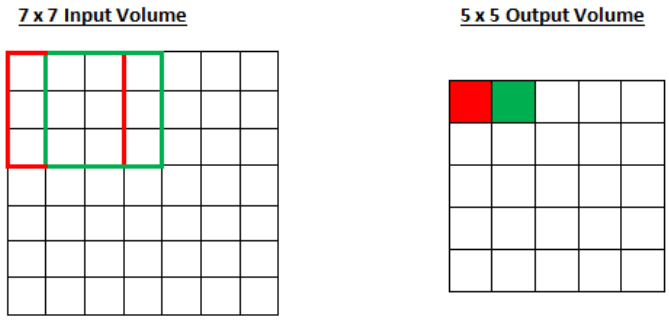


Figure 2.4: Representation of convolutional layer

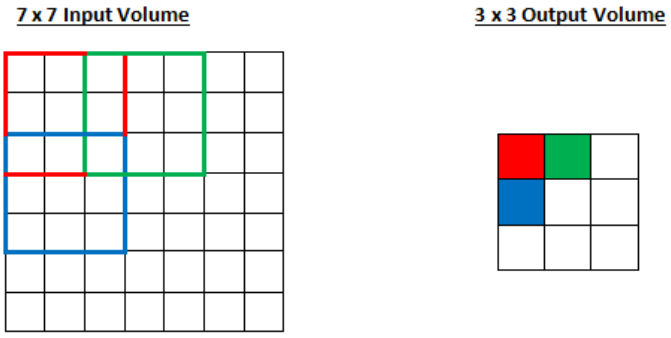
This convolutional structure is pretty important: beside keeping the number of parameter smaller, it also provides the translation invariance property. In fact, being able to use the same filter all over the image

means that we are able to detect a feature wherever it is. Should the image being translated, or the object represented moved, we will still be able to detect the feature.

The way the convolution is performed is part of the CNN design. In fact, the *size* of the kernels, the *stride* and the *padding* of the convolution are parameters chosen by the CNN creator.
The size of the kernels impacts both on performance and computational complexity. The dimension is usually chosen empirically even if some tricks can be taken into account. For example using two 3x3 filters is the same as using one 5x5 filter, but the number of weights to be stored is smaller.
Stride controls how the filter convolves around the input volume. For example, a stride set to 1 means that the filter convolves around the input volume by shifting one unit at a time. In Figure 2.5 is shown an example of a convolution with two different values of stride.



(a) Stride set to 1



(b) Stride set to 2

Figure 2.5: Representation of strides

Zero-padding, instead, is usually used in order to preserve the dimension of the input volume. As shown in Figure 2.5 convolving a given volume with a filter yields an output volume smaller then the input one. Sometimes this is inconvenient and we would prefer to have the output

volume of the same size of the input one. Padding allows achieving this result by expanding with zeros the input volume so that the convolution can be performed even on the edges.

In general the dimension of the output volume is given by the following formula:

$$O = \frac{W - K + 2P}{S} + 1 \tag{2.18}$$

Where O is the output height/width, W is the input height/width, K is the filter size, P is the padding and S is the stride.

But what these filters actually represents? Filters can be thought of as feature identifiers. Assuming the input of our CNN is a set of images, these filters may be things such as edges, simple colors, corners, curves etc. The first layer of the CNN usually identifies simple structures such as the ones mentioned before. Higher level filters, instead, identifies much more complex features [1].

2.4.1 CNN structure

In traditional CNN there are other layers that are interspersed between convolutional layers. These other layers provide non-linearities and preservation of dimension that help to improve the robustness of the network and control overfitting [8]. Examples of such layers are activation functions, pooling layers, fully connected layers and dropout layers.

Activation functions

Similarly to the regular neural networks, also in the CNN we use activation functions. These are usually placed just after the convolutional layer. Analogously to the neural networks, also in this case they are needed in order to let the network learning non-linear functions and to provide a sort of thresholding.

Pooling layers

After placing the activation function, it is often used a pooling layer. There are different options to perform the pooling, for example max-pooling or L2-norm pooling. The concept of the pooling layer is to take a filter (usually of size 2x2), a stride of the same length and apply this to the input volume. The output will be a volume that holds the maximum

number (or the L2-norm) of every subregion the filter has been applied to. The intuitive reasoning behind this layer is that once we know that a specific feature is in the original input volume (there will be a high activation value), its exact location is not as important as its relative location to the other features. This layer is useful for two reasons: the first is that it reduces the number of parameters by 75% and the second is that it will control overfitting.

Fully connected layer

This layer is usually placed at the top of the CNN. It is needed because once we have a set of high-level features, we need a function that is able to evaluate which features most correlate to a particular class. Often there are used a couple of fully connected layer at the top of the CNNs, one used to combine the features of the last convolutional layer and the other used to predict the class.

Dropout layer

Dropout layer has the only purpose of fight overfitting. This technique consists of setting to zero the output of each hidden neuron with probability 0.5. The neurons which are dropped out in this way do not contribute in the forward pass and do not participate in back-propagation. So every time an input is presented, the neural network samples a different architecture, but all architectures share weights. This technique reduces complex co-adaption of neurons since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons [10].

2.5 Common issues

In section 2.3 we explained how neural networks are trained. However, applying the back-propagation algorithm is not enough and some common issues must be addressed in order to successfully train the neural network.

2.5.1 Initializing weights

When we create a CNN, we have to initialize its weights. This process requires a bit of attention. Using weights equals to zero leads to zero derivatives and the algorithm never moves. Using weights that are too

large, instead, often lead to poor solutions. So, usually, neural networks weights are initialized to be random values near zero. Initializing weights in this way means that at the start of the training the whole neural network is approximately a linear model because the operative part of the activation function $\sigma(v)$ is roughly linear. The network introduces non-linearities where needed as the training goes on.

2.5.2 Overfitting

As mentioned previously neural networks often have too many weights and they tend to overfit the data at the global minimum of R. In early developments of the neural networks an early stopping rule was adopted in order to prevent this kind of behaviour. The training was performed for a while and was arrested well before the global minimum was approached. In order to apply such technique a validation dataset is required for determining when to stop, since we expect the validation error to start increasing.
Another method used to tackle the overfitting problem is *weight decay* which is analogous to Ridge regression used for linear models. Adding a penalty $\lambda J(\theta)$ to the error function and estimating λ via cross-validation is however pretty computational expensive and other methods such as Dropout has been adopted recently.

2.5.3 Scaling of the inputs

The scaling of the inputs determines the effective scaling of the weights in the bottom layer. For this reason standardize all inputs to have mean equal to zero and standard deviation equal to one may have large effect on the quality of the solution.

2.5.4 Internal covariate shift

During the training process, the distribution of each layer's input changes as the parameters of the previous layers change. This phenomena, known as *internal covariate shift* [11], slows down the training by requiring lower learning rates and careful weights initialization. It also makes it hard to train models with saturating non-linearities. In order to deal with this problem, a method called Batch Normalization has been proposed. The core idea of this method is to normalize the input of each layer using the statistic of the mini-batch $\mathcal{B} = \{x_{1...m}\}$ and learning two parameters γ and β to maintain the representation power of the network.

So, for every layer, the input are scaled in this way:

$$\begin{aligned}
\mu_{\mathcal{B}} &= \frac{1}{m}\sum_{i=1}^{m} x_i \\
\sigma^2_{\mathcal{B}} &= \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \\
\hat{x}_i &= \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \\
y_i &= \gamma\hat{x}_i + \beta = BN_{\gamma,\beta}(x_i)
\end{aligned} \tag{2.19}$$

Where ϵ is a small value used to avoid division by zero. During the training phase it is also required to store the statistic of the training set so that it can be used when it is necessary to process new data.
Batch-Normalization can also act as a regularizer and in some cases eliminates the need for Dropout [11].

