Alma Mater Studiorum · Università di Bologna

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

Exploring CNNs: an application study on nuclei recognition task in colon cancer histology images

Relatore:
Prof. Renato Campanini

Correlatore:
Dott. Matteo Roffilli

Presentata da:
Carmine Mattia

Anno Accademico 2015/2016

Abstract

In this work we explore the recent advances in the field of *Convolutional Neural Network* (CNN), with particular interest to the task of image classification. Moreover, we explore a new neural network algorithm, called ladder network, which enables the semi-supervised framework on pre-existing neural networks.

These techniques were applied to a task of nuclei classification in routine colon cancer histology images.

Specifically, starting from an existing CNN developed for this purpose, we improve its performances utilizing a better data augmentation, a more efficient initialization of the network and adding the batch normalization layer. These improvements were made to achieve a state-of-the-art architecture which could be compatible with the ladder network algorithm. A specific custom version of the ladder network algorithm was implemented in our CNN in order to use the amount of data without a label presented with the used database.

However we observed a deterioration of the performances using the unlabeled examples of this database, probably due to a distribution bias in them compared to the labeled ones.

Even without using of the semi-supervised framework, the ladder algorithm allows to obtain a better representation in the CNN which leads to a dramatic performance improvement of the starting CNN algorithm.

We reach this result only with a little increase in complexity of the final model, working specifically on the training process of the algorithm.

Sommario

In questo lavoro sono stati esplorati i recenti progressi nel campo delle *convolutional neural network* (CNN), in particolare nel compito della classificazione di immagini. Abbiamo approfondito inoltre un nuovo algoritmo basato sulle reti neurali artificiali, chiamato ladder network, che permette di addestrare in modo semi-supervisionato delle preesistenti reti neurali.

Queste tecniche sono state applicate ad un problema di classificazione di nuclei di immagini istologiche di cancro al colon.

In dettaglio, partendo da una esistente CNN costruita per affrontare questo compito, ne abbiamo migliorato le prestazioni utilizzando un miglior *data augmentation*, una migliore inizializzazione della rete e facendo uso del batch normalization layer.

Questi miglioramenti sono stati fatti con lo scopo di avere un'architettura allo stato dell'arte e che potesse essere compatibile con l'uso della ladder network. Una specifica e personalizzata versione della ladder network è stata implemetata sulla nostra CNN con lo scopo di poter utilizzare la quantità di dati senza una *label* presente nel database impiegato.

Tuttavia abbiamo osservato un peggioramento nelle prestazioni dell'algoritmo dovuto all'utilizzo dei dati non etichettati presenti nel database, probabilmente dovuto ad un bias nella distribuzione dei dati non etichettati rispetto alla distribuzione dei dati correttamente etichettati.

Anche senza fare uso dell'addestramento semi-supervisionato, la ladder network permette di ottenere una migliore rappresentazione nella CNN che si ripercuote su un drastico aumento delle prestazioni rispetto all'algoritmo CNN di base. Questi risultati sono stati raggiunti solo con un piccolo aumento in complessità dell'algoritmo finale, lavorando in particolar modo sul processo di addestramento dello stesso.

Contents

Introduction

The idea of creating machines able to think like a human being attracted, since old times, philosophers and scientists. In the last century, with the arrival of computers, this possibility started to take form, giving birth to a new research environment commonly known as *Artificial Intelligence* (AI).

One of the greatest challenges for the AI it's always been building machines that were able to execute tasks, very simple and intuitive for a human being, but with a hard mathematical formalization. Some examples of this kind of tasks are: recognition of spoken words, text production or recognition of complex images.

This kind of problems have been partially solved by building machines able to learn from experience, so not imitating the way human brain works but the way it learns. This research field take the name of *Machine Learning.*

Machine learning allowed, since it's birth, to carry on tasks at an even higher level of abstraction of the ones commonly made by computers. One of the most important turning points was the introduction of the *artificial neural network*, commonly known as *ANN* or simply as *neural network*, trying to imitate the structure and the behaviour of the human brain. The ANN immediately displayed a huge ability to solve difficult tasks like recognizing simple images.

Nevertheless, for years, we couldn't fully use the potential of these algorithm due to the computational limits caused by the complexity of these algorithm and the lack of large databases these machines need to get experience.

These limits were overtaken with the introduction of the graphical processing units (*GPU*) and the internet advents. As a matter of facts, if the GPU brought an extremely functional platform for the development of even faster and powerful algorithm, internet gave a platform and a community that was able to provide for the lack of big database.

In the last few years, the attention towards the machine learning, and towards the ANN in particular, has become extremely high, and a great number of researches are conducted on a daily basis. A specific type of ANN, in particular, specialized in image processing, rouse attention in this field, the *Convolutional Neural Network.* The introduction of the graphic processors and the possibility to get access to a vast amount of data made possible the realization of even more complex CNNs, that today are commonly knows as “Deep Learning”, a term generally used to describe particularly complex algorithms of the Machine Learning.

The CNN’s proved to be able to solve extremely complex tasks in the most disparate applications, from video games, to the surveillance, to the industrial and the medical field.

In this last area in particular, the CNNs has the potential to provide an important help in the medical imaging, as a backing to diagnosis. The work provided here is meant to be a contribution in the usage of the recent technologies developed for the CNNs in the field of image diagnosis. In particular, it faces the issue of recognizing cellular nuclei in routine colon cancer histology images. This work was carried on starting from a previous implementation of a CNN modified to improve it.

In details this work is structured as follow.

In Chapter 1 is presented a brief introduction on machine learning, exploring the principal topics and problems about this field. The basic algorithms are presented and the general method to train a machine learning algorithm, using the data and a supervisor which driving the learning process (*supervised learning*), are introduced.

In Chapter 2 we focus on Convolutional Neural Network algorithms, presenting the basic idea, the standard implementation and the recent improvements in this field. A newborn algorithm is also presented which enable the possibility to use the supervised learning process with the advantage of using data without a supervisor (*semi-supervised learning*).

In Chapter 3 is introduced the problem of nuclei classification in colon cancer histology images and the database used in this work is presented. The entire learning process is described in details, presenting the pre-processing operations and the exact CNN algorithms implemented.

In the last chapter the results on various algorithms were reported and a final test on the best performing algorithm is shown, compared with the results reported in other works.

Chapter 1

Machine learning

Machine learning is a subfield of computer science, evolved from the study of pattern recognition and artificial intelligence, that explores the study of algorithms that have the ability to learn from data, without the need of explicitly programming them on a particular task. In this section are exposed the basic principles of machine learning, with particular emphasis on machine learning on classification task.

The content of this chapter was adapted from [6] and [8].

1.1 Learning

In a general way, learning is the process of estimating an unknown structure of a system using a limited number of observations. This process involves three components: a *Generator*, a *System* and a *Learning Machine.*

Generator, a sampling distribution, produces independent random vectors $\vec{x}$ from a fixed unknown probability density $p(\vec{x})$

System produces an output value y for every input vector $\vec{x}$ produced by *Generator*, according with a fixed and unknown conditional density $p(y|\vec{x})$.

Learning Machine uses samples from *Generator* and outputs from *System* to form an approximation of the *System* itself. *Learning machine* is capable of implementing a set of functions chosen a priori before the learning process (*inference*). *Inference* choose the best function $f(\vec{x}, \omega)$, $\omega \in \Omega$, where Ω is a set of abstract parameters.

The entire process is schematized in fig. 1.1.

Learning problems can be grouped in two principal frameworks: *statistical model estimation* and *predictive learning.* The aim of the first one is an accurate identification of the unknown system, whereas the aim of the latter is an accurate imitation of it.

In this work we are interested in particular on predictive learning. An important

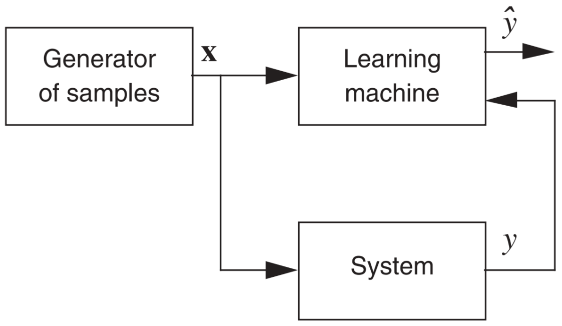


Figure 1.1: Block diagram of learning process. Image from [6].

difference between the two frameworks is that in predictive learning a good predictive model depends upon the distribution of the input samples, whereas it's not true on model estimation. Therefore an accurate model estimation provide a good generalization on prediction, while the opposite is not true. However, due to the finite amount of input samples, model estimation is a very difficult task.

Under this assumptions, we can define a machine learning algorithm as an algorithm able to learn using data. These data, called *Training data*, must be independent and identically distributed according to the probability density function (*pdf*):

$$p(\vec{x}, y) \;=\; p(x)p(y|\vec{x}) \qquad (\vec{x_i}, y_i),\ (i = 1, ..., n) \tag{1.1}$$

The goodness of the learning process can be measured by the *loss function* $L(y, f(\vec{x}, \omega))$, a discrepancy between the output produced by the System and the Learning Machine, for a given input $\vec{x}$. Conventionally, loss function takes only non-negative values, so near zero value correspond to accurate approximation. The expected value of the loss function can be calculated as follows, it is called the *risk functional*:

$$R(\omega) \;=\; \int L(y, f(\vec{x}, \omega))p(\vec{x}, y)\,\mathrm{d}\vec{x}\,\mathrm{d}y \tag{1.2}$$

After this formal definitions, we can say that learning is the process of choose $f(\vec{x}, \omega_0)$, from the available set of the Learning Machine, which minimizes $R(\omega)$. This problem became particularly difficult with finite data because we cannot know $p(\vec{x}, y)$ and we cannot find the exactly function which minimizes 1.2. So, to obtain an unique solution, learning process need some priori knowledge in addition to data. This priori knowledge are incorporated in the set of functions $f(\vec{x}, \omega)$).

Then, in real machine learning applications, is correct to talk about empirical risk, averaging loss function over the training data:

$$R_{emp}(\omega) \;=\; \frac{1}{n}\sum_{i=1}^{n} L(y, f(\vec{x},\omega)) \tag{1.3}$$

1.2 Learning Tasks

The generic learning problem can be divided considering the task solved. A lot of tasks can be solved with machine learning. In this section we described only some of them.

Density estimation In this task Learning Machine target is estimating the probability density function of $\vec{x}$. This is an unsupervised learning problem because the System output is not used. To do this, the algorithm needs to learn the distribution of data Generator, capturing the underline structure of the data it has seen.

A usually loss function for this task is:

$$L(f(\vec{x},\omega)) = -\ln f(\vec{x},\omega) \tag{1.4}$$

Classification: In classification task, machine learning algorithm is asked to specify an output categories for each input vector $\vec{x}$. Usually the learning machine produce a function $f : \mathbb{R}^n \to \{1, ..., k\}$, where k are the numbers of categories. This output vector is associated, symbolically, with the output y of the system. In some cases, output vector can be interpreted as a distribution probability over categories.

For example, in a two-class classification problem the output of the system takes only two values ($y = \{0, 1\}$ e.g.). We want that Learning Machine only outputs this two values, so we need a set of *indicator* functions $I(\vec{x},\omega)$ to choose from.

Let's look at this in a probabilistic way, an approach followed along this work. Using Bayes theorem, we can describe the probability than a given observation belongs to each class as:

$$\begin{aligned} P(y=0|\vec{x}) &= \frac{p(\vec{x}|y=0)P(y=0)}{p(\vec{x})} \\ P(y=1|\vec{x}) &= \frac{p(\vec{x}|y=1)P(y=1)}{p(\vec{x})} \end{aligned} \tag{1.5}$$

where $p(\vec{x}|y)$ are the conditional densities for each class, $P(y)$ are the prior probabilities and $P(y|\vec{x})$ are the posterior probabilities. Usually there is no reason to compute $p(\vec{x})$, because we are only interested in the relative magnitude of the posterior probabilities. The indicator function became:

$$I(\vec{x}) \;=\; \begin{cases} 0 & if\ p(\vec{x}|y=0)P(y=0) > p(\vec{x}|y=1)P(y=1) \\ 1 & otherwise \end{cases} \tag{1.6}$$

While the prior probabilities are described by the input data distribution, the conditional densities are unknown. This approach corresponds to estimate the parameters of the indicator functions that describe, indirectly and best as possible, this distributions in order to have a good decision rule.

A simple loss function that can be used, to minimize 1.2, in this problem is:

$$L(y, I(\vec{x}, \omega)) \;=\; \begin{cases} 0 & if\ y = I(\vec{x}, \omega) \\ 1 & if\ y \neq I(\vec{x}, \omega) \end{cases} \tag{1.7}$$

An example of classification is object recognition, where input vectors $\vec{x}$ is an image or a representations of it (*features vector*).

Regression: Regression is another common task in machine learning, whose target is estimating a real valued function based on noisy input vectors. In this case the System output is a random real value, interpretable as the sum among a deterministic function $t(\vec{x})$ and a zero mean error ξ.

$$y = t(\vec{x}) + \xi \qquad where \quad t(\vec{x}) = \int y p(y|\vec{x})\,\mathrm{d}y \tag{1.8}$$

A common loss function in regression task is the squared euclidean distance:

$$L(y, f(\vec{x}, \omega)) = (\, y - f(\vec{x}, \omega)\,)^2 \tag{1.9}$$

An example of regression is the prediction of the future prices of an object, given the past values.

Denoising A denoising learning algorithm must predict a clean example $\vec{x}$ from a noised input (*corrupted*). This task is quite similar to regression task, with some differences. System output is the clean example $\vec{x}$ of the corrupted input, while the input of the Learning Machine is the corrupted input vector $\vec{x}_{noisy}$.

$$\begin{gathered} y = \vec{x} \quad and \quad \vec{x}_{noisy} = \vec{x} + \xi \\ L(\vec{x}, f(\vec{x}_{noisy}, \omega)) = (\, \vec{x} - \vec{x}_{noisy}\,)^2 \end{gathered} \tag{1.10}$$

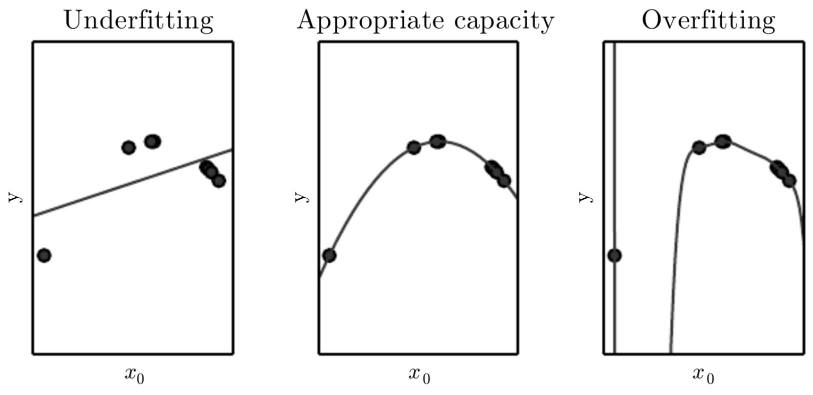


Figure 1.2: Intuition about capacity of an algorithm. With poor capacity, an algorithm can't fit appropriately the points of the training set (*underfitting*). However an excessive capacity produce only a memorization of the training points without generalization. An optimal capacity is raffigurate in the middle, with a smooth function. Image from [8]

1.3 Capacity and regularization

When we train a machine learning algorithm, we want principally good *generalization* performance. Generalization is the ability of perform well, according to task, on previously unseen data. Mathematically, during training, we can measure the algorithms performance according to 1.2 on the training data. We can also call this measure *training error*. Otherwise we referred usually to *generalization error* as *test error*.

Test data are examples that were collected separately from the training data but from the same probability distribution. In particular, *training* and *test* data must be *independent* and *identically distributed*.

Under this assumptions (*i.i.d. assumptions*) we can find a relation between the minimization of the *training error* and the minimization of the *test error*.

During training, we can only use training data to adjust parameters. From this we are interested in make the training error small while make the gap between training and test error as small as possible. These are two central challenges in machine learning, also referred respectively as *underfitting* and *overfitting*.

Underfitting is the condition in which the machine learning model is not able to reach a sufficiently small training error. Overfitting occurs when the gap between training and test error is too large. These conditions can be described introducing the concept of *capacity*, also represented in fig. 1.2.

