Adversarial Learning with Ladder Networks

Juan Maroñas Molano

September 18, 2016

Roberto Paredes Palacios Alberto Albiol Colomer

Department of informatic systems and computing
Politecnic University of Valencia

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Abstract

The use of unsupervised data in addition to supervised data in training neural networks has improved the performance of this classification paradigm. However, the best results are achieved with a training process that is divided in two parts: first an unsupervised pre-training step is done for initializing the weights of the network and after these weights are refined with the use of supervised data. We study a model that train both parts at the same time and get state of the art results.

On the other hand adversarial noise has improved the results of classical supervised learning. In this work we mix both training process and get state of the art classification, with several important conclusions on how adversarial noise can help in addition with new possible lines of investigation.

Aknowledgment

I would like to thank Alberto Albiol Colomer, Roberto Paredes Palacios from the Politecnic University of Valencia and Daniel Ramos Castro from the Autonomous University of Madrid.

KeyWords

Gradient, Adversarial Noise, Semi-supervised Learning, Neural Networks, Convolution, Latent Variables, Hierarchical latent variables.

Contents

List of Tables

List of Figures

1 Notation

This section is to briefly expose the notation that is going to be used along the work.

- In mathematical operations, vectors are column vectors.
- A multidimensional signal is represented lower case letter x.
- Each component of a vector is represented with $x_i, i \in \mathbb{N}$
- If the signal is a sequence we express it like: $x(s), s \in \mathbb{R}^k$
- A signal in a layer, l, is represented with: $x^l, l \in \mathbb{N}$
- If the context of the problem is probabilistic, x and all its variants represent random variables/process.
- X represent a specific realization of a random variable.
- $\mathcal{X} = \{X_1(k), X_2(k), X_3(k), ..., X_n(k)\}$ represent a sample of the distribution of x as a set, where X_1 represent the first sample of the set and $X_1 \in \mathbb{R}^k$.
- Matrix are represented with capital letter W.
- A function of an independent variable, x, (and possible random variable/process) is represented with $f(x)$. In this general case we will talk about functions that take as input a vector of arbitrary dimension.
- Capital $\mathcal{A}$ represent neural network activation functions (see ***Appendix 1***).
- Cost functions are represented with C and the learning rate with α.
- Letter x is used to represent: inputs to a layer (x^l) or data samples (x). Letter t is used for the outputs. Letter p represent parameters.
- When talking about hidden layers in a neural network if we say k layer neural network we are referring to a neural network with k hidden layers plus the output layer and the input layer, that is a neural network with k+2 layers.
- Widehat, $\widehat{\ }$, is used to represent the true value of a random variable. $\widehat{t}$ would represent the true values of the random variable t, that is, the true tags of a in input x whose mapping is the objective of the learning process.
- We will call a corrupted value of x those values computed from a perturbation of x and express them like $\widetilde{x}$
- A reconstructed value from x will be express like $\bar{x}$.

- When performing a derivative of a function we will express the derivative of a function $f(x)$ as $\frac{\partial f(x)}{\partial x}$ or $f(x)'$. In the second case the variable wrt we are doing the derivation will be inferred by the context.
- The symbol $\cdot$ will be used as a matrix product. We will represent the dot product using the inner product notation $\langle , \rangle$
- Bold italic words like: ***word*** represent links to other parts in the document.

2 Introduction

2.1 Project Motivation

Neural networks are powerful machine learning models that has been being studied for years, [Bishop, 1995]. However, its use in computer vision applications became popular when Alex Krizhevsky outperform the image Net classification problem in the ILSVRC-2012 competition with the famous convolutional neural network AlexNet, [Krizhevsky et al., 2012]. To that point people thought neural networks were not useful for real problems and the high computation cost for training them make them something unpopular. Support Vector Machine was one of the most important machine algorithms for learning discriminative tasks. There were important researchers such as Yann LeCun, Yoshua Bengio or Geoffrey Hinton whose contributions to neural networks where not applied further than demonstrating their utility in toy problems such as the MNIST handwritten recognition task. Krizhevsky presented the convolutional network with software for make use of the advantages of GPUs parallelization to make the training computationally possible. From that point all the people started using neural networks for any task (natural language processing, machine translation...), some toolkits for GPU programming where developed, GPU performance was improved... to reduce the time in training a neural network.