2.5.5 Number of hidden layer

Tuning the number of hidden layers has large impact on the performance of the network. In fact, using too few hidden units, may cause the model to not have enough flexibility to capture non-linearities in the data. So, in general, it's preferable to use a large number of layers. In this last case extra weights can be shrunk toward zero if an appropriate regularization technique is adopted.
There are not firm rules about the exact number of layers that should be used. Typically this number ranges from 5 to 100 with the number increasing with the number of inputs and the number of training cases. Usually the choice is guided by background knowledge and experimentation. Over the last few years we evolved from network such as AlexNet [10] that uses around 10 layers to networks such as Microsoft ResNet [12] that uses 152 layers.

2.5.6 Multiple minima

The error function $R(\theta)$ is nonconvex, possessing many local minima. As a result, the final solution depends on the starting values of the weights. In order to perform a good training, it is necessary to perform different trainings starting from different configurations and choosing the solution giving the lowest error.

2.5.7 Small training set

As mentioned previously, neural networks tend to be overparametrized. Hence, having large training set is beneficial in order to prevent overfitting. Sometimes, however, the available data isn't big enough and some data augmentation techniques should be adopted. Data augmentation techniques are procedures that let us enlarge the database using transformations of the original data. For example, if our dataset is composed by images, we can enlarge it by applying horizontal flips, vertical flips, random crops, color jitters, rotations etc. while keeping the label unchanged.

2.6 Evolution of neural networks

Over the last few years neural networks are evolved significantly. In this section we want to illustrate how the building blocks discussed previously have been used in order to achieve interesting result in computer vision.

2.6.1 AlexNet

One of the milestone in computer vision is the AlexNet [10]. His network was able to achieve impressive result in the ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) marking a top 5 test error of 15.4% and beating the previous record of 26.6%.
The network was made up of five convolutional layers, max-pooling layers, dropout layers and three fully connected layers. Even if the architecture is pretty simple, it is important because it was the first time that CNN was used to achieve such impressing score on the ImageNet dataset.

2.6.2 VGG Net

VGG Net was a model created in 2014 [13]. This network was composed by 19 convolutional layers that strictly used 3x3 filters with stride and pad of 1, along with 2x2 max-pooling layers with stride 2. Even if also this architecture was pretty simple, it illustrated that going deep with neural networks is beneficial in order to create an hierarchical representation of the data.

2.6.3 GoogLeNet

GoogLeNet is one of the first models that introduced the idea that CNN layers doesn't always have to be stacked up sequentially. In particular

the authors proposed the inception module that is a block that perform different computations in parallel. The result of these computations is then concatenated in order to produce the input to the next layer.

In a traditional CNN we need to make a choice of whether to have a pooling operation or a convolutional operation. Instead, the idea behind the inception module is that we can perform all this operation in parallel while remaining computationally manageable. The tool that keeps the complexity under control is the block that performs the 1x1 convolution. This layer outputs a volume with the same height and width of the input volume but with a depth arbitrary high and equal to the number of 1x1 filters used. This layer can be though as a "pooling of features" because we are reducing the depth of the volume similarly to how we reduce the dimensions of height and width with pooling layers.

In Figure 2.6 it is shown the GoogLeNet architecture and in Figure 2.7 the inception module (the highlighted area in Figure 2.6).

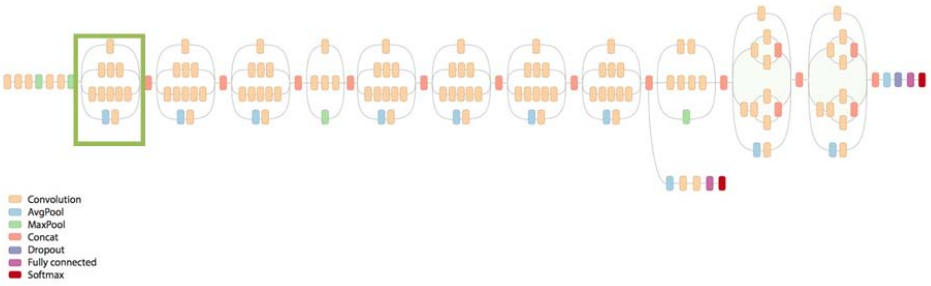


Figure 2.6: GoogLeNet

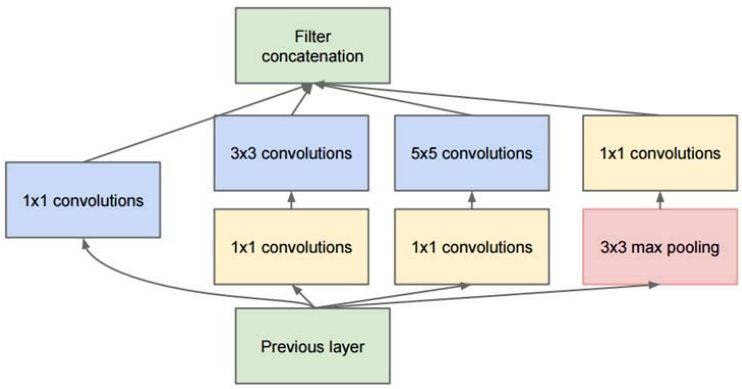


Figure 2.7: Inception module

2.6.4 Microsoft ResNet

Microsoft ResNet [12] brought back the idea of going deeper. With its 152 layer it has established the new record in the ILSVRC in 2015 with an error rate of 3.6%. The main innovation of this network is given by the Residual Block. The idea behind this block is to add the input x after each Convolutional layer-ReLU-Convulutional layer-ReLU series as shown in Figure 2.8.

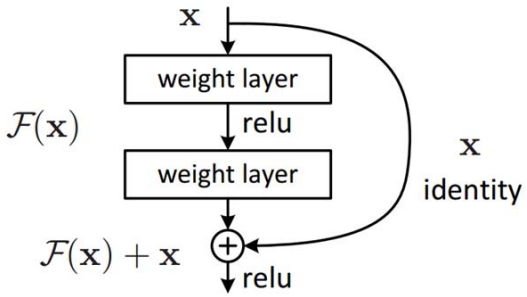


Figure 2.8: Residual Block

Chapter 3

Visual representations

Visual representation are defined in terms of minimal sufficient statistics of visual data, for class of tasks, that are also invariant to nuisance variability. Soatto and Chiuso derived analytical expressions for visual representations and have shown how these are related to feature descriptors commonly used in computer vision and to convolutional neural networks [15].
Our purpose was to progress that work and in particular we focused on deep convolutional architectures. We implemented and studied a CNN and subsequently we chanced parts of it in order to understand if the derived expressions behave as expected.

3.1 Visual representations

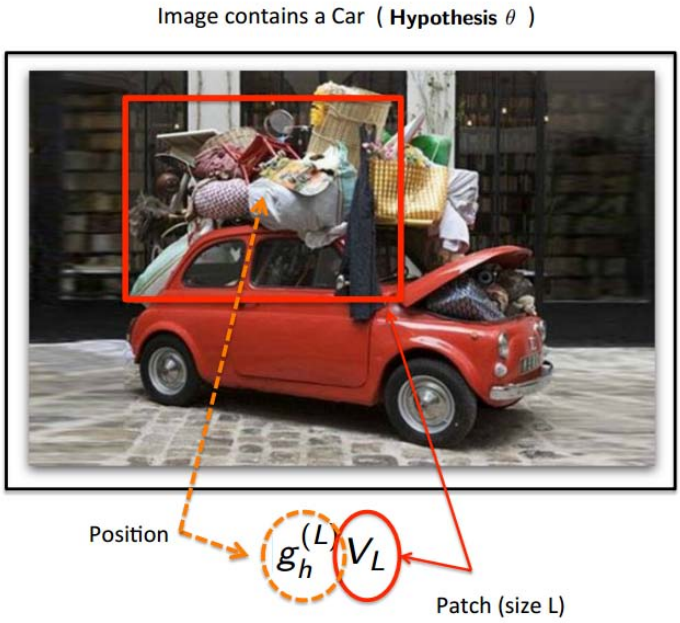
By definition a visual representation is a function $\phi(y)$ of the data y which is useful to a task. We would like representations to have some properties such as sufficiency (they are no less informative than the data), being simpler than the data itself (ideally minimal) and possibly invariant to the effect of nuisance variables. A representation that has these properties is called optimal.