Capacity, roughly speaking, is the ability of a Learning Machine to fit a large variety of functions. A low capacity is related with underfitting, the model is not able to capture the complexity of the data. High capacity, otherwise, can fall in overfitting and the algorithm simply memorize the training data with poor generalization performances, as shown in fig. 1.3.

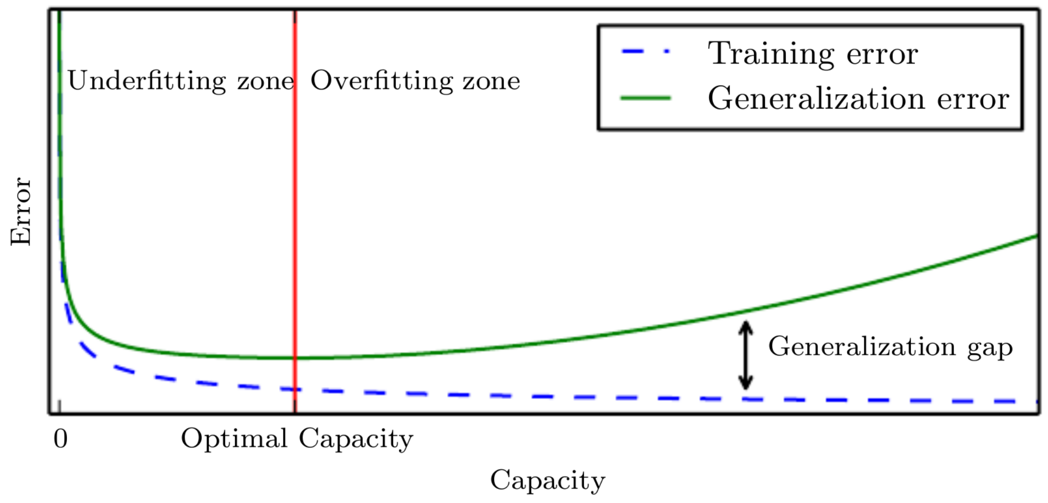


Figure 1.3: Relation between capacity and *Training* and *Test error*. In underfitting zone, both errors are high. Decreasing capacity, training error tend to reach the zero while generalization error increase. Image from [8].

Capacity of a model is intrinsically defined in the set of functions from which the Learning Machine can choose (*representational capacity*). Practically, once you have chosen an appropriate set of function $f(\vec{x}, \omega)$ to the task you attend to solve, finding the best function can be a very difficult optimization problem (*see section 1.5*). Often, optimization find just one function that significantly reduces training error, not the best. These mean that the *effective* capacity of the learning algorithm may be less than the representational one.

A common approach on the selection of the learning algorithm is to follow the principle of "*the simple, the best*" also know as *Occam's razor*. This principle was formalized under the *statistical learning theory* by Vapnik [39].

Statistical learning theory provides different means of quantifying model capacity, as *Vapnik-Chervonenkis dimension* (VC dimension) which measures the capacity of a binary classifier. This allow at the statistical learning theory to make quantitative predictions and choose the optimal capacity. Even if the statistical learning theory is rarely used in practice, due to the difficulties to determine the capacity of complex algorithm, it provides a formal justification that machine learning can work under correct assumptions.

1.3.1 Regularization

Regularization provides a formal approach for controlling the capacity of a machine learning model to fit correctly the training data with good generalization performance. Under this approach we introduce an a priori knowledge in the form of the regularization used (also called *penalty term*). We can rewrite 1.2 adding *penalty term*:

$$R_{pen}(\omega) \;=\; R_{emp}(\omega) + \lambda\,\phi[f(\vec{x},\omega)] \tag{1.11}$$

where λ is the regularization parameter, which control the strength of the penalization, and $\phi[f(\vec{x},\omega)]$ is a functional that assign a non-negative value for each function supported by the Learning Machine. This functional is constructed to become smaller with the increasing smoothness of $f(\vec{x},\omega)$ and larger for non smooth functions. In this way, while $R_{emp}(\omega)$ minimizes the training error, the penalization term ensures a smooth choice in the functions set.

We have express, with this approach, a preference for one kind of function over another. This is a common practice in machine learning where we need to choose a model which fit with a specific task instead with a generic one. The so-called "no free lunch theorem" [42] ensures that there is no best machine learning algorithm, so we must choose a form of regularization (and indirectly make a priori assumptions on the form of functions) that is well-suited with the task we want to solve.

1.3.2 Model selection

Usually, machine learning algorithms come with a several numbers of settings (*hyperparameters*) that control the behaviour and capacity of the algorithm. These hyperparameters are not learned by the algorithm itself and we need a procedure to find the best ones according to our task. We can't learn the hyperparameters for two main reasons: they can be difficult to optimize or, more frequently, learning hyperparameters arise in overfitting. If learned on the training set, Learning Machine will choose always the hyperparameters with the maximum possible model capacity.

To solve this problem we need to use a *validation* set of data. Validation set can be build from the training set. It's very important that test set is not used in any way in the validation set.

Practically, a general procedure to split correctly the data set given a bunch of data is the follow:

- Divide the available data in two: *training* set and *test* set. Training set is used to learn the model, test set is used *only* for estimating the final risk of the learned model.

- Divide the training set in two: *learning* set and *validation* set. Learning set is used to learn model parameters with different set of hyperparameters that can be selected using the validation set to control the optimal capacity.

Cross-validation To reach a more statistical significance on the validation test results, a *k-fold cross-validation* method is often used. In the k-fold cross validation, the training set is splitted in k mutually excluded subsets $(K_1, ..., K_k)$. At the first step we use the subsets $(K_1, ..., K_{k-1})$ as learning set and the remaining one K_k as validation set. The procedure is iterated k times, leaving out each time a different K as validation set and using the remaining $k-1$ as learning set. The final result is an average across all the k validation errors.

Cross-validation is used also when the sample set are too small and splitting it in training and test set implies an excessive statistical uncertainty. In this case a cross-validation result is given as final error, using the entire sample test instead of training set.

1.4 Machine learning algorithms

A wide range of learning algorithms were developed across years. However, two of these are particularly interesting due to their efficiency. They are the support vector machines and the neural networks. The latter are the focus of this work, and a particular variation of them are explored in details in the next chapter.

1.4.1 Support vector machine

Support vector machine (SVM) is one of the most important machine learning algorithm, principally used for binary classification tasks. SVM uses linear function $\vec{w}^\top\vec{x} + b$ to divide the input space and outputs a positive or negative class when the linear function is positive or negative. SVM finds the best hyperplane which separate at best the input classes and using some input as support vectors to maximize the margin between the hyperplane and this vectors.

Support vector machines began particularly powerful when associated with the *kernel trick*. It can be proof that the linear function of the SVM can be written as:

$$\vec{w}^T\vec{x} + b \;=\; b + \sum_{i=1}^{m} \alpha_i \vec{x}^\top x^{(i)} \tag{1.12}$$

where $\vec{x}^{(i)}$ is a training example and $\vec{\alpha}$ is a coefficients vector.

This allow to replace the dot product with a *kernel* function $k(\vec{x}, \vec{x}^{(i)}) = \phi(\vec{x}) \cdot \phi(\vec{x}^{(i)})$ and we can use the following functions in our Learning Machine:

$$f(\vec{x}) \;=\; b + \sum_{i=1}^{m} \alpha_i k(\vec{x}, \vec{x}^{(i)}) \tag{1.13}$$

$\phi(\vec{x})$ is a function which maps the input vector in a high dimensional space. $\phi(\vec{x})$ and $f(x)$ have a linear relationship, as mush as $\vec{\alpha}$ and $f(\vec{x})$. Using the kernel function is equivalent to apply $\phi(\vec{x})$ to all inputs, then leaning the linear model in the new high dimensional space.

Kernel trick enhances the support vector machines performances for two reasons: allows to learn models that are not linear in the input space, transporting the data in a new high dimensional space, and implements a more computational efficient algorithm instead of the calculation of the dot product between $\phi()$ in the new space, even if it has infinite dimensions.

Usually is used a *Gaussian kernel*, also know as *radial basis function* (*RBF*):

$$k(\vec{u}, \vec{v}) \;=\; N(\vec{u} - \vec{v}; 0, \sigma^2 \vec{I}) \tag{1.14}$$

This kernel corresponds to a dot product in an infinite dimensional space, impossible to compute using the dot product.

Support vector machines are particular relevant in machine learning field because use the formal results of the statistical learning theory.

1.4.2 Neural Network

Neural network, also know as *Multilayer Perceptron*, is another common and powerful class of machine learning algorithms. A neural network is composed by layers. A layer can be written as

$$g_j(\vec{x}, \vec{w}_j) \;=\; s\Big(w_{j0} + \sum_{k=1}^{d} x_k w_{jk}\Big) \;=\; s_j(\vec{x} \cdot \vec{w}_j) \tag{1.15}$$

where $s(t)$ is an univariate activation function (usually a logistic sigmoid or hyperbolic tangent), $\vec{x}$ is the input vector and $\vec{w}$ the parameters matrix. A multilayer perceptron is a nested function of different layers. A three layers MLP can be written as

$$g_3(g_2(g_1(\vec{x}, \vec{w}_1), \vec{w}_2), \vec{w}_3) \tag{1.16}$$

g_1 and g_3 are commonly referred as *input* and *output* layer, respectively. g_2, and generically any other layer in the middle, are know as *hidden* layer.

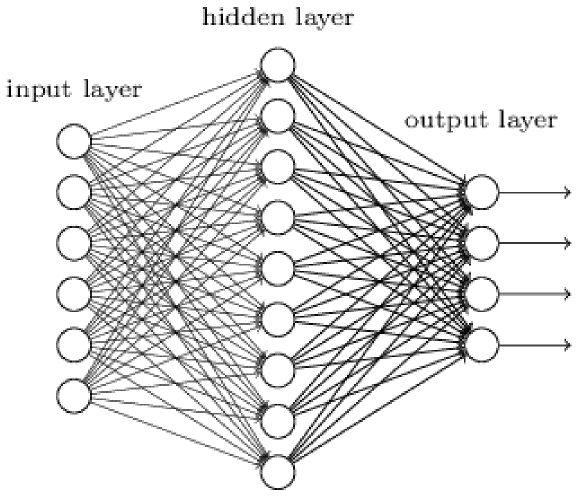


Figure 1.4: Representation of a MLP with one hidden layer. Each circle represent an unit. Image from [26].

While a neural network with a single layer output a linear function, it can be shown that a MLP with an arbitrary depth (depth indicate the number of layers) can approximate any continuous function. Unlike SVM, MLP can divide the input space in different folds and generalizes well for multi-class tasks.

MLPs are often called neural network because their structure is inspired by the neuroscience. Each hidden layer is a vector, every component of the vector plays the role of a neuron and we call it an *unit*. Each unit receives input from many other units and compute its own activation value (see fig. 1.4).

Error back-propagation As we will see in the next section, train a neural network require to find a local minimum in the parameters space (optimization). Optimization is done using the gradient of the function to minimize, the neural network in this case. To efficiently do this, an efficient optimization method was developed for neural networks, called *back-propagation*. A complete derivation of the back-propagation algorithm can be find in [6], in this section we report only the computational steps required to train a neural network.

Considering a two-layer neural network, where $\vec{w}$ and $\vec{v}$ are the parameter set of each layer, we can divide the back-propagation algorithm in two steps: *forward* and *backward*. At each iteration k, starting from an initial condition in which parameters are randomly initialized, in the forward step the output of each layer is calculated, while in the backward step the parameters are updated according to the true output expected from the network.

- Forward step

Input layer:

$$\begin{array}{rcl} a_j(k) & = & \sum_{i=0}^{d} x_i(k)v_{ij}(k) \quad j = 1, ..., m \\ z_j(k) & = & s(a_j(k)) \quad j = 1, ..., m \\ z_0(k) & = & 1 \end{array} \tag{1.17}$$

Output layer:

$$\hat{y} = s\Big(\sum_{j=0}^{m} w_j(k)z_j(k)\Big) \tag{1.18}$$

- Backward step

  Output layer:

$$\begin{array}{rcl} \delta_0(k) & = & \hat{y} - y(k) \\ w_j(k+1) & = & w_j(k) - \gamma_k \delta_0(k) z_j(k) \quad j = 0, ..., m \end{array} \tag{1.19}$$

Input layer:

$$\begin{array}{rcl} \delta_{1j}(k) & = & \delta_0(k)s'(a_j(k))w_j(k+1) \quad j = 0, ..., m \\ v_{ij}(k+1) & = & v_{ij}(k) - \gamma_k \delta 1 j(k) x_i(k) \quad i = 0, ..., d \;\; j = 0, ..., m \end{array} \tag{1.20}$$

where δ_k is a parameter called learning rate (see next section), and $y(k)$ the expected output of the network. The error back-propagation back-propagates the error from the output layer to the input one. The procedure described can be generalized at any network with an arbitrary number of layers.

1.5 Optimization: stochastic gradient descent

Optimization, in machine learning, refers to the task of minimizing the loss function by altering the parameters of the learning algorithm. In the case of neural network, this parameters are the element of the multiplication matrix w_{jk} also called weights. To minimize this function, we would find the direction in which it decreases fast as possible, using the gradient.

A common method is the *gradient descent* that decrease the loss function moving in the direction of the negative gradient by a step amount:

$$\vec{w'} = \vec{w} - \epsilon \nabla_{\vec{w}} L(y, f(\vec{x}, \vec{w})) \tag{1.21}$$

where ϵ is a positive scalar determining the step size, called *learning rate.*

With the increasing size of database and complexity of the learning algorithm, another method is often used: the *stochastic gradient descent* (*SGD*).

Stochastic gradient descent is an extension of the classical gradient descent which becomes more computationally efficient for large dataset.

Standard gradient descent algorithm evaluates the loss function, so calculate 1.3 becomes:

$$\nabla_{\vec{w}} R_{emp}(\vec{w}) \;=\; \frac{1}{m}\sum_{i=1}^{m} \nabla_{\vec{w}} L(y^{(i)}, f(\vec{x}^{(i)}, \vec{w})) \tag{1.22}$$

The computational cost of this operation is $O(m)$ where m is the number of examples.

Stochastic gradient descent defeats this problem with the insight that the gradient is an expectation, and this expectation may be approximately estimated using a small set of samples at time. Specifically we can estimate the expectation sampling at each step on a batch of examples drawn uniformly from the training set. This batch is usually fixed and chosen relatively small. The estimate of the gradient can be calculated as:

$$\vec{g} \;=\; \frac{1}{m}\sum_{i=1}^{m'} \nabla_{\vec{w}} L(y^{(i)}, f(\vec{x}^{(i)}, \vec{w})) \tag{1.23}$$

where m' is the number of samples in the batch. The stochastic gradient descent algorithm follow the estimated gradient to decrease the loss function and updates the weights:

$$\vec{w'} \;=\; \vec{w} - \epsilon\vec{g} \tag{1.24}$$

Increasing m will not extend the amount of training time needed and from this point of view the SGD run with a complexity of $O(1)$. The update cost using SGD does not depend on the training size and allows to use a very large training set, prohibitive to calculate with gradient descent.

Learning rate ϵ is a crucial parameter for the SGD algorithm. Learning rate can be a fixed parameter but is a common practice decrease it over time. Indeed a sufficient condition to guarantee convergence of SGD is:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \quad and \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty \tag{1.25}$$

where k is the iterations over time. To accomplish this is a common practice to decrease the learning rate over time.

Generally a learning rate too high introduces big oscillations in the learning curves, after all a learning rate too small can produces a very slow convergence.

To speeding up the convergence of the SGD in [29] was proposed a method, called *momentum*. Momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. The name momentum derives from a physical analogy, where the negative gradient plays the role of a force. More formally, the momentum algorithm introduces a variable $\vec{v}$ which is the analogous of the velocity: it is the direction and speed at which the parameters move through the parameters space.

THe SGD with the use of momentum became:

$$\begin{aligned} \vec{v}' &= \alpha\vec{v} - \epsilon\nabla_{\vec{w}}\left(\frac{1}{m'}\sum_{i=1}^{m'} L(y^{(i)}, f(\vec{x}^{(i)}, \vec{w}))\right) \\ \vec{w}' &= \vec{w} + \vec{v} \end{aligned} \tag{1.26}$$

where α is an hyperparameter.

An intuition about the introduction of momentum is shown in fig. 1.5.