Several techniques for improving the performance of this kind of model has been proposed in recent years. Neural networks where used for classification or regression problems with only supervised data. One of the key things researchers has been exploring in the last years is the use of unsupervised data for helping the learning process. The two main reasons are:

- There are much more unsupervised than supervised data.
- Very deep architecture suffer from vanishing gradient so early layers cannot be well trained.

Until know, all the techniques that make use of unsupervised data divide the learning process in two steps: first, unsupervised data is used in some way to initialize the network or pre-train the weights of the network; second, supervised data is used to train the discriminative model. Joining both steps in only one, that is, training the classifier at the same time the unsupervised data is used to help this training process has shown to outperform the results of models that make use of classical techniques using unsupervised data. We will call this semi-supervised learning.

On the other hand, researchers has found in the addition of noise to the training data an easy way for the network to better generalize. Adversarial noise [Goodfellow et al., 2014] (which will be correctly defined after) is a kind of noise computed from the cost function that has shown an improvement in supervised neural networks training compared to classical gaussian noise addition.

2.2 Objectives and Approaches

The main objective of this work is to study the use of adversarial noise in a state of the art model that uses supervised and unsupervised data at the same time for learning a discriminative classifier, and try to improve the results. All this study would be done in the context of MNIST handwritten task classification, with a fully connected neural network and all the tags from the training set. Finally, with the best results we will try to improve other classifications task that involve other databases, convolutional neural networks and a smaller number of labeled data.

The work would be structured as follows:

1. **Study of adversarial noise in the model**: The first part is to study if the model is robust to adversarial noise or not
2. **Incorporation of adversarial noise**: Study several possible ways of computing and adding adversarial noise to the network.
3. **Validation**: Validate the proposed model

Secondary objectives are the study and understanding of latent variable models which give a mathematical base to the technique showed in this work. Also, the study of adversarial noise in different vector spaces: data space and costs functions spaces; to show why this noise outperform the classification task.

The memory is structured in the next way:

- Mathematical Notation
- State of the art revision
- Mathematical Foundations
- Experimentation and Results
- Discussion

3 State of the Art

Neural Networks models are a wide field in machine learning applications. There are several kinds of models depending on the application. For that reason, there are lots of fields under exploration in neural network improvements. In this section we first define what is a neural network and what is common for all the neural networks and then revise the most important improvements in neural networks that have an influence in our field of study.

3.1 Classical Neural Networks

Neural networks are algebraic projections from an input vector space of arbitrary dimension to an output vector space. We restrict our models to real value mathematical spaces. The different interpretations of that outputs depends on the application:

$$t = f(x);\ x \in \mathbb{R}^k, t \in \mathbb{R}^{k'}, k, k' \geq 1 \tag{1}$$

The mapping function f, is of the form $f : \mathbb{R}^k \to \mathbb{R}^{k'}\ k, k' \geqslant 1$. In a neural network topology, see figure 1, this f is represented in steps. This means that we reach the last vector space through different vector spaces from arbitrary dimensions. Each layer in a neural network represent a different vector space. So f is a combination of different mappings through different vector spaces with possible different dimensions, from input to output.