The sampled anti-aliased likelihood (SAL) has been proven to be an optimal representation [15] and an hierarchical approximation has been proposed.
Starting from the decomposition of the hypothesis θ defined as follows

$$p(y|\theta) \simeq \sum_{h=1}^{N}[\sum_{k=1}^{M} p(y|\theta_k^{(L)}, g_h^{(L)}, V_L, \theta)p(\theta_k^{(L)}, g_h^{(L)}|\theta)] \tag{3.1}$$

where:

- y is an image.
- θ is the hypothesis (e.g. the image contains a car).
- $\theta^{(L)}$ are details added at level V_L and locations $g^{(L)}$. $\theta^{(L)}$ may be discretised $\rightarrow \theta_k^{(L)}, k = 1, ..., M$
- $g^{(L)}$ are locations and can also be discrtised $\rightarrow g_h^{(L)}$ with $h = 1, ..., N$.



and assuming the following equation holds:

$$\begin{aligned} p(y|\theta^{(L)}, g_h^{(L)}, V_L, \theta) &= p(y_{|g_h^{(L)} V_L}|\theta^{(L)}, \theta) \cdot p(y_{|g_h^{(L)} V_L^c}|\theta^{(L)}, \theta) \\ &= (y_{|g_h^{(L)} V_L}|\theta^{(L)}, \theta) \cdot const \end{aligned} \tag{3.2}$$

we obtain:

$$p(y|\theta) \simeq \sum_{h=1}^{N}[\sum_{h=1}^{M} p(y_{|g_h^{(L)} V_L}|\theta^{(L)}, \theta_k^{(L)}) p(\theta_k^{(L)}, g_h^{(L)}|\theta)] \tag{3.3}$$

Where the first term in the sum are the outputs of layer L at locations $g_h^{(L)}$ for hypothesis $\theta_k^{(L)}$ and the second term are mixing probabilities which can be interpreted as local filters at layer L.

The output of Layer L, which is the SAL, is:

$$\begin{aligned}\hat{p}_{\theta,\hat{g}}(y) &= max_{g_i} = \int_G p_\theta(gg_iy)dP(g) \\ &\simeq max_{g_i} \int_G \sum_{h=1}^{N}[\sum_{k=1}^{M} p(gg_iy_{|g_h^{(L)}V_L}|\theta_k^{(L)})p(\theta_k^{(L)}, g_h^{(L)}|\theta)]dP(g)\end{aligned} \tag{3.4}$$

Where the integral over g does spatial averaging and the maximization with respect to g_i implies local invariance.
The decomposition can be iterated for inner layers.

We based our simulations on the equation 3.4. In particular, although the hierarchical structure reminds convolutional neural networks, some constraints are not imposed in the actual realizations. For example the term that can be interpreted as a local filter at layer L is a probability and so it should be non negative, whereas in common implementation that parameter is free.

The purpose of our work is to use the knowledge about visual representation in order to force constraints on the neural network. While on the one hand this may reduce the freedom of some parameters and consequently reduce the learning capabilities, on the other hand it may drive the network in the right direction during the training.

3.2 Tools used

In order to implement the CNN, we used the Python package TensorFlow provided by Google [16]. The major strength of this package is that it makes easy to build and train CNNs. If fact, in order to create a CNN and train it, we need to define just the following entities:

- the operations that must be performed by our neural network, i.e. the model itself. For this scope TensorFlow also provide some utility functions that allow to perform 2D-convolution, max-pooling, softmax etc.

- the input that will be used by the network. For this scope TensorFlow uses objects called placeholders which are variables that aren't changed by the optimizer.

- the loss function. Defined as a function of the expected and the predicted output, this is the target of the optimizer.

- the optimizer. This object has the purpose to compute the derivatives in the back-propagation algorithm and takes care to update the weights. TensorFlow provides different optimizers such as the Gradient Descent or the Adam Optimizer [17].

Once we have defined the aforementioned entities, we can run the script providing the actual values that must be feed to the network. TensorFlow will then create the graph representing the CNN, apply the back-propagation algorithm and change the weights of our model accordingly. It must be noted that these computations are transparent to the developer.

Our code was developed on Linux using Phyton 3.5 (the Anaconda distribution [19]) and Jupyter [18].

3.3 Setup

The core network used on our simulations was composed by two convolutional layers and two fully connected layers. As design parameters we chose to use 5x5 filters, ReLU activation functions and after them a max-pooling layer. As regularizer we chose to use the Dropout and at the top of the network we placed the softmax.
The dimension of the filters and number of layers has just been chosen accordingly to our computational power in order to have simulations that didn't last too long. Probably, a deeper network would have had better performance but for the scope of our work that wasn't really important since the described network has been used just as a reference for other simulations.
In Figure 3.1 the scheme of the aforementioned network is depicted.

As cost function we used the cross-entropy defined as:

$$H(\theta) = -\sum_{i=0}^{K-1} p_i \, log \, q_i \tag{3.5}$$

Where K is the number of the classes, p_i are the targets and q_i are the logits. Since we used mini-batches of $m = 64$ images to train the model, our loss function was actually defined as:

$$L(\theta) = -\frac{1}{m}\sum_{i=0}^{K-1} p_i \, log \, q_i \tag{3.6}$$

As optimizer method we used the Adam Optimizer with a starting learning rate $\alpha = 10^{-4}$.

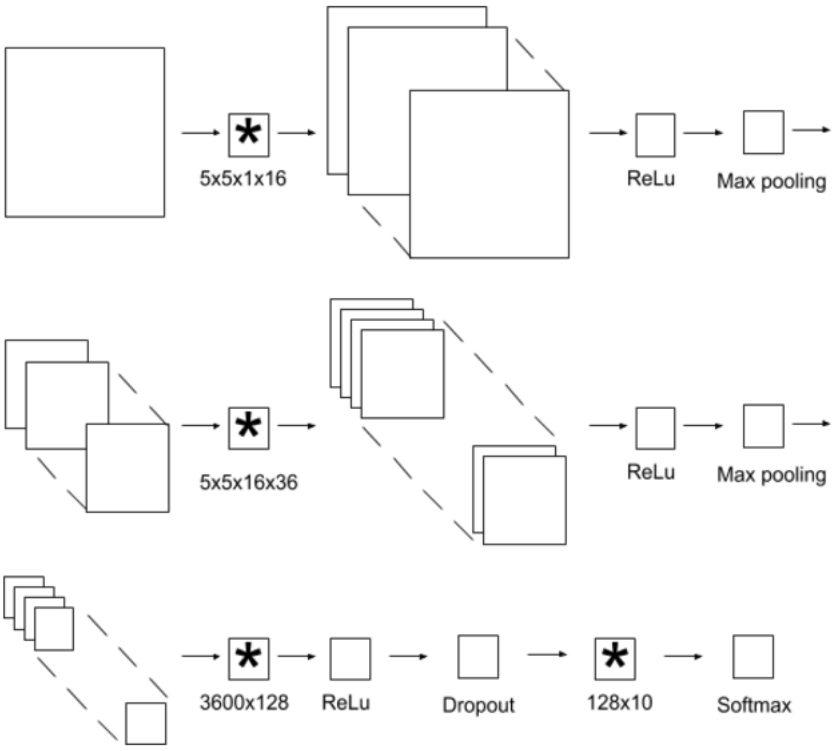


Figure 3.1: Core network

Lastly, as data set, we used the Cluttered MNIST database provided on the daviddao GitHub Repository [20]. It consists of 12000 handwritten digits of size 40x40: 10000 are used for training, 1000 for validation and 1000 for test. Differently from classic MNIST dataset, cluttered MNIST provides some artefacts as shown in Figure 3.2.

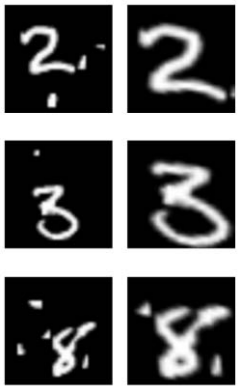


Figure 3.2: Examples of cluttered handwritten digits.