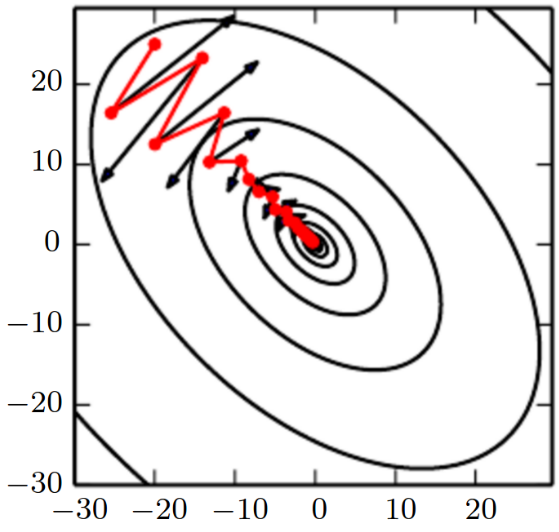


Figure 1.5: Representation of a gradient descent in a two dimensional parameter space with the use of momentum. Image from [8].

1.6 The curse of the dimensionality

The *curse of the dimensionality* is a phenomenon than occurs when the number of dimensions in the data becomes high. In high dimensional spaces, learning algorithm has many difficulties to collect enough samples with high density from finite data, necessary to obtain a smoothness and efficient function. The accuracy of function estimation depends on having enough samples within the neighborhood specified by the smoothness constraints. However, as the number of dimensions increases, the number of samples needed to give the same density increase exponentially.

The curse of dimensionality is caused to the geometry of high dimensional spaces:

- Sample sizes yielding the same density increase exponentially with dimension. If in 1-dimension n data points form a dense distribution, in d-dimensions we need n^d samples to reach the same density.

- As the dimensions increase, a very large amount of neighborhood are required to capture a small portion of the data. Data became very sparse in high dimensions spaces.

- Samples are almost closer to an edge than to every other points and almost every point is an outlier in its own projection.

With finite samples, the curse of the dimensionality becomes one of the principal problems in machine learning, in particular in images recognition in which the number of dimensions of the data can be very high.

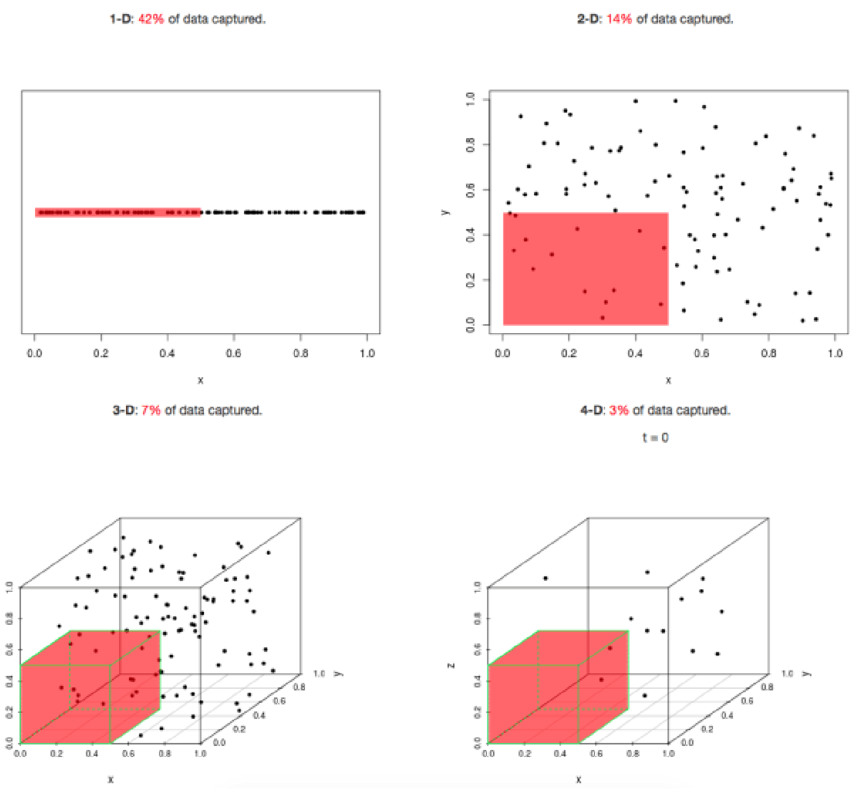


Figure 1.6: Curse of dimensionality. As the dimension increase, the amount of data captured with a fixed side hypercube decrease exponentially.

Chapter 2

Deep learning for images classification

The conventional machine learning algorithms, like the ones introduced in section 1.4, become limited when they process natural data in raw form, like images, due the high dimensionality of them. Shallow algorithms, to work well, require an expensive engineering process to design an efficient features extractor that can reduce the dimensionality of the data and representing correctly the differences between classes.

Deep learning methods introduce a powerful framework, called representation learning, which allows the machine learning algorithm to be fed with raw data and to automatically discovery the best features (*representation*) needed for classification (or other) task. We can describe a general deep learning algorithm as an algorithm with multiple level of representation, each one describing the representation at one level in a higher, and more abstract, one. Composing enough representation levels, the algorithm can amplify the aspects of the input which are discriminative for the task, and can learn a very complex function to handle these representations.

The most common form of deep machine learning algorithm is supervised learning, but unsupervised and semi-supervised algorithm are used to solve particular tasks. In this section we introduce the *convolutional neural networks* (*CNNs*), the most powerful feed-forward learning algorithm for images classification, and the *ladder networks*, a novel algorithm which enables the semi-supervised framework in the standard *MLPs* or *CNNs*.

2.1 Convolutional Neural Network

Convolutional neural networks are the most common deep learning algorithms used in the task of images detection and recognition. They are a particular variant of the MLPs, inspired by biology studies on visual cortex in mammals [15]. Visual cortex contains a complex arrangement of sensitive cells, each one responding to a small sub-region of the visual field (*receptive field*). These cells, acting as local filters over the receptive field, are able to exploit the spatially local correlation present in natural images.

This concepts was introduced with good results in machine learning in an early work on convolutional neural networks [22]. From this, a lot of improvement was added to the basic CNN architecture, allows to reach an human comparable performance on detection and recognition of objects in images. But, in simple, a CNN is a simply neural network that uses (also) convolution in place of general matrix multiplication.

The best way to describe the modern CNNs architectures is to visualize them as a stack of different layers. Combination of layers form a building block each learns a different representation. Typically the first building block, the one who reads the input raw data, represents the presence of edges at particular orientations, the second one learns to represent motifs which are a combination of edges, the third learns a combination of motifs which can constitute part of the image and so on [21].

The building blocks of a standard CNN are principally two: convolution block, formed by convolution, non-linearity and pooling layers, and classification block, formed by fully connected and non-linearity layers (MLPs).

In recent years a wide range of building blocks, and generally a wide numbers of architectures were introduced to solve more complex and challenging tasks in computer vision.

In the following section we introduce the common layers used in CNNs and describe a standard CNN architecture. Finally we introduced the recent develops in this field and take an overview on the state of the art architectures.

2.1.1 Layers

As previously introduced, the convolutional neural networks can be schematized as a stack of different layers, each one performing a different operation on the input of the previous layer.

Convolution Convolutional layer is the principal core of a CNN. In image processing, the discrete convolution operation on a two dimensional array I, like an image, with a kernel matrix K, can be written as:

$$S(i,j) \; = \; (K * I)(i,j) \; = \; \sum_m \sum_n I(i-m, j-n) K(m,n) \qquad (2.1)$$

This formulation is obtained from the standard discrete convolution formula, from the commutative property of this operation due to the fact that we have flipped the kernel. This is computationally efficient then the original one and can be proof which has the same results.

In CNN is used a similar formulation in which the kernel isn't flipped, this is commonly called *convolution* but is it in really a *cross-correlation* operation:

$$S(i,j) \;=\; (K * I)(i,j) \;=\; \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.2}$$

In CNN the appropriate values of the kernel are learned, so there is no necessity to flip it because the algorithm would simply learns the flipped version of it.

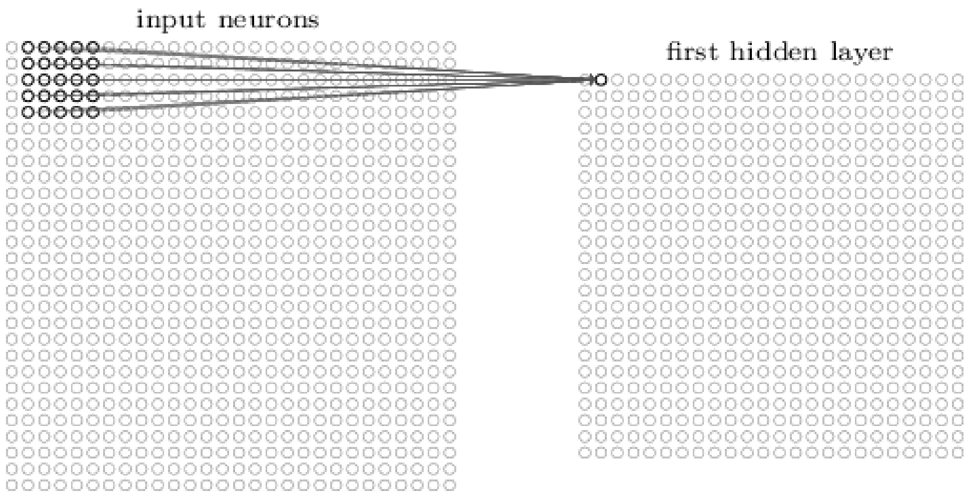


Figure 2.1: A representation of convolution operation with a $5x5$ kernel. Image from [26].

Convolution introduces three important ideas in a simple neural network which can improve the performance of the learning algorithm: *sparse interaction*, *parameters sharing* and *equivariant representations.*

Traditional neural networks, using general matrix multiplication, have a very dense interaction between units. For two layers MLP, for example, we have that every output unit interact with all the input units. Convolution operation introduces a sparse interactions, due to the fact that typically are used kernel smaller than the input. Compared to dense interaction we need to store fewer parameters reducing the memory requirements and the total numbers of operations, accelerating the computation time and improves the statistical efficiency.

In a traditional MLP, each parameter (*weight*) of the network is used only once when computing the output. Instead, in a CNN, each weight (the element of the kernel) is used over all the input. This means that during inference, due to the fact that more than one kernel is used in a CNN, the algorithm learns only one set of parameters instead that one for every location. This reduce dramatically the memory required and improves the statistical performance of the algorithm, introducing equivariance to translation. If we apply a translation on the input of the CNN, then applying the convolution, we obtain the same result convolving the input and then apply the translation. This becomes powerful when the input is an image, because convolution creates a $2D$ map of where certain features appear.

When using convolution in CNN usually we use a more general and complex operation than the one described. We can identify some differences.

Convolution, in neural network, is an application of many convolution operations in parallel. This means that we use a different numbers of kernels, each one extracting a different feature at many locations.

Often, the input images are a multi-channel matrix and not a simple grid of values. For example a standard RGB image is like a $3D$ matrix.

In some cases, we may want to skip over different location in the input, reducing the computational cost (but obtaining more unrefined features). This can be done introducing a *stride* in the convolution operation, that performs a downsample of the full convolution operation.

So, we can rewrite the convolution operation more generally:

$$Z_{i,j,k} = \sum_{l,m,n} \left[\vec{V}_{l,(j-1)xs+m,(k-1)xs+n} \vec{K}_{i,l,m,n} \right] \tag{2.3}$$

where $\vec{K}$ is the kernel tensor. $\vec{V}$ is the input with element $V_{i,j,k}$ with the value of the input unit in channel i, row j and column k, and $\vec{Z}$ is the output with the same format of $\vec{V}$.

We have assumed in this case that we can perform convolution along all the input positions, obtaining an output with the same dimensions of the input. In reality, without performing a zero-padding on the input, the output has a different dimensions depending on the kernel size, due to the impossibility to convolve kernel on the border pixels. In the general case, the output dimension is, along each axis:

$$W_{out} = \frac{W_{in} - K + 2P}{S} + 1 \tag{2.4}$$

where W_{in} is the pixel number of the input, K the filter dimension, P the number of padding pixels and S the stride dimension.

Using the convolution instead the general matrix multiplication introduces a strong prior, saying that the input of a convolution layer has only local interactions. This is a good assumption for natural images and makes convolution a powerful operation in the processing of them.

Non-linearity The non-linearity layer is the layer which performs the activation function, as introduced in section 1.4.2. A wide variety of activation function can be used with convolutional neural network. The most commons one are the logistic sigmoid activation function $\sigma(z)$ or the hyperbolic tangent activation function $tanh(z)$. Sigmoid was often used in the past but it tends to saturate across all the domain slowing down the training. Hyperbolic tangent working well, particularly with deep architecture, because allowing the net to use smaller weights. In the recent years, in deep architecture, is commonly used the rectified linear unit ($ReLU$):

$$s_{ReLU}(z) = max\{0, z\} \tag{2.5}$$

Due to its similarity with the linear function, $ReLU$ is easy to optimize and don't saturates when receives as input large values, allowing a faster convergence of the algorithm. Unlike sigmoid activation function, which can be used as output of the network, rectified unit doesn't have an output that can be interpreted as a probability. Usually in the output stage is used another non-linearity function called *softmax*, which naturally represents a probability distribution over a discrete variable with k possibly values, the numbers of classes:

$$s_{softmax}(z) = \frac{e^{z_i}}{\sum_i e^{z_i}} \tag{2.6}$$

Pooling Pooling layer performs an operation that calculates a summary statistic of some nearby outputs and replaces it at the different location of the output. Pooling can be of different types, but the commonly used performs a *max* or a *mean* operation. Max pooling, for example, reports the maximum output in a rectangular neighborhood (the pooling kernel) whereas mean pooling reports the average (or a weighted average) of the neighborhood in the kernel.

Pooling introduces an approximate invariant in the representation to small translation of the input and reduces, used with a stride greater than one, drastically the number of parameters across layers. This is known as pooling with downsampling and is commonly used to also improve the statistical performance of the network.

Like convolution, also pooling operation introduces a strong prior in the algorithm. In particular pooling works great when the assumption that the input (and the function that the layer learn, if pooling is used on this) is invariant to small translations is correct.

Fully-Connected Fully-connected layer is the classical neural network layer described in section 1.4.2 which perform the matrix multiplication. While the convolutional layer usually learns a good representation for the data, fully-connected layer has the

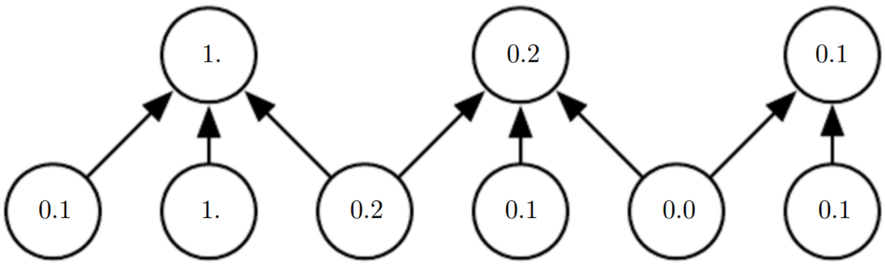


Figure 2.2: Max-pooling operation with stride of two on a one dimensional array. Image from [8].

task to compose this representation and learns a powerful discriminant function. In this sense, a stack of fully-connected layers is a MLP which discriminate the output features extracted by the convolution layers.

2.1.2 Standard architecture and training

As introduced before, the described layers can be stacked to obtain a deep CNN. There is no general rule to stack the layers, as we will seen in the following section, however some functional building blocks were identified and are commonly used to build a standard CNN algorithm which generally has a good performance on images recognition.

The first block is the convolutional one, it is composed by a convolutional layer, followed by a non-linearity layer, followed by a pooling layer. Usually are used small kernels for the convolution, with a stride of one pixel. $ReLU$ is commonly used as activation function and the most diffused pooling is the max-pooling with small kernel and stride with the same value of the kernel size.

The second building block is the classification block, formed by a stack of fully-connected layer with a non-linearity and is equivalent to a MLP. As output layer is used a fully-connected output with a number of outputs equal to the classes of the problem, followed by a softmax. A stack of a high number of fully-connected layers can produce an excessive capacity of the network and cause overfitting. To solve this problem, dropout was introduced along the fully-connected layers of this block [35]. During each inference step, each output of the non-linearity has a fixed probability (*dropout ratio*) to be multiplied by 0. This process is equivalent to stochastically substitute the MLP with sub-networks only during learning, restoring the full network during test.