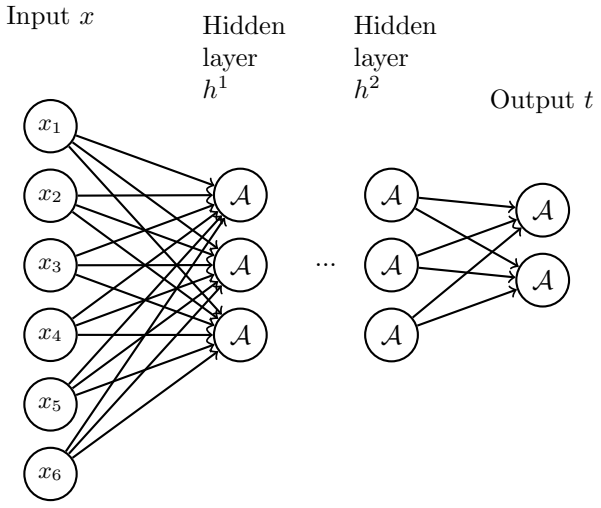


Figure 1: Neural Network topology. A mapping from input $x \in \mathbb{R}^6$ to output $t \in \mathbb{R}^2$ through at least two intermediate vector spaces $h^1, h^2 \in \mathbb{R}^3$

In the classical neural networks models, the operations to perform this projections are restricted to linear operations and non-linear activations. In a neural network the result or output of a neuron (a neuron represents one of the dimensions in the vector space in a layer) is the linear combination of the previous neurons outputs and a non-linear operation of that combination. The general notation of a neural network operation between two layers is the next one:

$$x^{l+1} = \mathcal{A}(W \cdot x^l + b), x^l \in \mathcal{R}^k, x^{l+1}, b \in \mathcal{R}^{k'}, W \in \mathcal{R}^{k \cdot k'}, k, k' \geqslant 1 \qquad (2)$$

Where $\mathcal{A}$ represent the non-linear operation, see ***Appendix 1***, W are the weights matrix and b is the bias vector of the linear combination. As we can see each component of the bias vector correspond to the bias of each neuron in the next layer. The weights are a matrix so in one product we represent the linear combination of the neurons in actual layer for each neuron in the next layer. W and b are also known as the parameters of f.

Activation functions are used to learn non-linear projections between spaces. Note that a sequence of linear operations can be represented in one only linear operation. This would be a 0-hidden layer neural network.

3.2 Neural Networks for i.i.d problems

In the problems we address, we have a set of samples drawn iid from a probability distribution. On the other hand, our samples are not stochastic process, that is, they do not depend on an independent variable such as time or position. Recurrent Neural Networks (RNN) are a kind of neural network models used for these problems. In RNN the input is made from the signal at time t and signal at time $t \pm r, r \in \mathbb{R}$. This leads to a model quite different from the one explained.

There are some neurons topology (different from the classical one), called LSTM neurons (long- short-term memory), whose mapping from the input space to the output space are an extension of the one in equation 2, but the basic idea is the same one.

For the problems we try to solve, there are two main networks: fully connected (FC) and convolutional. The fully connected is the one in figure 1. Convolutional networks are in some way similar to FC. The big difference is that the linear projection between vector spaces is given by a convolution of the input to that transformation and a kernel that represent the weights. For example a $3 \cdot 3$ kernel would result in 9 weights. These weights are shared by all the components of the input vectors to the projection. We will talk about convolutional networks after, but for the moment an example is given by figure 2.

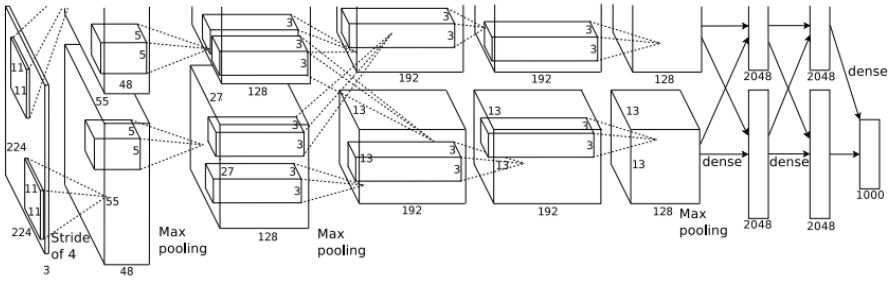


Figure 2: AlexNet,[Krizhevsky et al., 2012] Convolutional network topology.