In our simulation we haven't used any data augmentation technique.

3.4 Running the experiments

In our first simulation we used the network described before. The accuracy result obtained in this setup has been used as reference for further simulations to understand whether or not the network was performing better. All simulations have been carried out in the same way but with different configuration of the network. We trained the CNN with mini-batches of $m = 64$ images, we set a cap to the number of epochs equal to 20000 and if the CNN wasn't performing better (on the validation set) for over 2000 iterations then we stopped the training.

In our simulations we mostly focused on the first and last layer on the CNN. In particular we replaced the softmax layer with other normalizations and we normalized the weights of the filters in the first layer. For every configuration we have observed the accuracy, we analysed the weights and we tried to understand their behaviour.
In the next Chapter we illustrate and discuss the results obtained.

3.5 Repositories

The analysis of the CNN isn't completed yet and some other simulations could be performed. In order to let further researches in this field without the need to restart from scratch, we leave the repositories in which the code can be found.

https://github.com/Corsal8/Tensorflow
This repository contains the code used to perform the simulations. It is also possible to find the snapshot of the network so that it isn't necessary to retrain the model in order to visualize weights, output layers, or other kind of data.

https://github.com/Hvass-Labs/TensorFlow-Tutorials
This repository contains a TensorFlow tutorial. Since some code has been used from this repository, we mention this link because contains comments and informations that might be useful to understand also our work.

https://github.com/daviddao/spatial-transformer-tensorflow
This repository contains the Cluttered MNIST dataset used for the simulations.

In order to being able to use the code in the previous repositories it is needed to have Python installed, the TensorFlow package in some Python environment and a tool to run Notebooks (such as Jupiter).

Chapter 4

Results

4.1 Regular CNN

As mentioned previously, the first simulation has been performed using the network depicted in Figure 3.1. Here we report the results obtained with this setup.

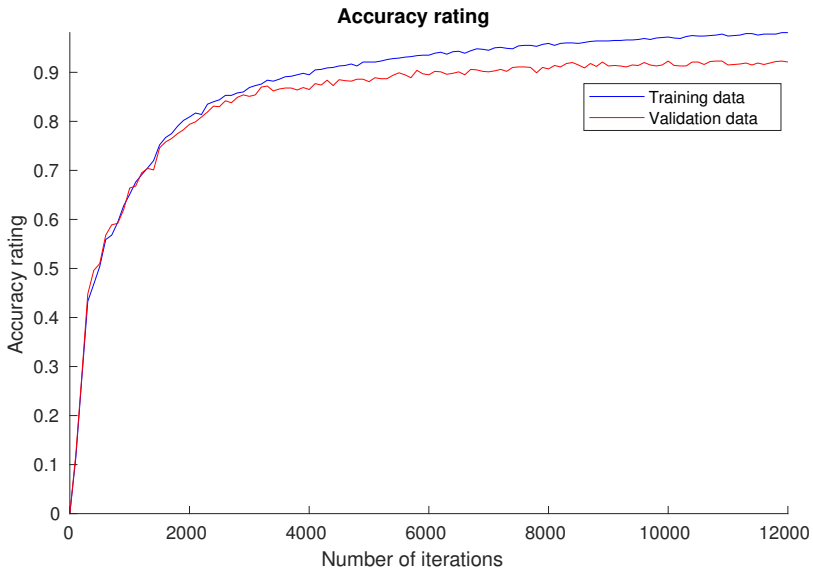


Figure 4.1: Accuracy rating of the core network as a function of the number of iterations

For the graph we can see that the training is pretty smooth and after around 10000 iteration we are able to achieve an accuracy rating of 98% on the training set and of 93% on the validation set.
In the next figures we can observe the values taken by the filters of the network. Red values represents positive weights, blue values represents negative weights.

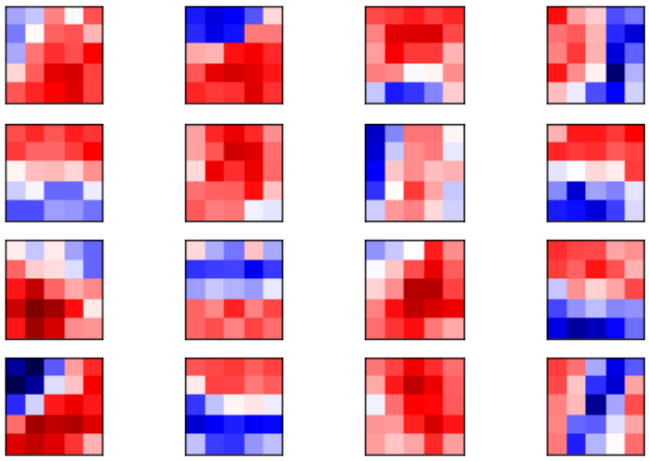


Figure 4.2: Values of the filters in the first convolutional layer

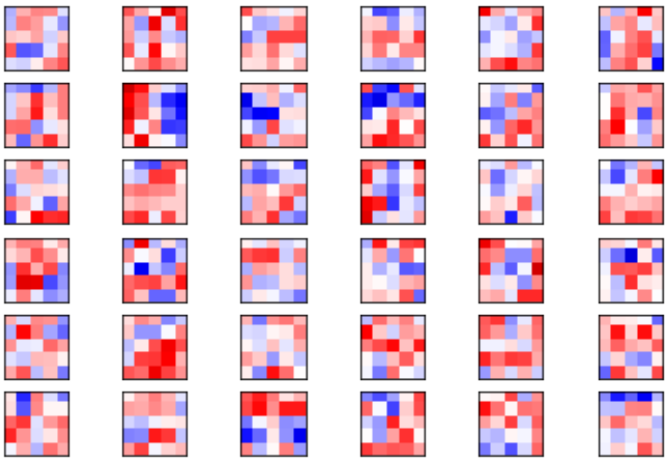


Figure 4.3: Values of the filters in the second convolutional layer

Figure 4.4: Values of the filters in the last fully connected layer

According to [1], the filters in the first layer are able to detect simple features (in this case they seems to be edges) while in the higher layers more complex ones are recognized (in this case it's almost impossible to say what they are identifying).

Taking a random input image, we can also observe the output of different layers.

Figure 4.5: Sample image

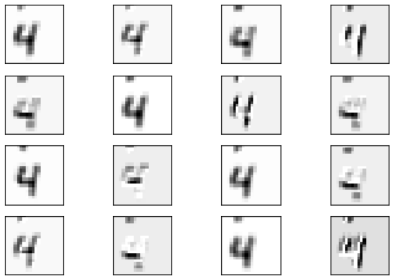


Figure 4.6: Feature maps at the output of the first convolutional layer

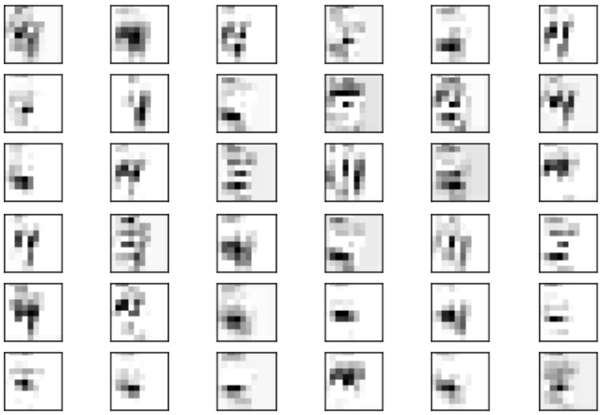


Figure 4.7: Feature maps at the output of the second convolutional layer

4.2 Replacing softmax

One of the first alternative configurations we examined has been the network without the softmax layer. Instead of placing it, we used other forms of normalization, in particular we applied the L1 and L2 norm. The idea behind this substitution is based on the interpretation we give to our network. In fact, based on the equation 3.4 we can interpret the output of a layer as the SAL. For this reason we expect the output of the last layer to be a probability and so we impose this constrain.
In fact, with softmax we would have a probability distribution just at the top of the CNN, but the function $g(x)$ learned at the output of the last fully connected layer would be a log-probability. With the aforementioned substitution we expect the network to being able to adjust its weights in order to compute the function $e^{g(x)}$ directly, eliminating the need of a softmax layer.