A simple CNN can be obtained stacking one or more convolutional block, each with their parameters with a classification stage on the top.

During inference the output of the CNN is picked up by the loss function in order to perform the back-propagation. For a multiclass problem the cross-entropy loss is

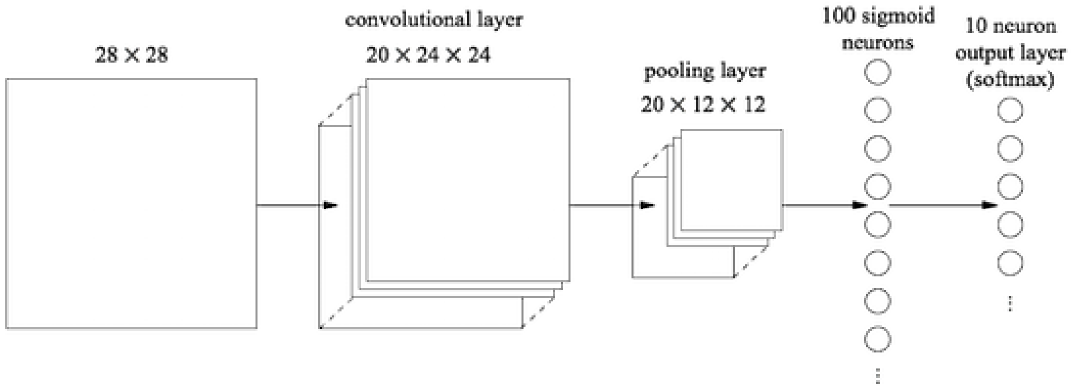


Figure 2.3: Architecture of a standard CNN with a convolutional block and a classification block. Image from [26]

frequently used:

$$L(y, s(z)) = -\sum_i y_i \log s_i \tag{2.7}$$

The optimization of the deep architecture like CNNs is commonly done using the SGD with momentum and the weights are randomly initialized to small near zero value, normally distributed, to speed up the convergence and allow to reach a better optimal minimum. As form of regularization is often used the L_2 regularization, also know as *weights decay.* At each iteration, during inference, the parameters of the networks are decreased of a small amount, reducing the risk of overfitting [25].

2.1.3 State of the art of CNNs

In recent years the deep learning research is focusing on method to improve the power of the CNN, introducing new architectures or optimizations. The trend, driven also to the increase in computational power of the GPUs, is to build networks very deep to increase the representational capacity and solve very challenging tasks in images recognition. This process is born with the famous *AlexNet* [19], a CNN architecture with five convolutional layer, and still going on with recent performing architectures which implement hundred of convolutional layers, like *GoogleNet* [38] or *ResNet* [11].

However, this recent architectures are more than a simple and deep stack of convolution and classification blocks, but implement others concepts that allow to train them efficiently and with high performances.

The improvements recently introduced in this field concern different aspect of the CNN, in this section we try to have a briefly overview on some of these, redirecting at

[9] for a more detailed description.

Most of this improvements concern new methods to re-arrange and connect different layers, like the idea behind GoogleNet. This architecture introduces a module, called *inception module*, which uses convolution with variable filter sizes to capture different patterns of different dimensions. More specifically inception module consists of one pooling layer and three types of convolution layers at each level of the network, each one working as dimensions reduction module which allows to stack a large amount of inception modules without the explosion in parameters size.

On other hand, the ResNet architecture makes use of shortcut connections between layers at different depth to allow an optimal propagation of the information across the network (*residual module*), using very small kernels but increasing exponentially the depth, until reach hundreds of modules stack with an efficient training.

This concept can be also joined together as reported in [37].

Starting from the idea proposed in ResNet architecture, others recent works observe that the depth can be increased using an efficient propagation of information across layers which allows to learn a powerful representation. Densely connected CNNs [13] redirect the output of each convolutional layer at the input of all the others one, taking advantage of recycle of features. Deep CNNs with stochastic depth [14], recently presented at ECCV2016, introduces the concept of training the ResNet randomly drops some modules. This allow to train efficiently, and with large improvements, architectures which implement thousands of layer .

Other operations to improve pooling were proposed which seems to better generalize than classical max-pooling or average-pooling operations. The most promising are the biologically inspired pooling [5], the mixed pooling [43] which combines both max and average operations, and the stochastic pooling [45] inspired by dropout operation, an so on.

Lot of improvement was proposed also in non-linearity layer, with new activation function. In addition to early described ReLU, a generalization of this, called *PReLU* [12] was proposed which introduces a learnable parameter on the slope of the negative values, allowing to reach a human level performance on benchmark dataset.

In a recent work [16] was also proposed a new layer, *Batch Normalization*, which allow to speed up training time of deep architecture and increase performance, normalizing the output of each layer.

To solve the problem of finding a good learning rate optimal for the architecture algorithm to optimize and speeding up the inference process, a high variety of optimizers, alternative at the SGD, were proposed. These methods are based on adaptive learning rates, so don't require a time expensive search parameters. They include ADADELTA [44] and ADAM [18] which are commonly used.

All these methods have improved dramatically the performance of convolutional neural networks, allowing to reach human-level performance on image classification, detection, segmentation and others difficult and more abstract tasks like pose estimation, action recognition, scene labelling and saliency detection.

2.2 Ladder Network

Semi-supervised learning allows to use both information from labeled and unlabeled examples during the inference to improve the performances of the learning algorithm.

The ladder network architecture, recently proposed [31], adds an unsupervised component to the supervised learning networks in order to use the unlabeled example enabling the semi-supervised learning in the existent architecture like MLPs or CNNs.

The unsupervised component of the ladder network is a stacked denoising autoencoder [41] which tries to reconstruct the input for all the examples while the labeled examples are used in a standard supervised way.

In this section we introduce briefly the ladder network architecture and his use in a CNN architecture.

2.2.1 Ladder network architecture

The ladder network algorithm uses deep denoising autoencoder (DAE) in which the noise is injected into all hidden layers. The loss function of the algorithm is a weighted sum of the cross-entropy loss, which tries to correctly classify each labeled input, and the reconstruction of the DAE at each layer of the decoder with a clean version of it, without noise addiction. This is accomplished using another encoder path with shared parameters which provides the denoised output at each layer.

The DAE has additionally laterals connections. Each layer of the noisy encoder is connected to its corresponding layer in the decoder, procuring additional information which allow the architecture to learn more abstract features [30]. The laterals connections needs to be combined at each level with the signal from the layer above in order to feed the successive layer. The general architecture of the ladder network is shown in fig. 4.5.

Take as example a MLP transformed in a ladder network, like the one reported in fig. 4.5. The ladder network was builded adding noise at each layer, after performing a normalization but before performing the batch-normalization correction (scale and shift) of the standard batch-normalization (see [16] or section 3.3). This part of the network is called the *clean encoder*. The decoder path is built from the normalized output of the last non-linearity, combined (with a particular *combinator function*) with the output of the last fully-connected, normalized and corrupted. This procedure is iterated all along the decoder path, replacing the fully-connected of the encoder and combining, at each level, the signal from the corresponding output of the layer of noisy encoder with the natural signal of the decoder.

Adding to this, ladder network implements an additional encoder which is a replica, sharing the same parameters, of the noisy encoder but without the addition of noise.

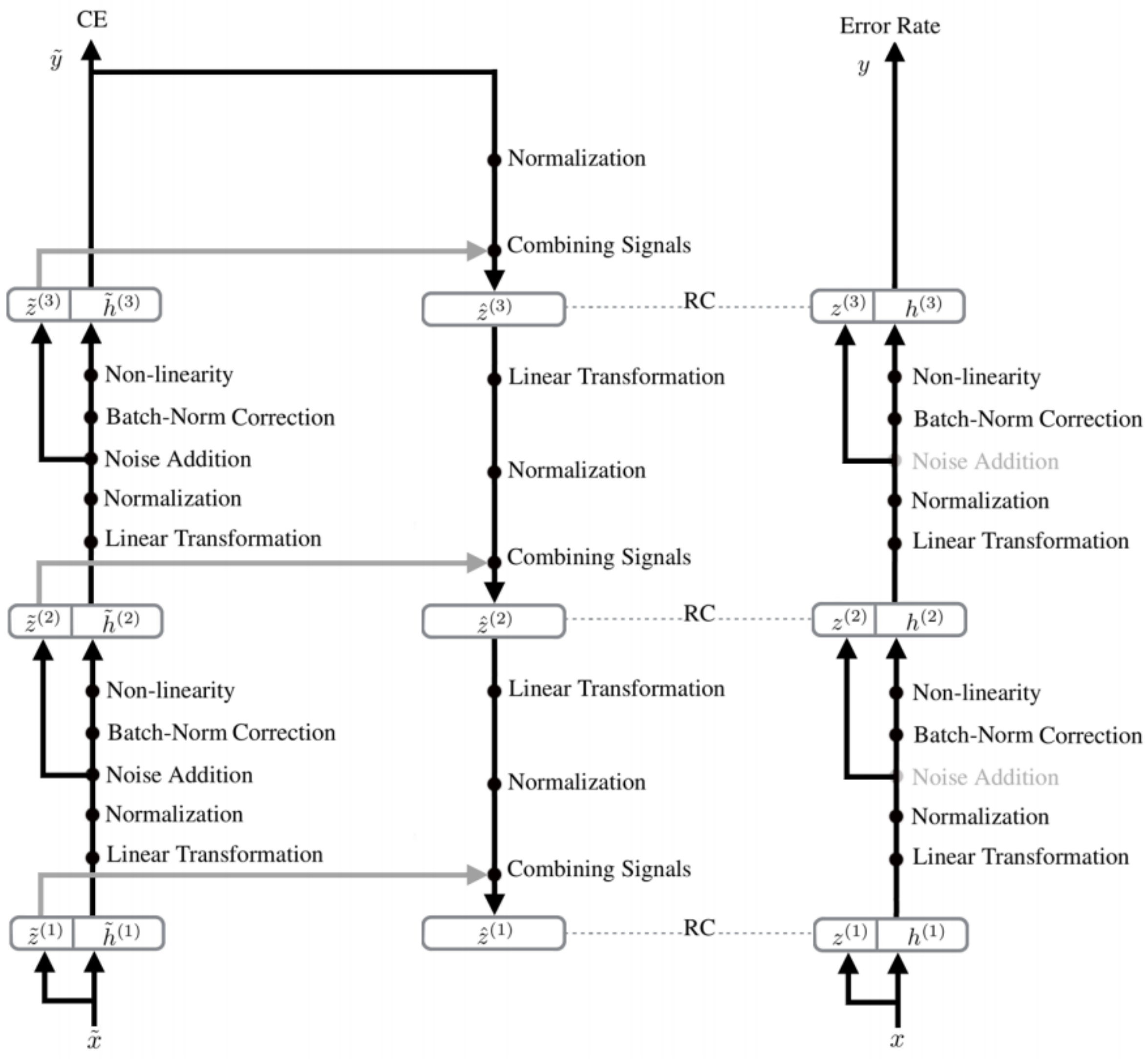


Figure 2.4: Ladder network architecture on a two layer MLP. On the left is reported the noisy encoder path, in the middle the decoder path and on the right the clean encoder path. Skip lateral connection are reported in gray while reconstruction loss is represented by the dotted lines.

During inference, the noisy encoder is used to classify the labeled input patches while the decoder path is used to reconstruct a denoised output of the noisy encoder from the output at each layer of the clean encoder. When an unlabeled example feeding the ladder network, the algorithm only learns to reconstruct it, without calculating the classification loss.

The ladder network architecture is used only during the inference process, during test only the base architecture, the clean encoder, is used.

An algorithmic description of the process can be found in 1.

As introduced before, ladder network uses a combinator function to mix the signal of the skip connections with the signal of the decoder. The combinator function proposed in [31] is the following:

$$\begin{aligned} g(\tilde{z}^{(l)}, u^{(l+1)}) \;=\; & b_0 + w_{0z} \odot \tilde{z}^{(l)} + w_{0u} \odot u^{(l+1)} + w_{0zu} \odot \tilde{z}^{(l)} \odot u^{(l+1)} + \\ & w_\sigma \odot Sigmoid(b_1 + w1z \odot \tilde{z}^{(l)} + w_{1u} \odot u^{(l+1)} + w_{1zu} \odot \tilde{z}^{(l)} \odot u^{(l+1)}) \end{aligned} \tag{2.8}$$

However, others combinator functions, simpler or more complicated, can be used instead 2.8 without get worse dramatically the ladder network performances.

Also if was born to be used in semi-supervised fashion, the ladder network architecture allows to reach a good representation at each layer's level even using only labeled data, resulting in better classification performance.

2.2.2 Augmenting CNN with ladder network

Ladder network enables a simple way to implement the semi-supervised framework on the existing feed-forward architectures. Nevertheless implementing the entire ladder network architecture in the existing deep CNN can be a non-trivial problem due to the decoder path.

Write an autoencoder for a CNN can be quite difficult for many problems. While a deconvolution operation is well-defined, there is not a standard way to implement the un-pooling operation. Moreover the presence of the decoder implicate the double of the weight parameters that can be hard to handle in memory.

In [31] was proposed a simple way to enable a semi-supervised framework in a CNN with the ladder algorithm. The idea is to use only the reconstruction cost on the top, setting the others to zero. In this way the decoder path become useless and can be omitted, and the ladder network tries only to denoise the output layer. More formally this architecture, called ladder-Γ is obtained setting to zero all the λ_l in 1.4, except the one for $l = L$.

A ladder-Γ CNN architecture can be constructed from a standard CNN adding the noise at each convolutional and fully-connected layer, using the batch normalization as reported in 1.4. The output of the last fully-connected, corrupted, is combined with the

Algorithm 1 Ladder network algorithm from [31]

```
**Noisy encoder path:**
$\vec{h}^{(0)}_{noisy} \leftarrow \vec{z}^{(0)}_{noisy} \leftarrow \vec{x}(n) + noise$
**for** $l = 1$ **to** $L$ **do**
    $\vec{z}^{(l)}_{noisy} \leftarrow batchnorm(\vec{W}^{(l)}\vec{h}^{(l-1)}_{noisy}) + noise$
    $\vec{h}^{(l)}_{noisy} \leftarrow activation(\gamma^{(l)} \odot (\vec{z}^{(l)}_{noisy} + \vec{\beta}^{(l)}))$
**end for**
$P(y_{noisy}|\vec{x}) \leftarrow \vec{h}^{(L)}_{noisy}$

**Clean encoder path:**
$\vec{h}^{(0)} \leftarrow \vec{z}^{(0)} \leftarrow \vec{x}(n)$
**for** $l = 1$ **to** $L$ **do**
    $\vec{z}^{(l)}_{pre} \leftarrow \vec{W}^{(l)}\vec{h}^{(l-1)})$
    $\vec{\mu}^{(l)} \leftarrow batchmean(\vec{z}^{(l)}_{pre})$
    $\vec{\sigma}^{(l)} \leftarrow batchstd(\vec{z}^{(l)}_{pre})$
    $\vec{z}^{(l)} \leftarrow batchnorm(\vec{z}^{(l)}_{pre})$
    $\vec{h}^{(l)} \leftarrow activation(\gamma^{(l)} \odot (\vec{z}^{(l)} + \vec{\beta}^{(l)}))$
**end for**
$P(y_{noisy}|\vec{x}) \leftarrow \vec{h}^{(L)}_{noisy}$

**Decoder path:**
**for** $l = L$ **to** $0$ **do**
    **if** $l = L$ **then**
        $\vec{u}^{(L)} \leftarrow batchnorm(\vec{h}^{(L)}_{noisy})$
    **else**
        $\vec{u}^{(l)} \leftarrow batchnorm(\vec{V}^{(l+1)}\vec{z}^{(l+1)}_{decoder}))$
    **end if**
    $\forall i \ : \ z^{(l)}_{i,decoder} \leftarrow combinator(z^{(l)}_{i,noisy}, u^{(l))}_{i}$
    $\forall i \ : \ z^{(l)}_{i,decoder,BN} \leftarrow \frac{z^{(l)}_{i,noisy} - u^{(l)}_{i}}{\sigma^{(l)}_{i}}$
**end for**

**Cost function:**
$C_{classification} \leftarrow 0$
**if** $label(n)$ **then**
    $C_{classification} \leftarrow CrossEntropy(\vec{y}_{noisy}|label(n))$
**end if**
$C_{total} \leftarrow C_{classification} + \sum_{l=0}^{L} \lambda_l \|\vec{z}^{(l)} - \vec{z}^{(l)}_{decoder,BN}\|^2$
```

output of the softmax, normalized. During inference we try to reconstruct this signal from the last fully-connected output of the clean version of the CNN, both for labeled and unlabeled examples.