In the next two subsection we will talk about the main improvements in neural network training strategies. We will talk about the improvements for that networks that suit our problem. Some of these characteristics are common to RNN networks. We divide this in the improvements in the training criteria and algorithms; and the improvements in neural network topologies.

3.3 Improvements in Learning Process

The objective of a learning process in a machine learning application is minimizing the cost function which respect to the parameters, see ***Appendix 2***.

The way we approximate the search of the optimum (maximum or minimum) is by a classical (vanilla) method call stochastic gradient descent (SGD). SGD is defined as in Algorithm 1. In neural networks we implement the derivatives in SGD by back-propagation [Rumelhart et al., 1986], see ***Appendix 2*** for more details:

input : $\mathcal{P} = \{P_1, P_2, ..., P_M\}$,$\mathcal{X} = \{X_1, X_2, ..., X_N\}$,$\alpha$,$C$,Iterations
output: $\mathcal{P}^{'} = \{P_1^{'}, P_2^{'}, ...P_M^{'}\}$
Require: $\alpha \geq 0$;
Variable Initialization;
it $\leftarrow 0$;
while *it* $\leq$ *Iterations* **do**
  for $p \in \mathcal{P}$ **do**
    $p^{'} = p - \alpha \cdot \frac{\partial C}{\partial p}$;
  end
  $\mathcal{P} = \mathcal{P}^{'}$;
  it += 1;
end

Algorithm 1: Vanilla Stochastic Gradient Descent

Momentum

Momentum,[Rumelhart et al., 1986], is a modification of the vanilla SGD to avoid local minimum and prevent the SGD from oscillations which is also known as the poor conditioning problem. In some way it has two purposes: first is accelerating the optimization process and avoid local minimum.

The idea is to keep going in the same direction to where the older computed gradient pointed to. Imagine a ball in a parameter space, if we throw the ball through the cost function, it will keep going in the same direction unless it finds a big change in the slope. Due to the acceleration and the velocity if we find a point with a higher value in the cost function than the actual (a local minimum) we are capable of avoiding it. With this same example if the gradient at some point start to oscillate, we are capable of keeping the principal direction in the minimization process.

The algorithm 1 changes to Algorithm 2. Let's call the momentum fraction $m \in [0, 1]$. As we can see the parameter updating has an influence of the gradient in previous steps so it has memory of what has been happening during the optimization process.

input : $\mathcal{P} = \{P_1, P_2, ..., P_M\}$,$\mathcal{X} = \{X_1, X_2, ..., X_N\}$,$\alpha$,$C$,Iterations, m
output: $\mathcal{P}^{'} = \{P_1^{'}, P_2^{'}, ...P_M^{'}\}$
Require: $0 \leq m \leq 1$ *and* $\alpha \geq 0$;
Variable Initialization;
it $\leftarrow 0$;
$accM \leftarrow 0 \in \mathcal{R}^m$ *Vector of shape given by number of parameters*;
while *it* $\leq$ *Iterations* **do**
  for $p \in \mathcal{P}$ **do**
    $p^{'} = p - \alpha \cdot \frac{\partial C}{\partial p} + m \cdot accM_p$;
    $accM_p = -\alpha \cdot \frac{\partial C}{\partial p}$;
  end
  it += 1;
  $\mathcal{P} = \mathcal{P}^{'}$;
end

Algorithm 2: Momentum Stochastic Gradient Descent

Figure 3 shows a comparison between vanilla SGD and momentum SGD. As we can see optimization is improved. In this example we address the problem of poor conditioning. The poor conditioning problem appears when little changes in the input to the function suppose big changes in the optimization function. When adding noise to input this effect can appear and momentum minimizes the effect that could have these changes in the optimization process in vainila SGD. These effects are basically oscillations in the optimization process. We can see how the momentum optimization is influenced by the previous gradient

direction, giving bigger steps at each iteration. If the cost function start to oscillate, momentum keeps the optimization in the same direction which improve the time in reaching the optimum.