4.2.1 L2 normalization

First we show how the network behaves when we replace the softmax layer with the L2-norm. In order to impose this constrain we used the following equation:

$$\hat{y}_j = \frac{t_j}{\sqrt{\sum_{i=0}^{K-1} t_i^2}} \tag{4.1}$$

where $\hat{y}_j$ is the j-th component of the output layer of the network and t_i is the i-th component of the last fully connected layer.

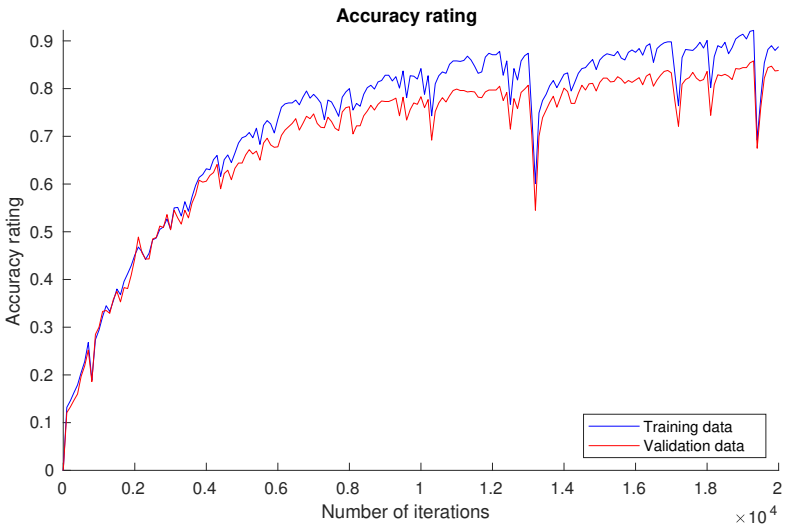


Figure 4.8: Accuracy rating of the network with L2 normalization as a function of the number of iterations

In Figure 4.11 we can observe that the training becomes slower and a bit more unstable. In fact, around 13000, 17000 and 18000 iterations there are spikes that decreases considerably the accuracy of the network. We can also notice that even after 20000 iterations this configuration isn't able to achieve the same score of the previous one.
Inspecting the weights of this setup we obtain the following pictures.

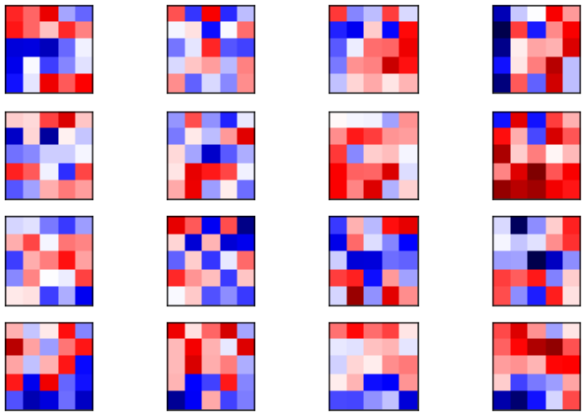


Figure 4.9: Values of the filters in the first convolutional layer

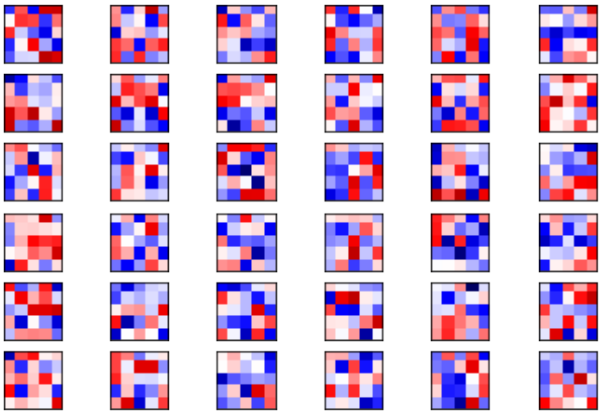


Figure 4.10: Values of the filters in the second convolutional layer

In this case we can see that the first layer filters are weird: while some seems to identifies edges or curves, the others don't show understandable patterns. The second layer filters doesn't show recognizable structure, but they appears more messy compared to the ones in the previous configuration.

4.2.2 L1 normalization

Similarly to the previous case, we ran the simulation using the L1 norm. In order to impose this normalization we used the following formula:

$$\hat{y}_j = \frac{t_j}{\sum_{i=0}^{K-1} |t_i|} \tag{4.2}$$

where $\hat{y}_j$ is the j-th component of the output layer of the network and t_i is the i-th component of the last fully connected layer.
In the following pictures the accuracy rating as a function of the number of iterations and the values taken by the weights are depicted.

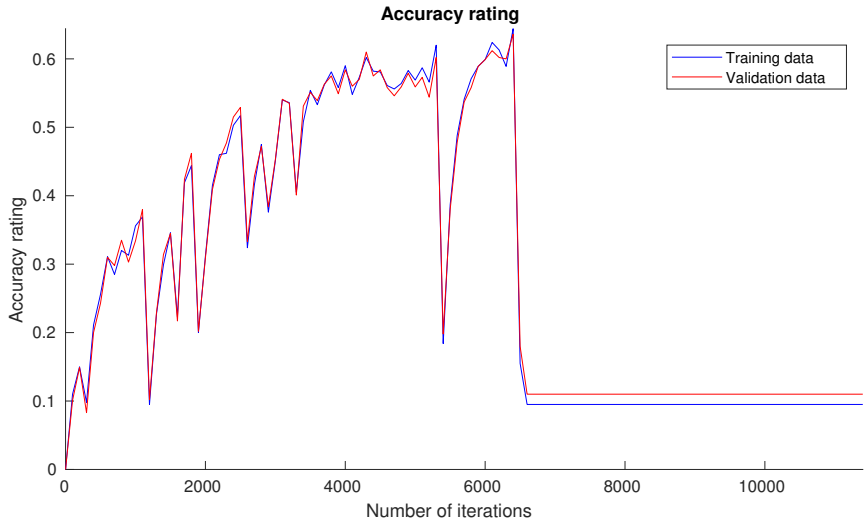


Figure 4.11: Accuracy rating of the network with L1 normalization as a function of the number of iterations

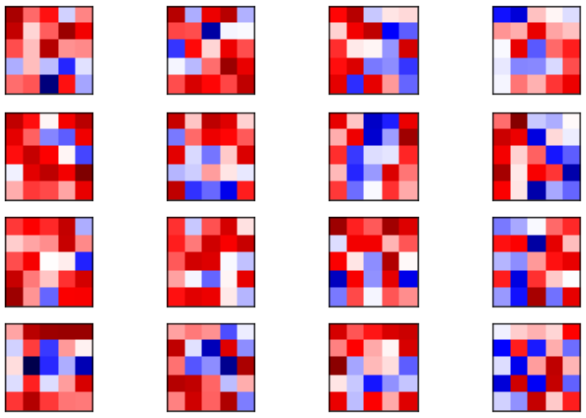


Figure 4.12: Values of the filters in the first convolutional layer

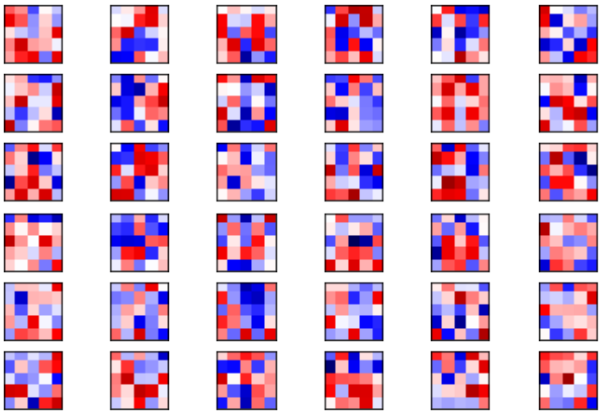


Figure 4.13: Values of the filters in the second convolutional layer

Here we can see that the training becomes extremely unstable and the top accuracy rating is barely 65%. On top of that we reach a point in which the network isn't able to discriminate the inputs and isn't able to update its weights in order to achieve better results. Weights in the first layer doesn't present recognizable kernels.