This architecture doesn't exhibit the full potential of the full ladder network architecture, but due to its implementation's simplicity enables a powerful framework on standard supervised CNN.

Chapter 3

Problem and methodology

3.1 Classification of nuclei in routine colon cancer histology images

The advent of whole slide digital scanners has resulted in a substantial amount of clinical and research interest in digitization of tissue slides (digital pathology). Digitization of tissue slides facilitates many communication tasks: from transfer of images between distant locations, for a remote pathology consult, to the improvement of clinical workflow, reducing the space amount needed for storing and the risk of lose or break the slides.

Digital pathology has been transformative for computational imaging research, in particular recently researches are focusing on developing new algorithms specifics for whole slide images. In particular the use of machine learning algorithms to approach segmentation, detection or classification of histological primitives, like cell's nuclei, seems to be very promising.

For a number of disease (like cancer) it has been long recognized that the underlying differences in the molecular expressions of the disease tend to manifest as tissue architecture and nuclear morphologic alterations. This allows to build algorithms for automated tissue classification, disease grading and precision medicine using the slide images. Recent papers ([23], [40]) suggest which computer extracted image features from biopsy tissue specimens can help predict the level of disease aggressiveness. These features can be handcrafted, for the particular problem, or unsupervised, like in the case of CNNs. The firsts provide more transparency but can be very hard to engineer and require a deep understanding of the disease and its manifestation in images. The unsupervised features, while produce a lack of interpretability, can be applied quickly to any domain or problem without understanding the manifestation of the disease in morphological structures.

One of the cancer types that can be addressed with the study of morphological structures is the colorectal carcinoma. Colorectal carcinoma is one of the most common human cancers, and one of the principle causes of cancer-related death. About 96% of all colorectal cancers are adenocarcinomas which originate from the epithelial lining of glandular tissue in the colonic mucosa. These adenocarcinomas could be curable when they are early detected and treated, but often many of them can be undiagnosed or misdiagnosed. Therefore there is the needed to develop of more precise early diagnostic methods before the growth of the tumor.

Many methods were developed across the years to deal with this problem, however, even with the costs and risks associated, the endoscopic biopsy hematoxylin and eosin stained (*H&E*) is still the best standard for local screening. H&E is one of the principle stains in biology, in particular in medical diagnosis where allow to stains cell's nuclei in blue and cytoplasm connective tissue in pink. An H&E histopathological images can contain thousands of cell's nuclei, and the spatial arrangement of all the heterogeneous cell types, in particular of the inflammatory cells, has also been shown to be related to cancer grades. Therefore, the correct localization and classification of nuclei opens the doors to a better understanding of this type of cancer and to a better diagnosis of it.

In the case of colorectal adenocarcinoma, dysplastic and malignant epithelyal cells often have irregular textures and a high variability in the appearance of the same type of nuclei. Adding to this, inferior image quality may arise due to poor fixation and poor staining during the tissue preparation process or autofocus failure during the digitization of slides. Both these factors make classification and localization of individual nucleus a hard task. [24] [1]

In this work we will focus on classification of cell's nuclei in colon cancer histology images using deep learning algorithm based on CNNs. A complete diagnostic method includes both detection stage of nuclei and classification stage of the same, however in this work we suppose to have a detection stage which finds the exact location of the center of all the nuclei. Our intent is to develop a method able to recognize the type of the nuclei detected, improving a pre-existing algorithm developed with this intent.

3.1.1 Database

In this work was used a database[1] of histology images of colorectal adenocarcinomas packaged by the Department of Computer Science, University of Warwick. The database was composed of 100 hematoxylin and eosin H&E stained histology images, each with size of $500x500$ pixels, *RGB* at 24 bit resolution (8 bit each channel). The images were cropped from non-overlapping areas of 10 whole-slide images from 9 patients, with a pixel resolution of $0.55\mu m$ using a $20X$ optical magnification. The whole-slide images were obtained using an Omnyx VL120 scanner. The cropping areas were selected to represent a variety of tissue appearance from both normal and malignant regions of the

[1]Database is available at `http://www2.warwick.ac.uk/fac/sci/dcs/research/combi/research/bic/data/crchistolabelednucleihe`

slides. This also comprises areas with artifacts, over-staining, and failed autofocusing to represent outliers normally found in real scenarios.

Manual annotation of nuclei was conducted mostly by an experienced pathologist and partly by a graduate student under supervision of and validation by the same pathologist. A total number of 29756 nuclei were marked at the center, 22444 of those have an associated class label: ephithelial, inflammatory, fibroblast and miscellaneous. The inflammatory nuclei includes lymphocyte plasma, neutrophil and eosinophil but this fine classification was not specified. The miscellaneous class includes cell type that do not fall in the others three class and contains adipocyte, endothelium, mitotic figure, nucleus of necrotic cell, etc. [34].

Example images with the correspondent annotations are reported in fig. 3.1.

3.2 Dataset creation

The images described in section 3.1.1 are *bitmap* images which comes with associated *mat* files, one for each class, containing the pixels coordinates of the center of the nuclei. To create the different sets to train a Convolutional Neural Network we need to extract a patch containing enough information to allow a performing classification. As reported in [34] we need a $27x27$ pixel patches nuclei-centered to feed the CNN architecture. In order to perform a data augmentation with resizes and rotations, we extract $42x42$ pixel patches (see section 3.4.1 for more details).

To extract nuclei-centered patches along the edges of the images we perform a mirroring reflection, an approach which eliminate the discontinuities introduced by a zero-padding which can creates artifacts [32]. In fig. 3.2 are report some example of the extracted patches.

From the entire set of labeled patches were extracted randomly 30% of them to build the *test set*. The test patches was cropped at the size of $27x27$ pixel. The remaining labeled patches, randomly shuffled, constitute the *training set*.

To perform a *3-fold cross-validation*, patches in *training set* were divided in three folds of the same size. Alternatively, two of them were joined together to form a *learning set* and the remaining one the *validation set*. In this way we obtained three learning sets and three validation sets, these one opportunely cropped at $27x27$ pixel size.

In table 3.1 is reported a complete situation of the class distribution over training and test sets.

An extra training set with the unlabeled examples was prepared. This was done inserting randomly the unlabeled patches in the training set described above. The *3-fold* procedure was repeated in a similar way. The unlabeled patches are 7312 and constitute the 31.76% of the training set, now consisting of 23023 examples. In this work, we have explained the use of this training set instead of the standard one.

All the datasets were stored in *lmdb* files. Epithelial, fibroblast, inflammatory and

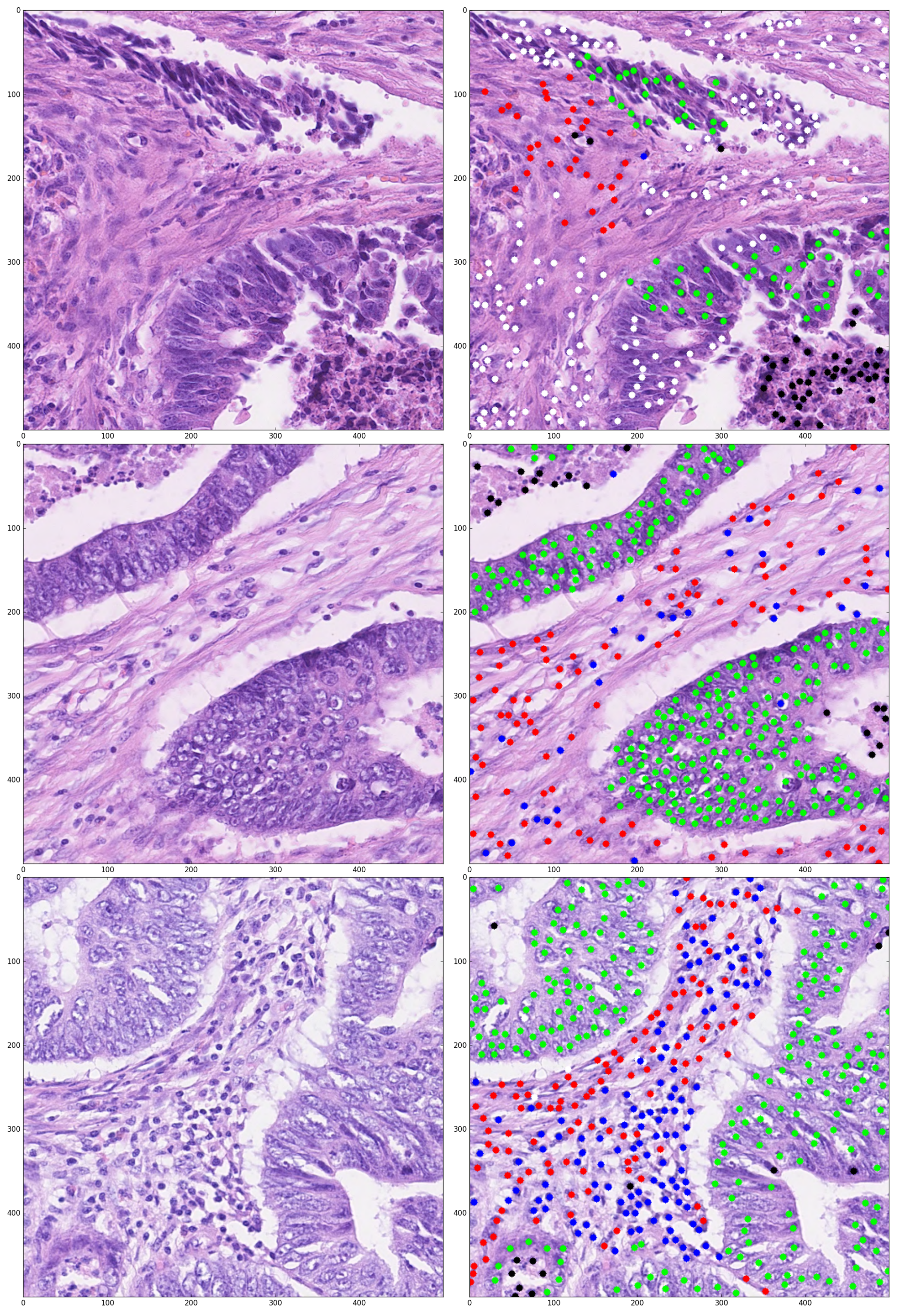
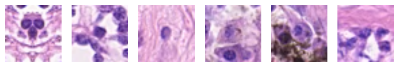


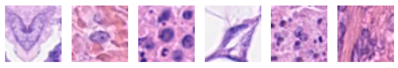
Figure 3.1: On the left, three different examples from the database. On the right the same images with the marked nuclei. The white dots indicate the unlabeled nuclei, the red ones indicate the fibroblasts nuclei, the green dots the epithelial nuclei, the blue dots the inflammatory nuclei and the black dots the miscellaneous nuclei.

(a) Epithelial nuclei

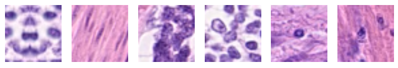
(b) Fibroblast nuclei



(c) Inflammatory nuclei



(d) Miscellaneous nuclei



(e) Unlabeled nuclei

Figure 3.2: In the images from (a) to (d) are reported some randomly extracted 42x42 pixels patches divided by their class. In the last row are shown the unlabeled example. On the first column is visible the effect of the mirroring reflection policy.

miscellaneous have, respectively, a class label from 0 to 3. A fake class label 4 was used for the unlabeled examples, to have a consistent lmdb file.

The entire procedure of dataset creation was made using *python 2.7* with additional opencv[4] and sklearn[27] libraries.

3.3 Build a CNN

In this section are presented the different CNNs architectures used in this work with their relative implementation details. To build the different architectures was followed a step-improve approach: starting from a ground architecture, different improvements were introduced and tested to enhance the performance of the network. The obtained architectures were grouped in three categories.

The convolutional neural networks were implemented using Caffe[17], a professional deep learning framework optimized for feed forward networks like CNNs.

Ground architecture

The ground architecture implemented is the basic one reported in[34]. The network consists of two convolutional layers, each followed by a pooling operation, and three

	Training set	Test set	Total	Relative %
Epithelial nuclei	5412	2310	7722	34.41
Fibroblast nuclei	3955	1757	5712	25.45
Inflammatory nuclei	4885	2086	6971	31.06
Miscellaneous nuclei	1458	580	2038	9.08
Total	15710	6733	22444	
Relative %	70.00	30.00		

Table 3.1: Dataset examples distribution over training and test sets.

fully-connected layers. The detailed architecture, with the dimensions of each filter and layer output, is shown in table 3.2.

Layer type	**Filter dimensions**	**Output dimensions**
Input		27 *x* 27 *x* 3
Convolutional	4 *x* 4 *x* 3 *x* 36	24 *x* 24 *x* 36
Max-Pooling	2 *x* 2	12 *x* 12 *x* 36
Convolutional	3 *x* 3 *x* 36 *x* 48	10 *x* 10 *x* 48
Max-Pooling	2 *x* 2	5 *x* 5 *x* 48
Fully connected	5 *x* 5 *x* 48 *x* 512	1 *x* 512
Fully connected	1 *x* 1 *x* 512 *x* 512	1 *x* 512
Fully connected	1 *x* 1 *x* 512 *x* 4	1 *x* 4

Table 3.2: Grond architecture. In the table is reported the filter dimension and output of each layer.

A rectified linear unit ($ReLU$) was used as activation function after each convolutional layer and after the first two fully connected layers. After the last fully connected layer, a *softmax* nonlinearity is used previous to feed the loss function. The loss function used is the cross-entropy loss (also know as *multinomial logistic*). All the weights of the network were initialized with Gaussian random numbers with 0 mean and standard deviation of 0.01, while all the biases were set to 0. In order to avoid overfitting, dropout was implemented after the first two fully connected layers, with a dropout ratio of 0.2.

A graph of the entire architecture can be seen in fig. 3.3.

Improved architectures

Improved architectures were obtained changing some parameters or adding different layers to the ground architecture. In particular we have explored the Xavier initialization method and the use of the Batch Normalization layer.

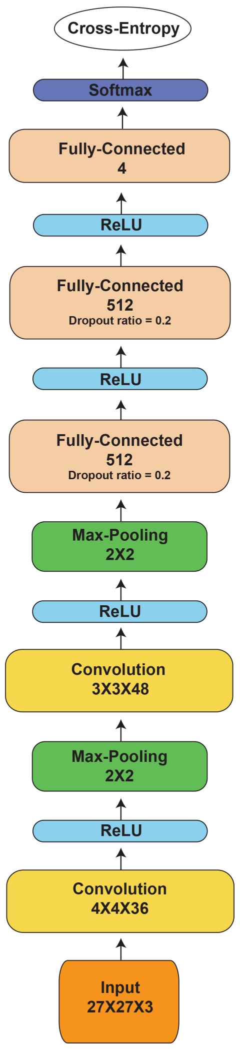


Figure 3.3: Architecture of the ground convolutional network. The weights of the convolutional and fully-connected layer are initialize with a zero-mean normal distribution with a standard deviation of 0.01. This architecture was also trained with a Xavier initialization of the weights.

Xavier initialization To improve the performance, we initialize the ground network architecture presented earlier, with a different random number generator. We use a Xavier initialization method which allow a better and faster convergence during training, as reported in [7].

Xavier initialization consist in randomly sampling from a uniform distribution and fill each layer. The distribution depends on the number of inputs of each layer, as following:

$$W_{i,j} \;=\; U\Big[-a, a\Big] \quad where \quad a = \frac{1}{\sqrt{n}} \tag{3.1}$$

n is the size of the previous layer, the number of columns of W.

Caffe framework implements a little bit different version of this, in which $a = \sqrt{\frac{3}{n}}$.

All the weights of the network were initialized using this, while the biases were maintain to 0.

Batch normalization As second improvement, starting from the previous network, batch normalization layers were added in the architecture. As reported in [16], batch normalization reduces the *internal covariate shift* of the network, allowing a faster convergence and the achievement of a better local minimum [16].