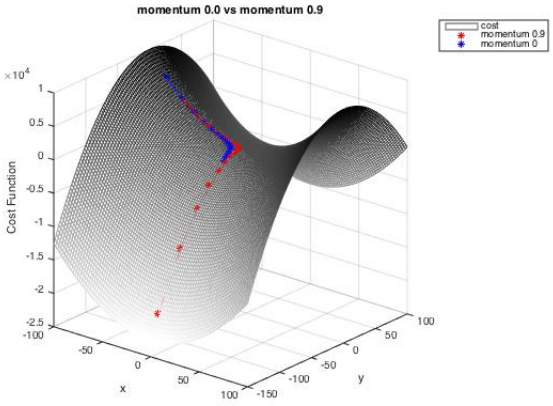


Figure 3: Comparison between vanilla and momentum SGD. Experiment done with learning rate 0.1 and 15 iterations

Momentum is also useful to avoid local minimum. Looking at figure 4 we can see how the red optimization is capable of "jumping" in the cost function and will always reach a minimum lower than the blue optimization. Higher learning rates can also show this effect but here the learning rate is exactly the same.

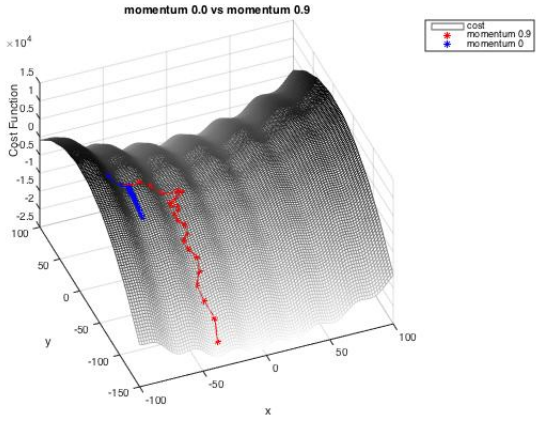


Figure 4: Comparison between vanilla and momentum SGD. Experiment done with learning rate 0.03 and 23 iterations

3.3.1 Learning Rate Adaptation

Adapting the learning rate has shown to improve the results. This adaptation allow our optimization process not to oscillate so easily. There are different ways of doing it. Learning Rate Annealing is a simple technique that consist in keep reducing the learning rate when we are not improving the number of errors achieved.

This can be done in several ways. We can decay the learning rate with a function that depends on the number of epochs. We can reduce one magnitude order when we see that we are not doing better than a number of errors between epochs...

Another technique is the one called *Bold Driver* [Battiti, 1989]. The *Bold Driver* algorithm compare the loss between two epochs and increase or decrease the learning rate depending on if we are doing better or not.

Finally, we could find the local rate adaptation. As we explained in SGD we do not take the partial cross derivatives so we do not care about how the function changes in one direction wrt to the other. This means that we can have a function that have high different behaviors depending on the dimension and we are giving the same step in all the directions, and we should not. One direction could need a little learning rate and the other a higher. There are several ways of doing this local adaptation. One example is optimizing first the cost function wrt the learning rate and then perform parameter update.

Other variants of SGD focus their attention in how modifying the learning rate: Adagrad, Adadelta, RMSprop, and Adam.

3.3.2 Divide training set in batches

As we defined in ***Appendix 1***, the cost we minimize when training a neural network is a summation over the training set. However, a typical way of training is by dividing the training set into batches, that is, portions of the training set and then updating the parameters each time we compute the cost for a batch.

There are three main ways of parameter updating when training a neural networks. When we use only one data to compute the cost and after perform parameter updating we call it online learning. This is a suitable way of parameter updating when we know that our data changes through time. A famous online learning algorithm is the online passive aggressive [Crammer et al., 2006].

The other extreme is the use of the whole data set to compute the cost and then perform the parameter updating. This is for sure a less noisy representation of the cost function because all the data drawn from our distribution is present in the computation of the cost function. However, this slow down the learning.