Both L2 and L1 normalization haven't performed so well. The reason why these methods haven't worked nicely might be caused by the optimizer which computes smaller errors and consequently updates the weights slower.
In fact, the optimizer tries to minimize the cross-entropy defined as:

$$H(\theta) = -\sum_{i=0}^{K-1} p_i \, log \, q_i(\theta) \tag{4.3}$$

In our case this computation is equivalent to the following

$$H(\theta) = -y \cdot log \, \hat{y} \tag{4.4}$$

Where y represent the true label one-hot encoded and $\hat{y}$ represents the output vector of our neural network. For example, if the true class is 8 and the output vector of our CNN is $\hat{y} = [0\ 0\ 0.3\ 0\ 0\ 0\ 0\ 0\ 0.7\ 0]$, the cross entropy is given by the following computation:

$$
\begin{aligned}
&y \leftarrow oh(8)\\
&y = [0\,0\,0\,0\,0\,0\,0\,0\,1\,0]\\
&e = [\epsilon\,\epsilon\,\epsilon\,\epsilon\,\epsilon\,\epsilon\,\epsilon\,\epsilon\,\epsilon\,\epsilon]\\
&H(\theta) = -[0\,0\,0\,0\,0\,0\,0\,0\,1\,0] \cdot log(e + [0\,0\,0.3\,0\,0\,0\,0\,0\,0.7\,0])\\
&= -log(0.7 + \epsilon) = 0.3567
\end{aligned}
\tag{4.5}
$$

where oh is the one-hot encoder function and e is a vector of small values $\epsilon = 10^{-8}$ added to avoid the computation of the $log(0)$.
Since all the digits in our database belong to a single class, we expect y to be a vector composed by all elements equal to zero beside the j-th equal to one. With this consideration we can write

$$H(\theta) = -log(\hat{y}_j) \tag{4.6}$$

where $\hat{y}_j$ is the j-th component of the output layer of the network.
In case we use softmax, $\hat{y}_j$ is defined as follow:

$$\hat{y}_j = \frac{e^{t_j}}{\sum_{i=0}^{K-1} e^{t_i}} \tag{4.7}$$

where t_i is the i-th component of the vector that must be transformed with the softmax.
In this case the cross entropy is

$$
\begin{aligned}
H(\theta) &= -log(\hat{y}_j) =\\
&= -log(\frac{e^{t_j}}{\sum_{i=0}^{K-1} e^{t_i}}) =\\
&= -t_j + log(\sum_{i=0}^{K-1} e^{t_i})
\end{aligned}
\tag{4.8}
$$

Instead, using for example the L2 norm, we have

$$\hat{y}_j = \frac{t_j}{\sqrt{\sum_{i=0}^{K-1} t_i^2}} \tag{4.9}$$

that means that the cross-entropy is

$$
\begin{aligned}
H(\theta) &= -log(\hat{y}_j) =\\
&= -log(\frac{t_j}{\sqrt{\sum_{i=0}^{K-1} t_i^2}}) =\\
&= -log\, t_j + \frac{1}{2} log(\sum_{i=0}^{K-1} t_i^2)
\end{aligned}
\tag{4.10}
$$

Since the t_i are always grater then zero because they are the output of an activation function, comparing (4.10) and (4.8) we notice that the latter yields grater errors causing stronger weights updates.
The computation can be generalized to the case in which the loss function is the mean of the cross-entropies computed over the batch.
While this consideration might justify slower training rates, this does not justify the accuracy drops that occurs using these configurations.

In order to go deeper into this problem, we repeated the simulations tracking how the weights were updated over time. In Figure 4.14 and 4.15 we have the mean and the standard deviation of the variation of the weights using the softmax while in Figure 4.16 and 4.17 we have the analogous graphs using the L1 normalization.

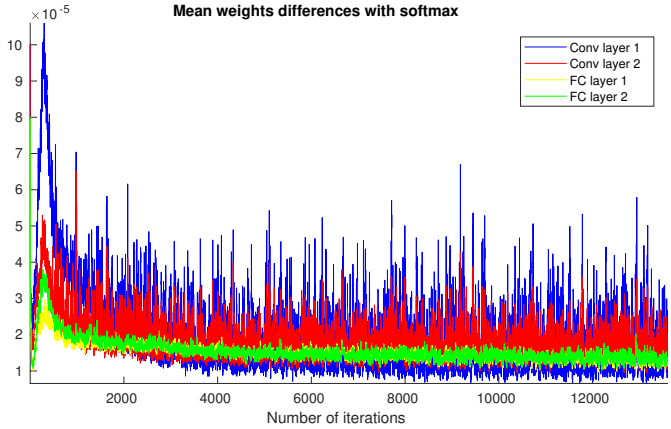


Figure 4.14: Difference of weights at each iteration using the softmax.

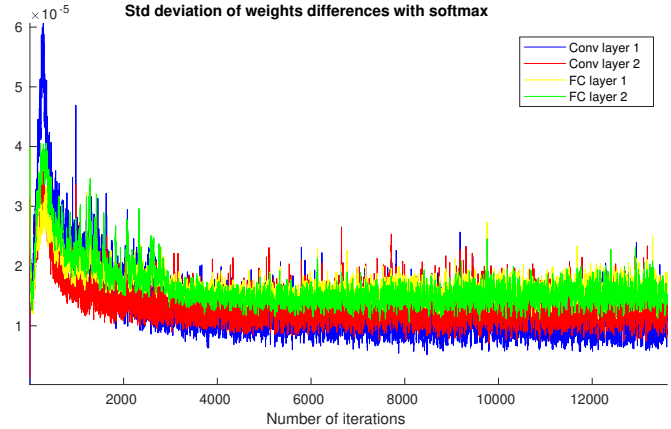


Figure 4.15: Standard deviation of the variation of the weights using the softmax

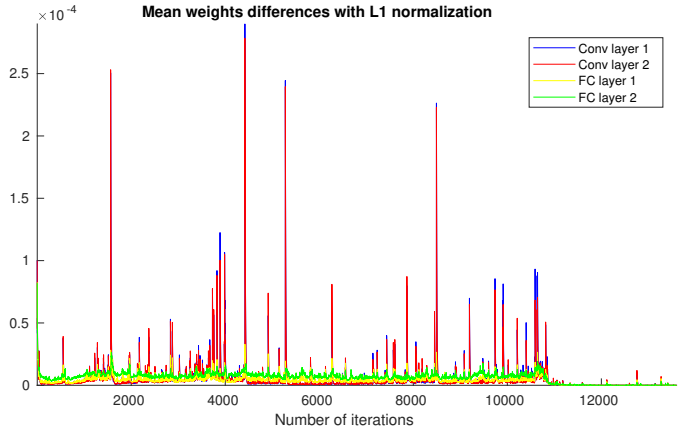


Figure 4.16: Difference of weights at each iteration using L1 normalization.

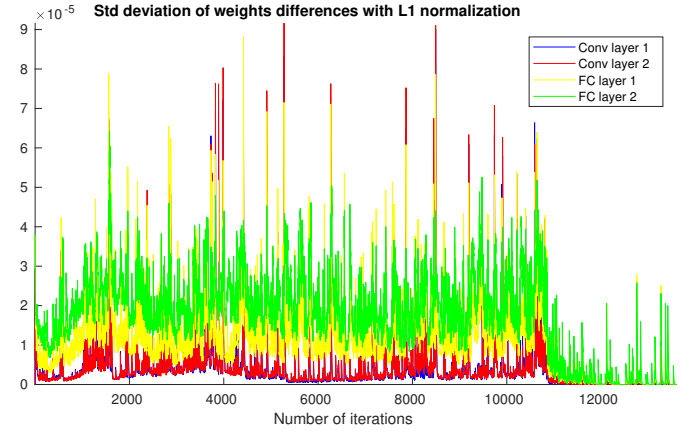


Figure 4.17: Standard deviation of the variation of the weights using L1 normalization

From these last graphs we can see that the most significant variations occurs in the first two layers. This is somehow counter-intuitive since we would expect that the first layers suffers from gradient vanishing more then the layer at the top of the network. However, the main difference between the network with the softmax and the one with the L1 normalization is that in the first case the variation tends to decrease together with the standard deviations while in the second case there are several points in which the weights receives considerable swings.