In multilayer architectures, the training is complicated by the fact that the inputs to each layer are affected by the preceding one, small changes in the parameters of a layer can produce an amplified effect if the network became deeper and the layers need to continuously readjust to the new situation. This change in distribution of layer's inputs is called *covariance shift.* Batch normalization tries to reduce this shift fixing the means and the variances of the layer's input.

$$\begin{aligned} BN_{\gamma,\beta}(x_i) \;&=\; \gamma\hat{x_i} + \beta \quad where: \\ \hat{x_i} \;&=\; \frac{x_i - \mu_B}{\sigma_B} \end{aligned} \tag{3.2}$$

Every input x_i is normalized (in each dimension) using the mean calculated across a batch of m elements and the variance of the same batch. The normalized input is shifted and scaled with γ and β parameters, which are learned during inference. These parameters guarantee to maintain an optimal representational capacity also after normalization.

We use batch normalization in our network at the end of every convolutional or fully connected layer (or equivalently before each ReLU and Softmax layer). It was done in Caffe combining the Batch Normalization layer, which performs only the normalization step, with the Scale layer which enables the learnable scale and shift parameters.

In [16] is reported that batch normalization can regularizing the model and replace, in some cases, the dropout. We choose to maintain the dropout regularization in our architecture. The architecture is reported in fig. 3.4

Augmented architecture

Augmented architecture was obtained attaching external modules to the CNN only in the training phase, to have a better inference. In particular we explore the ladder network architecture which is expected to improve the representation made by the CNN making use of unlabeled data.

Ladder-Γ Starting from the improved network with batch normalization, we implement a custom version of the ladder-Γ network described in section 2.2. First of all we add a Gaussian noise, with mean zero and adjustable standard deviation, to the input and after each normalization layer, before the scale and shift learnable parameters. As reported in [8] the noise has a strong regularization effect, following this we have removed dropout from our architecture. We call this part of the network the *noisy encoder*. In parallel the same architecture was replaced without the use of the Gaussian noise in order to have a denoising reconstruction, this form the *clean encoder*.

The combinator function implemented is a basic version of the one presented in section 2.2. Following [28] we use a simple addition which is reported to do not affect dramatically the performance of the ladder network architecture. Therefore we sum the output of the last fully connected layer of the *noisy encoder*, after noise addition but before the scaling and shifting, with the output of the softmax of the same encoder, normalized. The reconstruction was done between the last normalized fully connected output of the *clean encoder* and the normalized combinator function. An euclidean loss was used as reconstruction function with an adjustable cost (λ). A representation of the entire architecture is reported is fig. 3.5

$$Reconstruction \;=\; \|Normalized(x_{lastFC}) - Normalized(Combinator)\|_2^2 \tag{3.3}$$

During inference the cross-entropy loss ($C.E$) is calculated only in the noisy encoder while only the encoder path is used in test phase. The final loss function, during training, is a weighted sum of the two loss:

$$Lossfunction = C.E. + \lambda \cdot Reconstruction \tag{3.4}$$

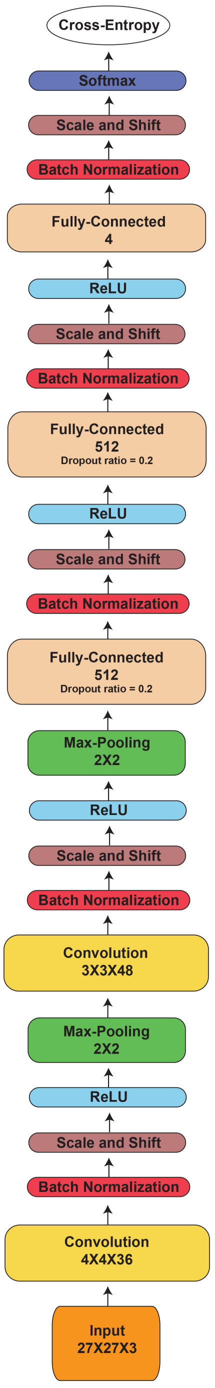


Figure 3.4: Architecture with the use of batch normalization layer and scale and shift layer to reproduce the correct behaviour of batch normalization correction.

This network allows us to use also unlabeled examples in a semi-supervised learning. To do this, the architecture was simply modified to ignore the fake labels in the *C.E.*. When an unlabeled example feed the network, only the reconstruction loss contributes to the final loss.

3.4 Training

In this section was described the entire training procedure followed in this work. In particular we introduce the different data augmentation methods used and all the different parameters setting.

3.4.1 Data Augmentation

Data augmentation is a common practice in image recognition and consists in generate fake data in order to have a bigger training set. A large amount of data allow the learning algorithm to generalize better and avoid overfitting. There are many techniques to generate fake data: they can be generated randomly knowing the statistical distribution (*generative model*) or can be generate from the available training set performing transformation on it. The latter is the common approach used, principal for his simplicity and for the possibility to introduce the wanted invariance performing a specific transformation. If we want to improve the generalization error on translated images we can augment our data introducing a shifted version of them, and so on. [8]

With this approach is important to not introducing transformations that could change the correct class. For a classification task in which the size of the represented object is a discriminant, introduce a scale transformation can compromise the performance of the algorithm.

We perform data augmentation "on the fly" (fig. 3.6), the augmented training set wasn't really created and saved in a file, but each sample was transformed iteratively and randomly each time before feeding the network. This avoid the data store problem and allows a better generalization, reducing the probability than the algorithm look two times exactly the same sample during inference.

Standard The first data augmentation implemented is pretty similar to the one reported in [34]. Each example was transformed in HSV domain. The sample was shifted along x and y axis of a random pixel unit number between -5 and 5, than the sample was randomly flipped along X and/or Y axis and rotated by 0, 90, 180 or 270 degree. Variability in color distribution is commonly founded in histology, so this was perturbed multiplying HSV channels separately by random numbers sampled in a uniform range $r_{channel}$. In particular we use $r_H \in [0.95, 1.05]$ and $r_S, r_V \in [0.9, 1.1]$. Samples were cropped to $27x27$ pixel dimension from the original $42x42$ pixel size and finally re-mapped in RGB domain.

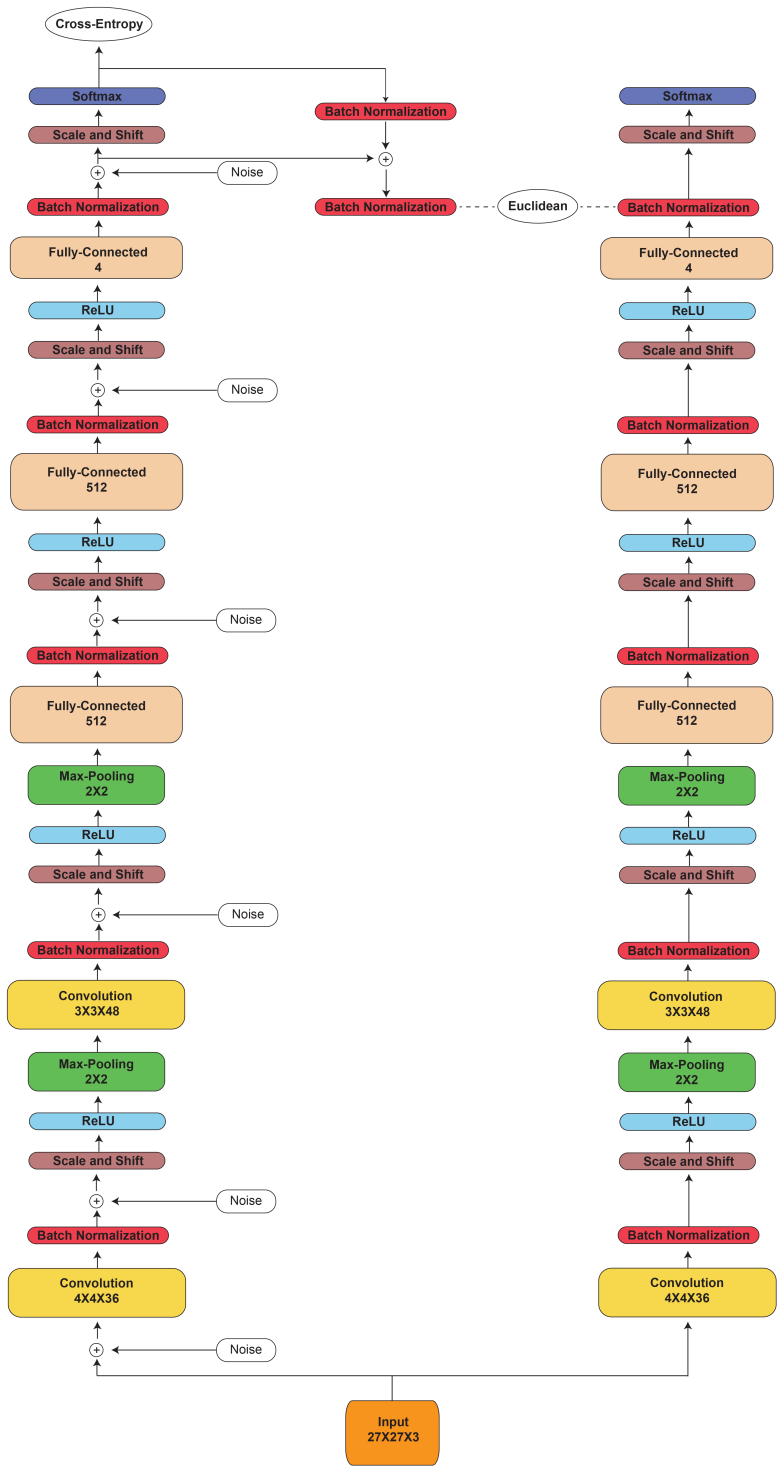


Figure 3.5: Ladder-Γ network architecture. On the left side is reported the noisy encoder path on the right the clean encoder path.

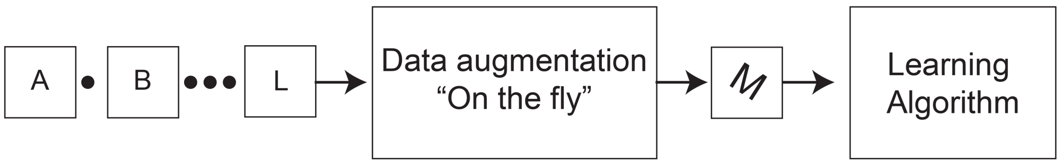


Figure 3.6: Data augmentation "on the fly", representation.

Improved A second data augmentation procedure was implemented to achieve a better generalization. In particular we introduce a more generally rotation and scaling and we used a different colors perturbation.

Each $42x42$ pixel patch from the training set was randomly resampled of a random value between $[0.9, 1.1]$, introducing a scaling, but maintaining the same patch dimensions. The patch was then rotated in a range between $[0, 360]$ degrees and flipped along X and/or Y axis.

Color perturbation was done performing a PCA on the set of RGB channels in the training set (equivalently in the learning sets). At each sample was added a multiple of the found principal components, randomly drawn from a zero-mean Gaussian with standard deviation of 0.1, of the found principal components, with magnitudes proportional to the corresponding eigenvalues times a random variable drawn from a Gaussian with mean zero and standard deviation 0.1 [19].

Therefore, each pixel of the image was transformed adding the following quantity:

$$[\vec{p}_1, \vec{p}_2, \vec{p}_3][\alpha_1\lambda_1, \alpha_2\lambda_2, \alpha_3\lambda_3,]^\top \tag{3.5}$$

where $\vec{p}_i$ are the eigenvectors, λ_i the eigenvalues of the $3x3$ covariance matrix and α_i the random values extracted from the gaussian distribution. We do not perform shifting due to the nuclei-centered patches.

Finally, the patches were cropped at the final size of $27x27$ pixel.

3.4.2 Training parameters

To avoid a very time-consuming hyperparameters search, due to the fact which train a deep CNN can require different hours of computation, some of them have been chosen and fixed, also in order to observe the contribution of the introduced modifications.

In the choice of the optimizer and its parameters we are inspired from [34].

Caffe framework use the concept of *iteration* instead of the most usual *epoch*, where an iteration is the forward computation of a batch. We use a stochastic gradient descent with momentum, starting from a learning rate of 0.01 with a momentum of 0.9 for the firsts 5000 iterations. At this iteration the learning rate was reduced to 0.001 until iteration

8000 where is reduced again to 0.0001. The learning procedure stops at iteration 10000. This procedure allows a fast but suboptimal convergence on the first part of inference, followed by two fine tuned searches which improve the convergence and find a better local minimum.

A batch size of 256 is used, a compromise to the memory required available and the use of a batch size large enough to smooth the noise during train [3]. Moreover, Caffe framework is optimized to compute batch of size 2^n.

As regularization term we use a weight decay of 0.0005.

All the patches were scaled by a 256 factor before feeding the network to normalize the values and achieve a faster convergence of the algorithm.

Chapter 4

Results

In this chapter are presented the results on the CNNs architectures described in section 3.3. In the model selection section the architectures were cross-validated training on learning sets and testing on validation sets. As measure are reported the loss error, on validation and learning, and the accuracy. This latter was choose as measure for the best model. The accuracy evaluates the percentage of corrected classified examples.

In test section are reported the measures on the test set of the best model discovered in model selection, trained on training set. In addition to accuracy, the weighted average F1 score and the *multiple area under curve* score, a multiclass generalization of binary *area under curve* (AUC), are reported.

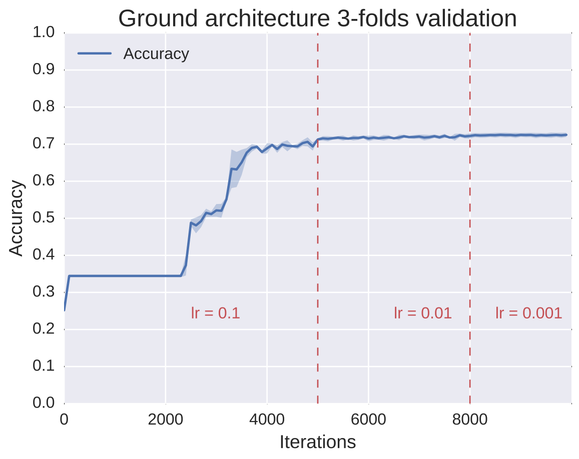
4.1 Model selection

In this section are presented the results of the cross validation on the different architectures and training methods introduced in the previous chapter. The cross-validation was run on 3-fold and the results were averaged on these different runs.

4.1.1 Ground architecture

Standard data augmentation

The ground architecture, described in section 3.3, was trained as first with the standard training setup introduced earlier. This model has some difficult to converge across the firsts 2000 iterations, where the network identifies all the test samples as *epithelial nuclei*, the class with the most number of examples in the dataset. Due to this, the accuracy is fixed at 34%. Around the 4000 iterations the learning curve starts to be stable and it is finetuned by the decrease of the learning rate at 5000 and 8000 iterations. We can see that we have a good capacity due to the absence of overfitting. At its best the curve



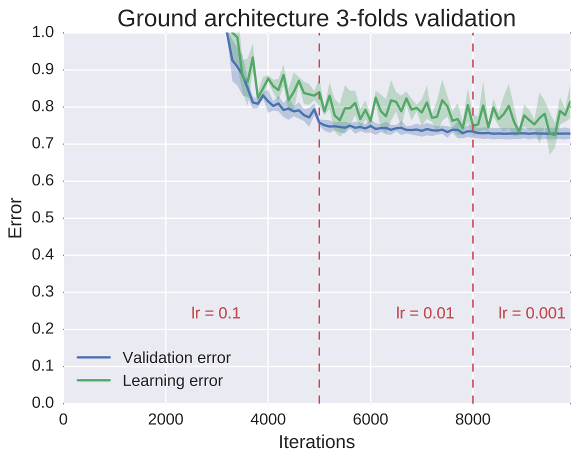
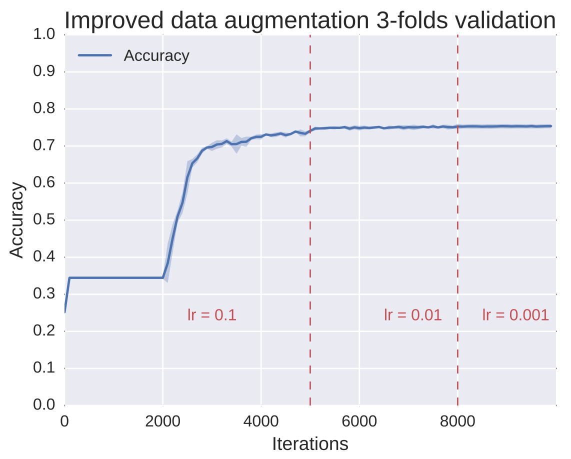


Figure 4.1: Learning curve of the ground architecture trained with the standard data augmentation procedure.

reaches an accuracy of 0.725 ± 0.006 at iteration 8600. The correspondent validation error is 0.73 ± 0.01 and the learning error is 0.78 ± 0.05. We can see the learning curves in fig. 4.1.