A typical solution is the use of mini-batches, that is, portions of the training data. This makes our cost function a bit more noisy but we are capable of learning quicker due to the fact that the operation of a minibatch take advantage of the hardware architecture. It is much quicker to perform a operation of a whole minibatch (remember we could express operations using matrix products and there are lots of techniques for parallel matrix multiplication) than m operations on individual data, and, for sure, is much quicker than performing through the whole data set. However the size of the minibatch is a hyperparameter to search: the bigger the minibatch the better the estimation of the cost function but maybe we do not take all the advantage of the hardware architecture. Normally a noisy gradient implies a slow down in the learning process, however in modern computing platforms the advantages of hardware operation overcomes the slow down due to noisy gradients.

3.3.3 Second Order Optimization

Until know we have seen how can we minimize a function using first order derivatives. When minimizing we "look" around the point in the parameter space we are and go in the deepest direction for each parameter.

The second order term or Hessian give us information about the curvature of the surroundings of the point in which we are. We can use this curvature information to outperform the minimization. An easy example is that if we are exactly in a saddle point we cannot continue moving using vainilla SGD (maybe with momentum we can get out from here), however if we have information about curvature, we will know that there is a change in the curvature and that means we are not at a minimum or maximum.

The very big problem of these approaches is the computation cost due to the computation of the second order term that increases quadratically with the number of parameters. In chapter 7 from [Bishop, 1995] there is a good description of non-linear optimization algorithms that use the hessian. Non-linear algorithms are algorithms that minimizes functions with non-linear operations.

3.4 Generalization

Neural networks are powerful models for representing data. For that reason it is easy to overfit, that is, a very good representation of the training data but poor generalization of the distribution. There are several techniques to prevent the network from overfitting. In this section we revise the most important ones.

3.4.1 Data transformation

Data transformation is an easy way to improve performance in neural networks. If we change the input to the neural network we will have more training samples,

even though these new samples are virtual samples. In a general case adding uncorrelated noise, that is gaussian noise, is a good way to improve generalization. Adding uncorrelated noise means, in some way, generating new possible data that the underlying distribution could have also generated. The mean and variance of this gaussian noise depend on the data. We can not add noise with a 0 mean to data that is not centered in 0. Noise has to have a correlation with the data structure.

In computer vision typical operations to the data are: flips, crops and shifts:

- flip: Consists in flipping the images.
- crop: Consists in taking random parts of the images instead of the whole image.
- shift: Consists in shifting the images with affine transformations.

These operations have sense. For example we can find an image with an 8 and other image with an 8 rotated and if we are learning 8s our network should learn those possibilities. We could find also 8 partially occluded. Another way of producing new data is by performing morphological operations such as dilation and erosion and combinations of them: opening and closing.

Finally, a very typical way of improving neural networks performance and training cost is by data normalization. There are two main ways of doing this normalization. One is what we call zscore and consist in making the data have zero mean and one standard deviation. The other main way is by fitting our data to the range 0-1. The next two equations shows how to perform this normalization. Given our data set $\mathcal{X} = \{X_1, X_2, ..., X_N\}$, $X_i \in \mathbb{R}^k$:

Zscore:

$$X_j(i) = \frac{X_j(i) - \mathbb{E}[\mathcal{X}]}{\sqrt{Var[\mathcal{X}]}} \tag{3}$$

Zero-One range:

$$X_j(i) = \frac{X_j(i) - \min(\mathcal{X})}{\max(\mathcal{X}) - \min(\mathcal{X})} \tag{4}$$

Data normalization helps for two main reason. The first one is because we have all our data in the same range. Maybe the train data and the test data distributions are not the same one (variability problem and there are other ways to solve this) but at least all the data is in the same range. If the data is in a different range we would not perform well with test data. Also, notice that we have feature spaces $\mathbb{R}^k$. Nothing ensures that one of the dimensions is much bigger than the other ones. This means that the linear combination of

the features can be governed by one of the features for the same reason that will be exposed when talking about regularization. This would influence the training time (we will need more time to adjust the weights) and would be more sensible to weight initialization. Note that the normalization is done for each dimension because of how the mean operator, $\mathbb{E}[\cdot]$ is defined. Remark that the min operator in equation 4 is the minimum dimension in the data set.