Studying these cases hasn't really clarified why alternative form of normalization yield worse result then the regular CNN. Inspired by another work [11] which apply the normalization before the activation function, we tried to use restart the simulations following the same approach. The results of this configuration are exposed in Figures 4.18, 4.19 and 4.20. We can observe that the normalization applied before the activation function yields considerable improvements on the performance of the network. Both L1 and L2 normalization are able to achieve the same results of the network that uses softmax even if with the training is a bit slower.

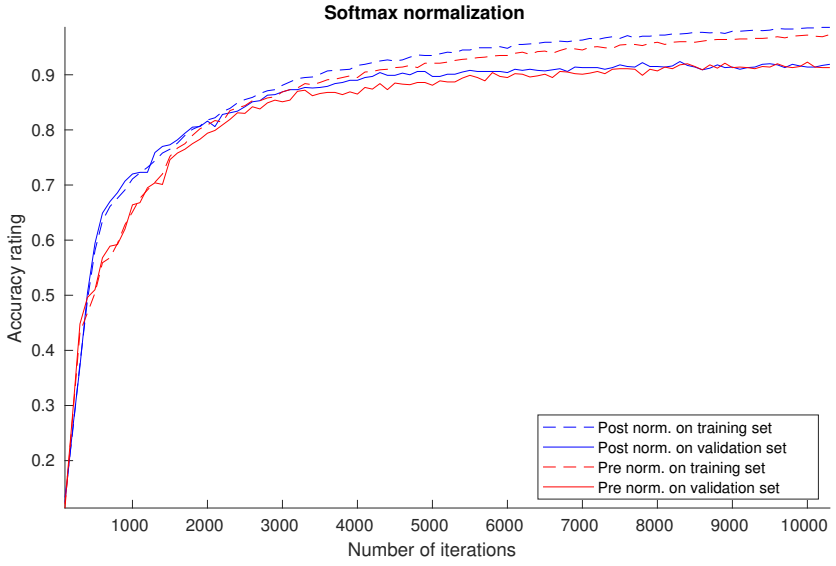


Figure 4.18: Softmax

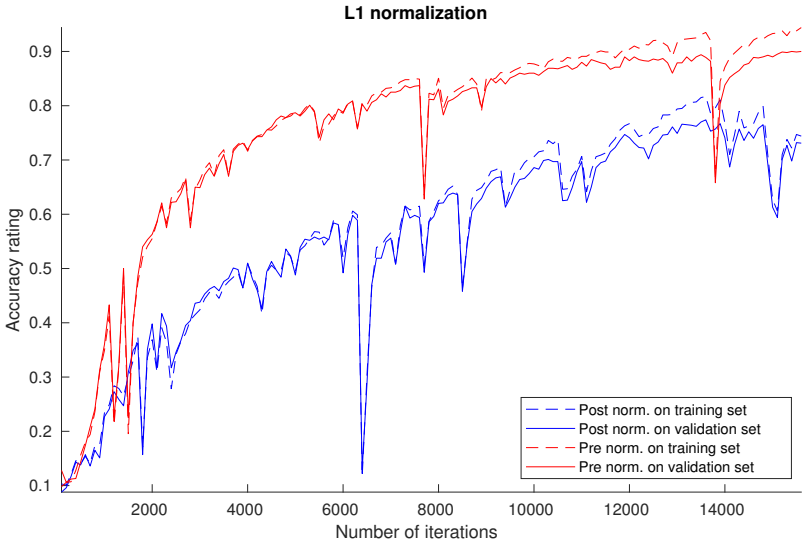


Figure 4.19: L1 normalization

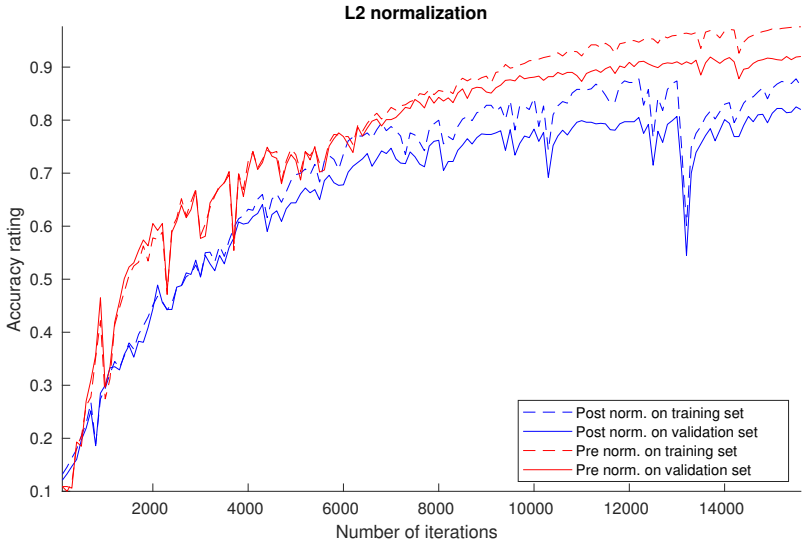


Figure 4.20: L2 normalization

4.2.3 Normalizing weights

Another configuration that we used has been the one in which the weights of the filters of the first convolutional layer was normalized. Usually just features map are rectified and normalized, not the filters. However, in order to consider weights as probabilities as suggested by the equation 3.4, we tried to normalize the filters.
The result is shown in Figure 4.21

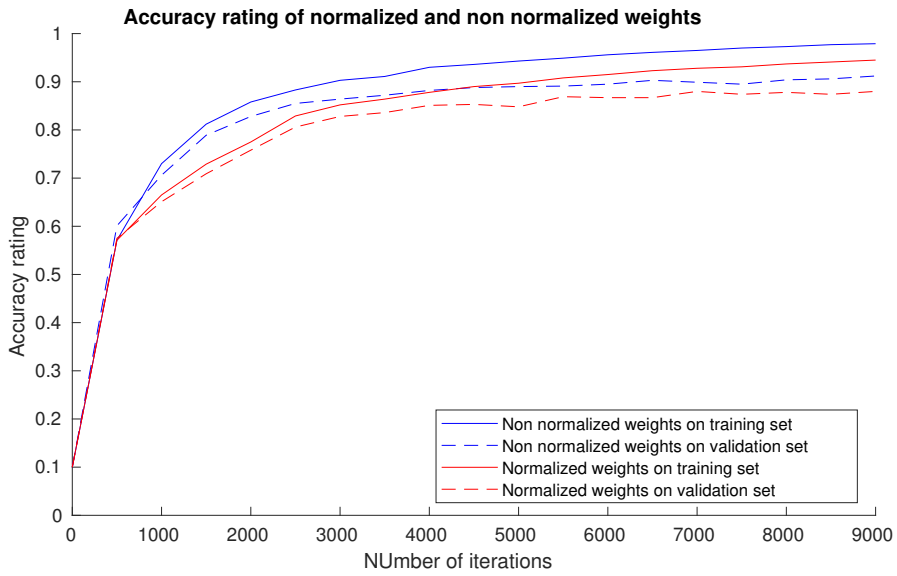


Figure 4.21: Accuracy rating as a function of the iteration number when weights are normalized

4.2.4 Replacing ReLu with Sigmoid

Our last configuration has been the one in which we replaced the ReLU activation function with the Sigmoid function. In the first place we used the sigmoid without any form of normalization. The result with this setup was poor because the optimizer wasn't able to train the network in order to prefer a class over another.
After having introduced the normalization, instead, the network recovered its discriminative power but no real benefits has been found. In Figure 4.22 the accuracy rating as a function of the number of iterations is shown.