Improved data augmentation

The same CNN architecture was trained with a more sophisticated data augmentation described in the section 3.4. The results are shown in fig. 4.2. We can see that the curve has a similar behaviour to the precedent one, with an improvement in particular in the generalization error, as expected. This allow a better accuracy performance on the validation set. This model reaches its best at iteration 9500 with an accuracy of 0.754 ± 0.006 and a validation error of 0.65 ± 0.01 and a corresponding . The learning error is instead of 0.65 ± 0.04.



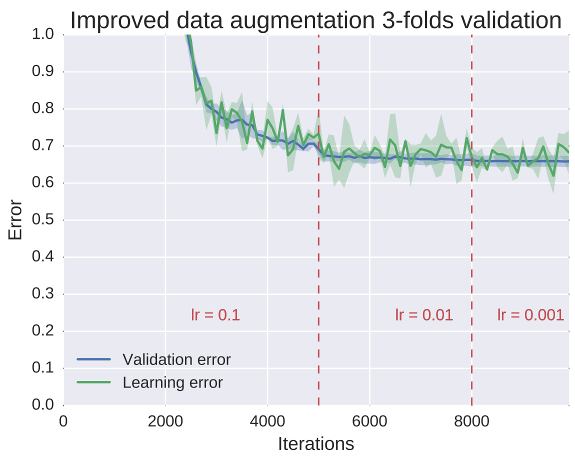
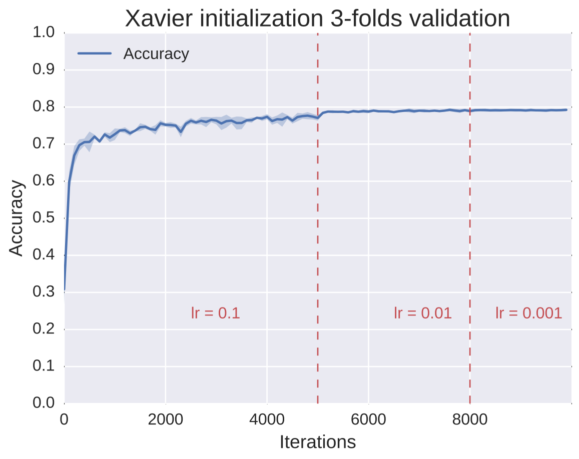


Figure 4.2: Learning curve of the ground architecture trained with the improved data augmentation procedure.



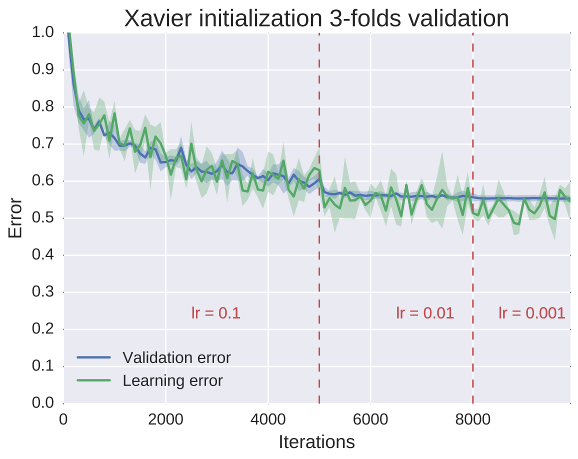


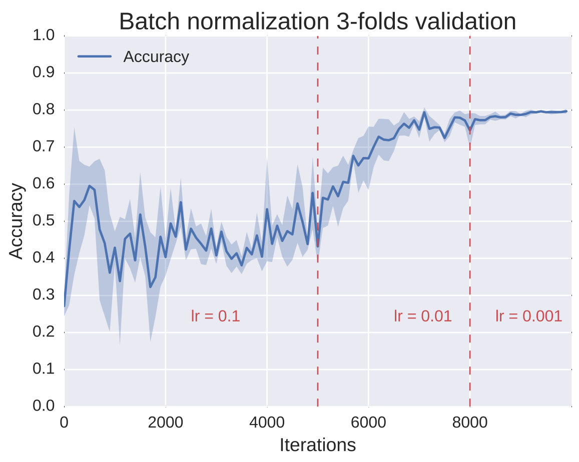
Figure 4.3: Learning curve of the ground architecture trained with the improved data augmentation procedure and initialize with Xavier.

4.1.2 Improved architecture

Xavier initialization

In fig. 4.3 we can see the effects of Xavier initialization on the learning. Xavier introduces a better learning in particular on the first 1000 iterations, respect to the same network with Gaussian weight initialization (see section 4.1.1). This result is in according with the expected behavior. Xavier initialization also involves in a better general learning, decreasing the validation error and increasing the accuracy.

This model reaches its best at iteration 7600 with an accuracy of 0.793 ± 0.004 and a validation error of 0.556 ± 0.006 and a corresponding . The learning error is instead of 0.553 ± 0.006.



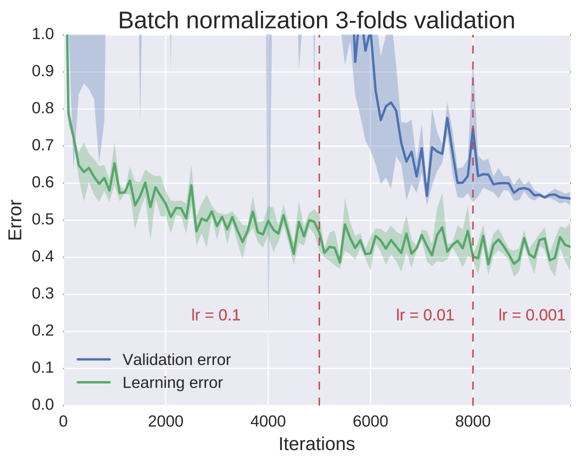


Figure 4.4: Learning curve of the ground architecture trained with the improved data augmentation procedure and initialize with Xavier. Batch normalization layer was added before each non-linearity.

Batch Normalization

The introduction of the batch normalization in our architecture produces a particular and unexpected behaviour on the learning curve. We can see that the algorithm has very difficult to recognize correctly the examples in the validation set, this is due to the strong overfitting that arises since the first iterations. The algorithm starts to generalize only when the learning rate decrease and converges completely only with the smaller learning rate adopted (fig. 4.4).

This model reaches its best at iteration 9900 with an accuracy of 0.797 ± 0.006 and a corresponding validation error of 0.56 ± 0.02. The learning error is instead of 0.43 ± 0.06.

We can't observe the enhancements reported in [16] on the error and in particular on the initial boost. This is probably due to our learning parameters.

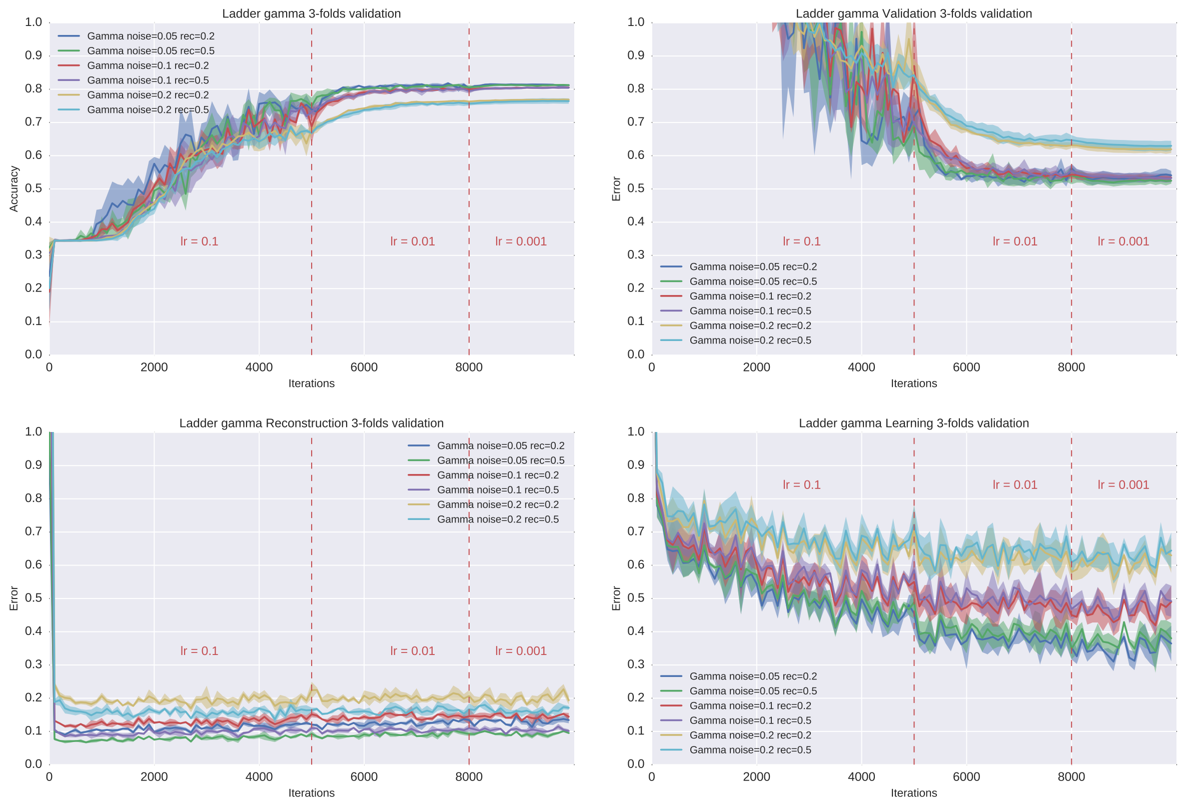


Figure 4.5: Learning curves of ladder-Γ architecture trained varying the noise and the reconstruction cost.

4.1.3 Augmented architecture

Ladder-Γ network

We explore the ladder-Γ architecture reported in section 3.3 varying the standard deviation of the noise introduced across layer and the reconstruction cost (the λ parameter). The standard deviation of the noise was modified in the same way along all layer and we refer at this value using the definition of *noise*, for simplicity.

As first effect we can observe in fig. 4.5 that this architecture improves the convergence of the learning curve reducing drastically the overfitting introduced by the batch normalization. The noise has a regularization effect and this probably has the strongest impact on the convergence across the first 5000 iterations.

The ladder-Γ model with noise of 0.05 and a reconstruction cost of 0.2 performs well than the others and reaches its best at iteration 8800 with an accuracy of 0.818 ± 0.003 and a corresponding validation error of 0.51 ± 0.01. The learning error is instead of 0.36 ± 0.04.

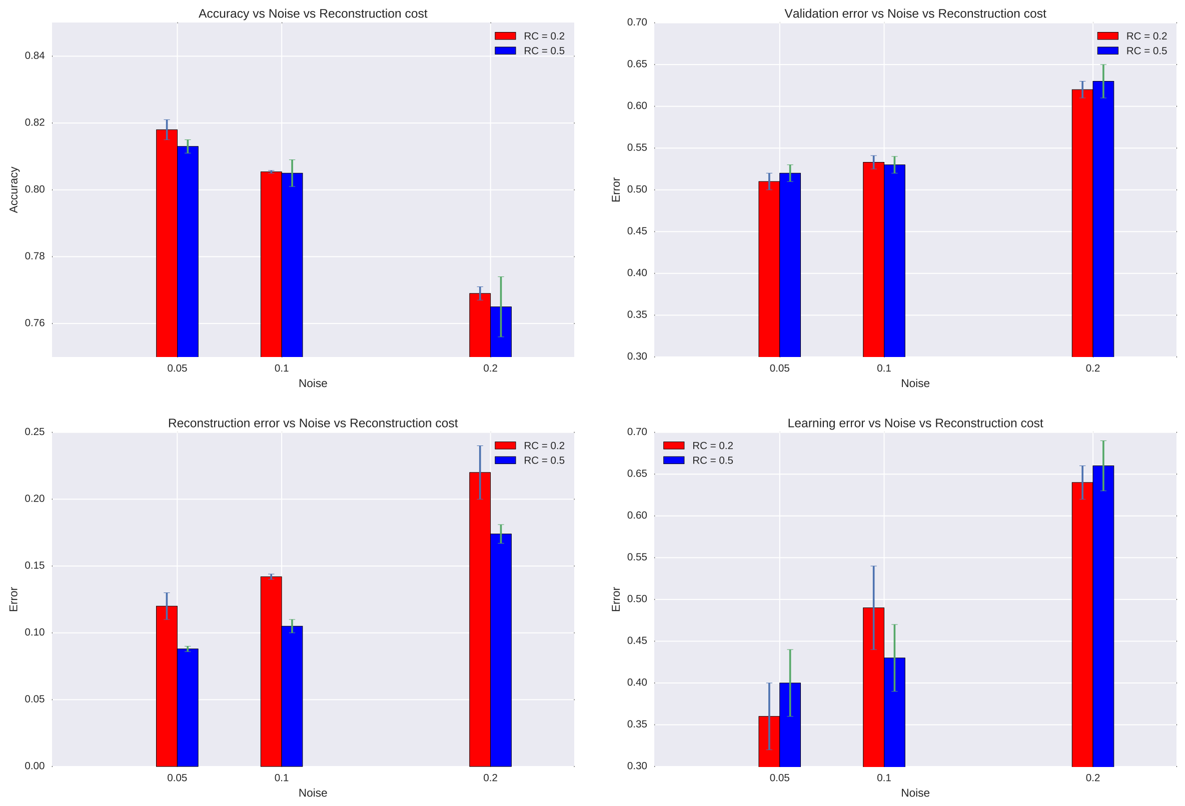


Figure 4.6: Best performance values on ladder-Γ for each value of noise and reconstruction cost. We can see that noise has a more strong impact on the learning.

We can observe that the noise has a strong effect on the performance of the network on the contrary of the reconstruction cost. While modifying the latter does not change significantly the performance of the algorithm, the noise produces a substantial effect on the learning curve.

In particular, in fig. 4.6, is clear which a reduction of the noise improves the general performance of the algorithm, augmenting the accuracy and reducing all the errors. However reducing the noise to near-zero value can causing overfitting. Reconstruction cost has a strong effect only on the reconstruction error, as expected, but doesn't exhibit a significant effect on accuracy or loss.

Ladder-Γ network with unlabeled example

As introduced in section 2.2, ladder network was born to introducing a semi-supervised training in the standard feed forward architectures. The unlabeled data can provide additional information on the structure of the underlying distribution, allowing the network

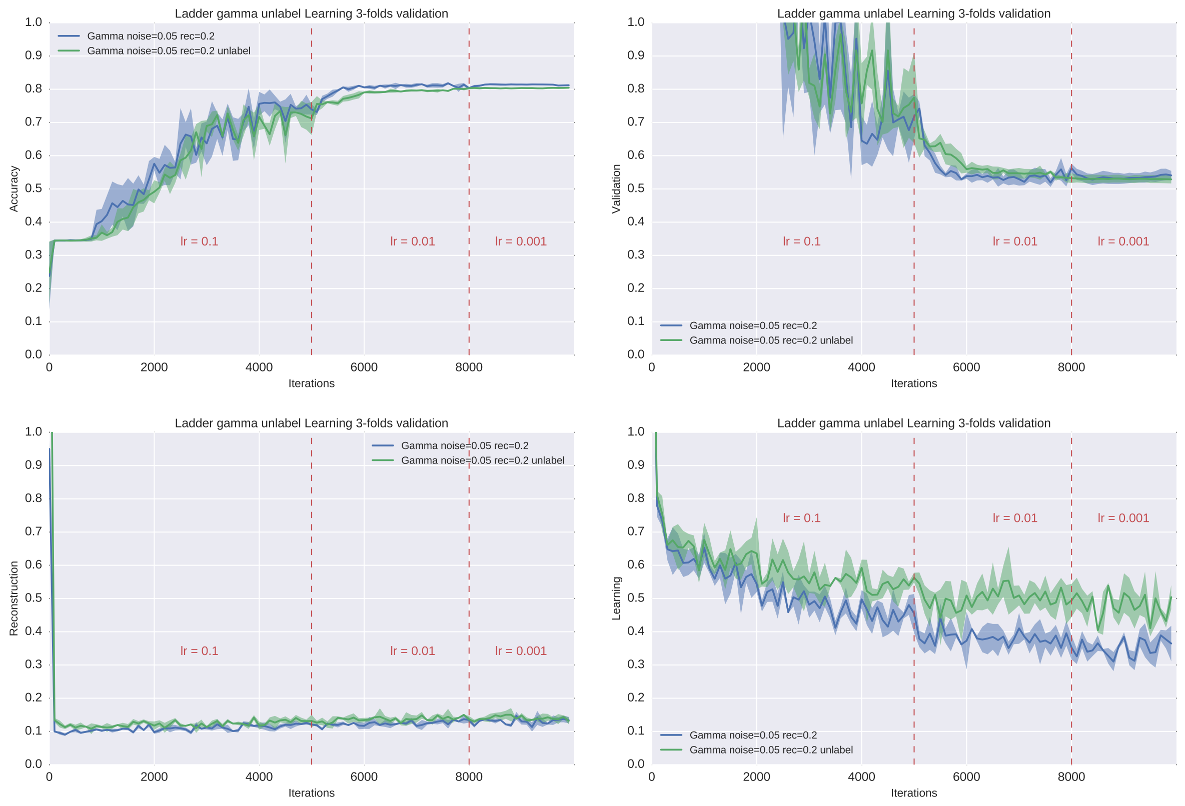


Figure 4.7: Learning curve of ladder-Γ architecture with noise of 0.05 and reconstruction cost of 0.2. Comparison on train with standard learning set and learning set with the unlabeled examples.

to learn a better representation.