3.4.2 Dropout

Dropout [Srivastava et al., 2014] is a good way of improving neural networks performance. It is based on the idea of combining different neural networks for a task. If we have to recognize objects in an image, the best way would be training different topologies and use all of them to perform the final classification. This, for sure, is a computation prohibitive task. Dropout was proposed with the idea of simulating this effect.

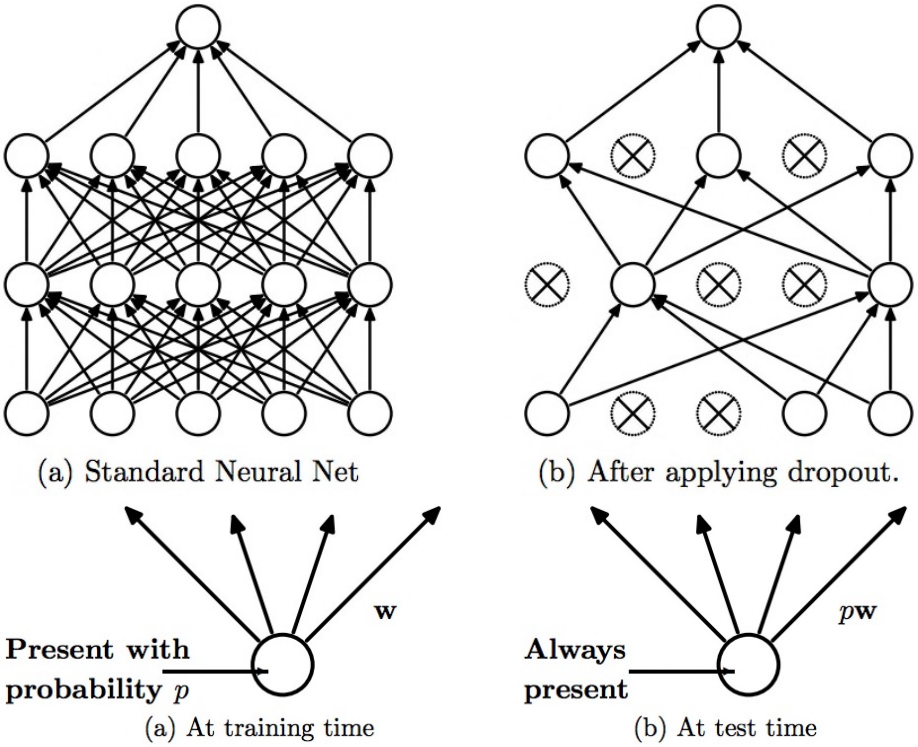


Figure 5: Dropout Images [Srivastava et al., 2014]

The easiest way to explain dropout is looking at figure 5. Dropout consists in dropping out neurons in training with some probability, that is, set to zero the

output of the neuron. Each iteration, the neurons that participates in training are different and this means each time the network topology changes. In test the weights are multiplied by the dropout probability. The output at test time is the same as the expected output at training time [Srivastava et al., 2014].

A similar technique to dropout is drop connect [Wan et al., 2013]. In this case instead of dropping out the output of the neuron we drop out some connections of the linear combination, that is, setting to zero the value of the weights.

Finally, [Goodfellow et al., 2013] propose a technique called maxout. Maxout is designed to both facilitate optimization by dropout and improve the accuracy of dropout's fast approximate model averaging technique.

Maxout is a feed forward neural network as the one described above. What change is the activation function. In this case the input to the activation function is a tensor product. The result of the activation function is the value of the maximum product computed. This technique permits implementing lots of different activation functions, each product from the tensor represent different linear parts of the activation function. Figure 6 depicts this effect.