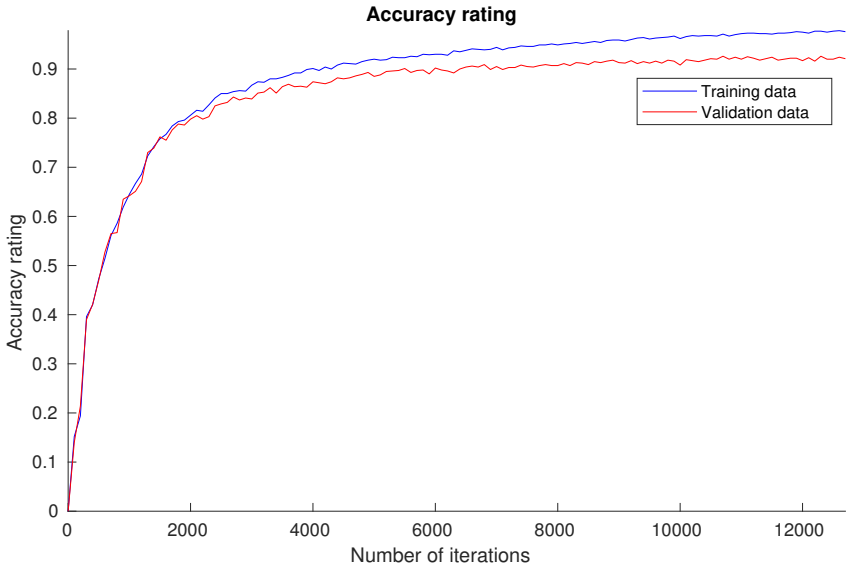


Figure 4.22: Accuracy rating as a function of the iteration number when using the Sigmoid

4.3 Table of results

In the previous sections we have shown the results obtained with different configurations of the convolutional neural network. For each configuration many simulations have been performed, still the graphs represents particular cases of each. Plotting the mean would have been pointless since some simulations using the same setup given very different results and so the mean would have shown something unrealistic. By the way, it is still useful to have a statistic description of the result obtained in the different setups used.

Simulation	BS	μ_{BS}	σ^2_{BS}	$Iter_{BS}$	$\mu_{Iter_{BS}}$	$\sigma^2_{Iter_{BS}}$ (x10^6)
Regular CNN	93.1	92.55	0.17	10300	11967	1.42
L2 norm	86.8	84.4	4.1	20000	19040	1.9
L1 norm	77.4	72.7	37	15200	14840	29.8
Soft pre-norm	92.9	92.4	0.23	10500	12140	2
L2 pre-norm	93	92.4	0.21	16400	16020	2.3
L1 pre-norm	92.1	91	0.52	15800	15660	1.54
Weights norm	92.2	91.1	1.71	8600	9100	3.1
Sigmoid	92.6	91.3	3.6	10700	10360	1.2

The different columns represents the following values:

- *Simulations* is a the short description of the setup used.

- BS is the value of the best score achieved.
- μ_{BS} is the mean of the best scores obtained.
- σ^2_{BS} is the variance of the best scores.
- $Iter_{BS}$ is the number of iterations performed to achieve the best score.
- $\mu_{Iter_{BS}}$ is the mean of the iterations performed to achieve the best scores of each simulation.
- $\sigma_{Iter_{BS}}$ is the variance of the iterations performed to achieve best scores of each simulation.

The number of simulation performed in each setup was ranging between 8-10.

Chapter 5

Conclusions

In this document we wanted to verify the relationship between convolutional neural networks and visual representations. In order to do so we implemented a convolutional neural network and modified it by implementing some operations that reflected the approximation made for the SA likelihood [15].

The first simulation we made was replacing the softmax layer with other forms of normalization. The results obtained in this way weren't optimal and the training was pretty unstable. We can't state if this result is obtained because the outputs of the last layer of the convolutional neural network cannot be interpreted as probabilities or because other factors degraded the training. For example, if the output of the last layer are close to zero, normalizing would result in a division by a very small number that may cause some instability problems.

Normalizing the weights of the filters in the first layer haven't really changed the performance of the network (on average we lost 1.45% of accuracy rating). This means that interpreting weights as probabilities is compatible with the current implementations of the CNNs, still it doesn't give additional information about the approximation of the SAL.

Lastly we replaced the activation function with the sigmoid. The results with this setup was unsurprisingly good since this kind of function is largely adopted in traditional architectures. The only interesting thing to note is how the softmax or the sigmoid have the capability to stabilize the process of training making it extremely smooth. This is probably due to the region where these function operates. In fact, when the network is firstly initialized, the weights are around zero and these functions operates in the linear region. Once the training proceeds, they start to

operate in the saturation region making them more insensitive to the changes. This has the benefit that the weights doesn't chance drastically between an iteration an another leading consequently to a smooth training.

In conclusion we haven't been able to verify or deny the relationship between CNN and visual representations. Normalizing the weights has been tested to be compatible with the current realizations of the CNNs whereas replacing the softmax led to bad results. However, this isn't necessary caused by a coarse approximation of the likelihood but may be caused by other elements. A deeper investigation on the parameters of the network is required in order to understand which might be the factors that lead to these results.

Bibliography

[1] Matthew D. Zeiler and Rob Fergus "Visualizing and Understanding Convolutional Networks". http://www.matthewzeiler.com/pubs/arxive2013/arxive2013.pdf. 28 November 2013.

[2] https://en.wikipedia.org/wiki/Artificial_neural_network

[3] Trevor Hastie, Robert Tibshirani, Jerome Friedman "The Elements of Statistical Learning" Second Edition Cap 11, pag 393.

[4] Mu Li, Tong Zhang, Yuquiang Chen, Alexander J. Smola "Efficient mini-batch training for stochastic optimization" published in Proceedings of the 20th ACM SIGKDD international conference of Knowledge discovery and data mining. Pages 661-670. 24-27 August 2014.

[5] https://en.wikipedia.org/wiki/Backpropagation

[6] Razvan Pascanu, Tomas Mikolow, Yoshua Bengio"On the difficulty of training recurrent neural networks". http://www.jmlr.org/proceedings/papers/v28/pascanu13.pdf. 2013.

[7] https://en.wikipedia.org/wiki/Convolutional_neural_network

[8] https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

[9] Yann LeCun, Leon Bottou, Genevieve B. Orr and Klaus-Robert Muller "Efficient BackProp". http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.

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

[11] Sergey Ioffe, Christian Szegedy "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". https://arxiv.org/pdf/1502.03167v3.pdf. 2 March 2015.

[12] Kaiming He, xiangyu Zhang, Shaoquing Ren, Jian Sun "Deep Residual Learning for Image Recognition". https://arxiv.org/pdf/1512.03385v1.pdf. 10 December 2015.

[13] Karen Simonyan and Andrew Zisserman "Very Deep Convolutional Networks for Large-Scale Image Recognition". https://arxiv.org/pdf/1409.1556v6.pdf. 10 April 2015.

[14] Christian Szegedy, Wei Liu, Yangquing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelow, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich "Going Deeper with Convolutions". http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf.

[15] Stefano Soatto, Alessandro Chiuso "Visual Representation: Defining Properties and Deep Approximations". https://arxiv.org/pdf/1411.7676v9.pdf. 29 February 2016.

[16] https://www.tensorflow.org/

[17] Siedderik P. Kingma, Jimmy Lei Ba "Adam: A Method for Stochastic Optimization". https://arxiv.org/pdf/1412.6980v8.pdf. 23 July 2015.

[18] http://jupyter.org/

[19] https://www.continuum.io/downloads

[20] https://github.com/daviddao/spatial-transformer-tensorflow

[21] Ron Kohavi, "A study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection". Appears in the International Joint Conference on Artificial Intelligence (IJCAI), 1995. http://ai.stanford.edu/ ronnyk/accEst.pdf.

[22] Kurt Hornik, "Approximation Capabilities of Multilayer Feedforward Networks". Neural Networks, Vol. 4. pp. 251-257, 1991. http://zmjones.com/static/statistical-learning/hornik-nn-1991.pdf.