Following this, we have try to train our ladder-Γ architecture using the dataset containing the unlabeled examples described in the section 3.2. We compare the resulting training using our best ladder-Γ architecture, as shown in fig. 4.8.

We can observe that the use of the unlabeled examples deteriorates the performance of the algorithm, in particular of the accuracy, according to the statistic. This model reaches an accuracy of 0.805 ± 0.002 and a corresponding validation error of 0.530 ± 0.001. The learning error is instead of 0.50 ± 0.06.

Our best hypothesis is that the unlabeled examples present in the database have a different distribution of the nuclei than the one of the labeled examples. The ladder network tries to reduce this bias between the two distribution causing a degeneration on the learning algorithm (*i.i.d.* principle it fails).

As confirmation of these hypothesis, to exclude a malfunctioning of the algorithm, we have reproduced the condition to have a set of unlabeled data coming from the

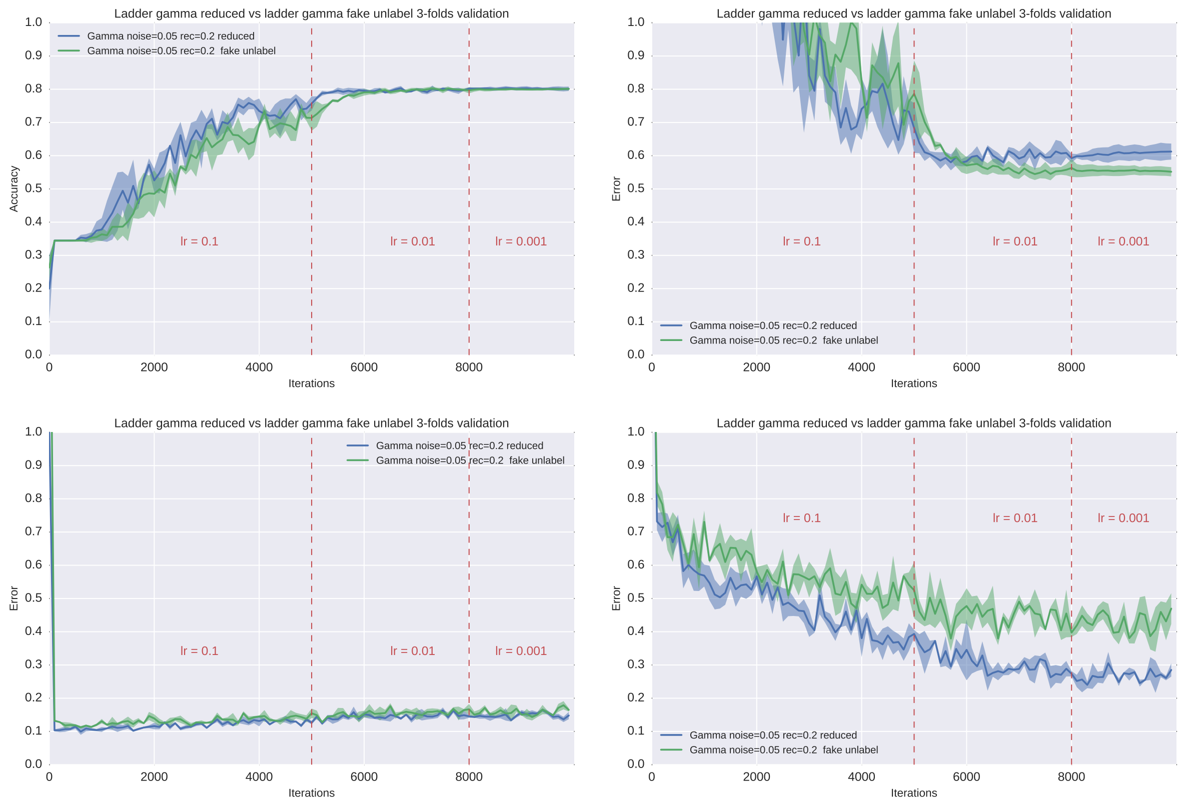


Figure 4.8: Learning curve of ladder-Γ architecture with noise of 0.05 and reconstruction cost of 0.2. Comparison on train with learning set reduced by 30% and learning set with fake unlabeled examples obtained transforming 30% of labeled examples in unlabeled.

same distribution of the labeled data. This was done transforming randomly 30% of the examples in learning set in unlabeled examples to observe the behaviour of the ladder network. This percentage reproduces moreover the ratio of the unlabeled examples of the original training set. The result was compared with a learning set from which we have completely removed the same amount of examples.

We can see that the use of unlabeled examples slowing down sensibly the convergence but allow to reach the same accuracy level. The effect of the use of unlabeled examples can be observed on the validation error, where the validation error is smaller, and on learning error, where we have a better generalization. In details, while the supervised train reaches an accuracy of 0.803 ± 0.006, the semi-supervised one is 0.802 ± 0.006 at iteration 8700 and 7600 respectively. They are consistent considering the error bars. The correspondent validation errors are 0.60 ± 0.002, for the supervised curve, and 0.54 ± 0.02 for the unsupervised one. For the learning errors we observe values of 0.31 ± 0.03 and 0.467 ± 0.007. On these errors the discrepancies become significants.

4.1.4 Model selection

In table 4.1 is reported a summary of the results on different architectures explored in model selection. For the ladder network is reported only the model with noise and reconstruction error values which performing better.

We can see that our best model is the ladder-Γ architecture with a noise value of 0.05 and a reconstruction error of 0.2, trained on the learning set containing only labeled examples.

Architecture	**Accuracy**	**Validation error**	**Learning error**
Ground	0.725 ± 0.006	0.73 ± 0.01	0.78 ± 0.05
Improved data augmentation	0.754 ± 0.006	0.65 ± 0.01	0.65 ± 0.04
Xavier	0.793 ± 0.004	0.556 ± 0.006	0.553 ± 0.006
Batch normalization	0.797 ± 0.006	0.56 ± 0.02	0.43 ± 0.06
Ladder-Γ	0.818 ± 0.003	0.51 ± 0.01	0.36 ± 0.04
Ladder-Γ with unlabeled examples	0.805 ± 0.002	0.530 ± 0.001	0.50 ± 0.06

Table 4.1: Comparison between the architecture explored in the model selection. The ladder-Γ architecture with noise 0.05 and reconstruction cost of 0.2 performs better than the others reaching the highest accuracy and the lowest validation error.

4.2 Test

As final architecture we choose the best performing on model selection, the ladder-Γ with noise 0.05 and a reconstruction cost of 0.2. We have trained this architecture on training set with the standard training parameters described in section 3.4. We select the model at the last iteration to run the performance measures on test set.

With this model we reach an accuracy of 0.8173% with a test error of 0.497. The train error and the reconstruction error are respectively 0.417 and 0.130.

In table 4.2 is reported the confusion matrix on the test set.

As comparison we report the results presented in [34] in which the database was introduced and tested. In this work was used the same CNN ground architecture described earlier and was developed a classification method which improves the classification performance, called Neighboring Ensemble Predictor (*NEP*). The measures reported are the weighted average F1 score and the Multiclass AUC [10]. This latter was calculated using the probabilities score of the softmax output as prediction, in our case computed with [20].

	Epithelial	**Fibroblast**	**Inflammatory**	**Miscellaneous**
Epithelial	2104	81	103	22
Fibroblast	192	1238	293	34
Inflammatory	66	105	1866	49
Miscellaneous	56	102	127	295

Table 4.2: Confusion matrix

Architecture	**Weighted Average F1 score**	**Multiclass AUC**	**Accuracy**
Ground [34]	0.748	0.893	N/A
Ground+NEP [34]	0.784	0.917	N/A
Ladder-Γ	**0.812**	**0.940**	0.8173

Table 4.3: Comparison between the results reported in [34] and our results. We reach better scores on both weighted average F1 score and multiclass AUC. Accuracy scores are not available for a comparison.

We can see in table 4.3 the comparison. We reach better results on both the scores.

Another recent work approaches this problem of classification on this database [2]. In this work the problem was approached using very deep architectures like *GoogleNet* [38], *AlexNet* [19] and *VGG-16* [33] trained with transfer learning method, which use a pre-trained model on a different task to fine-tuning on the desired problem. They reach their best with the *VGG-16* architecture, which implements 13 convolutional layers, with an accuracy of 0.8803. However a complete comparison is not possible due to the fact that they have eliminated the miscellaneous nuclei class in their problem, which are checked to be the most difficult to classify, according to the confusion matrix 4.2.

Our system shows good performances on the task of nuclei classification in histological colon cancer images, especially considering the poor depth, compared with modern architectures, of the CNN used. THe greatest part of the improvements, produced on the algorithm of base from which we have begun, has only concerned the procedure of training, resulting in a final algorithm with the same complexity of that initial.

Conclusions

In this work we have explored the modern advances in the machine learning field, in particular in the development of the convolutional neural networks. CNNs are powerful algorithms to solve many difficult tasks, in particular show incredible performance on the images classification problem.

We have tried to apply this methods to a task of classification of nuclei in colon cancer histology images obtaining good results.

Specifically, starting from a previous work, we have explored some alterations on the basic CNN architecture reported. Almost each of these alterations has introduced an improvement on the performance. In details, we have trained the CNN with an improved data augmentation method, we tried out the Xavier initialization and we introduced the batch normalization layer. Only this latter seems to not performing as expected, but resulting anyway in a good final performance.

We had a particular focus on the implementation of an architecture called ladder-Γ which enables the semi-supervised framework on CNN architecture. The database of colon histology images comes with a set of unlabeled examples which allow us to test this framework. However the use of unlabeled examples results in a degradation of performances. Our best hypothesis is that these unlabeled data (collected for detection purpose only) present a bias distribution compared with the labeled data. Also if used only in supervised learning fashion, the ladder-Γ architecture improves efficiently our algorithm allows the reaching of the best performance in our models.

The final result on test set compared with the results reported in the original paper show a significant improvement in performances. These were obtained without augmenting the complexity of the final algorithm but mostly carefully setting the inference process. It is not excluding that deep CNNs can performing better on this particular task, however the comparison with a work which using very deep architectures, to deal with similar problem, shows comparable measures.

This work is a starting point for a more complete method which implements a detection stage and that can produce a useful support in the diagnosis of colon cancer.

Bibliography

[1] Javier Adur, Mariana Bianchi, Vitor B Pelegati, Silvia Viale, María F Izaguirre, Hernandes F Carvalho, Carlos L Cesar, Víctor H Casco, et al. Colon adenocarcinoma diagnosis in human samples by multicontrast nonlinear optical microscopy of hematoxylin and eosin stained histological sections. *Journal of Cancer Therapy*, 5(13):1259, 2014.

[2] Neslihan Bayramoglu and Janne Heikkilä. Transfer learning for cell nuclei classification in histopathology images.

[3] Yoshua Bengio. *Practical Recommendations for Gradient-Based Training of Deep Architectures*, pages 437–478. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[4] G. Bradski. *Dr. Dobb's Journal of Software Tools.*

[5] Joan Bruna, Arthur Szlam, and Yann LeCun. Signal recovery from pooling representations. *arXiv preprint arXiv:1311.4025*, 2013.

[6] Vladimir S. Cherkassky and Filip M. Mulier. *Learning from Data.* Adaptive and Learning Systems for Signal Processing, Communications and Control. Wiley, 1998. Concepts, Theory, and Methods.

[7] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and D. Mike Titterington, editors, *AISTATS*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org, 2010.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[9] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, and Gang Wang. Recent advances in convolutional neural networks. *arXiv preprint arXiv:1512.07108*, 2015.

[10] David J Hand and Robert J Till. A simple generalisation of the area under the roc curve for multiple class classification problems. *Machine learning*, 45(2):171–186, 2001.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

[13] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[14] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. Deep networks with stochastic depth. *CoRR*, abs/1603.09382, 2016.

[15] David H. Hubel and Torsten N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology (London)*, 195:215–243, 1968.

[16] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[17] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[18] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[20] Stuart Lacy. Mauc implementation in python. `http://stuartlacy.co.uk/20150414-MAUC-python`. (Accessed on 11/22/2016).

[21] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[22] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[23] James S Lewis Jr, Sahirzeeshan Ali, Jingqin Luo, Wade L Thorstad, and Anant Madabhushi. A quantitative histomorphometric classifier (quhbic) identifies aggressive versus indolent p16-positive oropharyngeal squamous cell carcinoma. *The American journal of surgical pathology*, 38(1):128, 2014.

[24] Anant Madabhushi and George Lee. Image analysis and machine learning in digital pathology: Challenges and opportunities. *Medical Image Analysis*, 33:170–175, 2016.

[25] J Moody, S Hanson, Anders Krogh, and John A Hertz. A simple weight decay can improve generalization. *Advances in neural information processing systems*, 4:950–957, 1995.

[26] Michael Nielsen. Neural networks and deep learning. `http://neuralnetworksanddeeplearning.com/`. (Accessed on 11/18/2016).

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[28] Mohammad Pezeshki, Linxi Fan, Philemon Brakel, Aaron C. Courville, and Yoshua Bengio. Deconstructing the ladder network architecture. *CoRR*, abs/1511.06430, 2015.

[29] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[30] Antti Rasmus, Tapani Raiko, and Harri Valpola. Denoising autoencoder with modulated lateral connections learns invariant representations of natural images. *arXiv preprint arXiv:1412.7210*, 2014.

[31] Antti Rasmus, Harri Valpola, Mikko Honkala, Mathias Berglund, and Tapani Raiko. Semi-supervised learning with ladder network. *CoRR*, abs/1507.02672, 2015.

[32] John L Semmlow and Benjamin Griffel. *Biosignal and medical image processing*. CRC press, 2014.

[33] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[34] Korsuk Sirinukunwattana, Shan E Ahmed Raza, Yee-Wah Tsang, David R. J. Snead, Ian A. Cree, and Nasir M. Rajpoot. Locality sensitive deep learning for detection and classification of nuclei in routine colon cancer histology images. *IEEE Transactions on Medical Imaging*, 35(5):1196–1206, may 2016.

[35] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[36] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1139–1147. JMLR Workshop and Conference Proceedings, May 2013.

[37] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[39] Vladimir Naoumovitch Vapnik. *Statistical learning theory*. Adaptive and learning systems for signal processing, communications, and control. Wiley, New York, 1998.

[40] Mitko Veta, Robert Kornegoor, André Huisman, Anoek HJ Verschuur-Maes, Max A Viergever, Josien PW Pluim, and Paul J van Diest. Prognostic value of automatically extracted nuclear morphometric features in whole slide images of male breast cancer. *Modern Pathology*, 25(12):1559–1565, 2012.

[41] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

[42] David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Comput.*, 8(7):1341–1390, October 1996.

[43] Dingjun Yu, Hanli Wang, Peiqiu Chen, and Zhihua Wei. Mixed pooling for convolutional neural networks. In *International Conference on Rough Sets and Knowledge Technology*, pages 364–375. Springer, 2014.

[44] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[45] Matthew D Zeiler and Rob Fergus. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*, 2013.