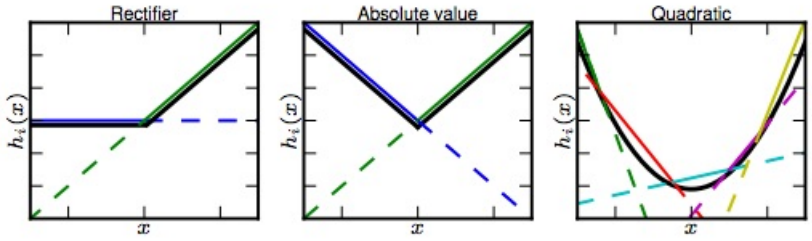


Figure 6: Max Out Activation function. Figure obtained from [Goodfellow et al., 2013]

3.4.3 Regularization

Regularization is a term that could have been included in the algorithmic section because it is something that affects the SGD directly. However, the objective of regularization is preventing overfitting and that is the reason to talk about regularization in this section.

Regularization is a term that refers to avoiding the weights of the network from having a big norm. To understand this concept think about the input of a preactivation in a neuron with a previous layer of m dimensions:

$$x^l_{pre_1} = w_1 \cdot x^l_1 + w_2 \cdot x^l_2 + w_3 \cdot x^l_3 .... w_m \cdot x^l_m + b_1 \tag{5}$$

What would happen if $w_i >>> w_j,\ i \neq j,\ 1 \leq j, i \leq m$ is that the result of the linear combination would be approximately $w_i \cdot x_i$ and we will be learning $m-1$ dimensions for nothing. To force the different weights in the network to explain the data, that is, making all the weights in the network have influence in the result there are several strategies. For sure these strategies try to avoid weights having big norms. In this section we expose three typical regularization methods: max norm, L2 and L1.

Max Norm establish the highest norm that a weight can have. If the norm of the weight is higher than a value c the norm of the vector is projected to the c norm subspace.

$$w = \begin{cases} c \cdot \frac{w}{||w||}, & if\ ||w|| \geq c \\ w, & else \end{cases} \tag{6}$$

L1 and L2 acts directly in the minimization of the cost function. Both types of regularization only change in the value of the computed norm in equation 7. Looking at equation 7 we see the expression we now minimize. This expression is a sum of the cost and a term called regularization term.

$$\min_{wrt\ p} \{C + \lambda \cdot ||p||_i\}, p \in \mathcal{P}, i \in \mathbb{N}, \lambda \in \mathbb{R} \tag{7}$$

The parameter λ controls the importance of the regularization term. The regularization term acts as a penalty on the complexity of C. Depending on if we use L_1 or L_2 we will have different properties such as the new shape of the function or the shape of the added function, if the new function is now differentiable or not... These are hard properties to study and there is lot of literature to read about. The key idea is that the greater the norm of the parameter is, the greater we penalize the cost function increasing that cost. Other point of view is that now we want to minimize C and minimize how big is p.

Finally, a very intuitive way of generalizing is called early stopping. Early stopping refers to the fact of giving enough epochs to train but not so many so our neural network does not finally learn the data. This can be easily done with cross validation.

3.5 Topology and Elements of a Neural Network

We have already talked about how can we improve the performance of a neural network with techniques that affect directly the optimization algorithm and with techniques that improve generalization. We will now see how can we make modifications (for example in the activation functions) in the neural network to improve the performance.

3.5.1 Rectifier Linear Unit

The activation function plays an important role in SGD performance. Gradient vanishing is a problem that appears in very deep architectures in part due to the derivative of certain activation functions. For example the sigmoid activation function has a 0 gradient in the saturating part of the function and very little gradient in the linear part.

Rectifier Linear Unit, [Glorot et al., 2011] appears as an activation function whose gradient get 0 o 1 value and supposed and increase in performance in very deep architectures because we do not suffer that much from vanishing gradient. Until that moment deep architectures where pre-trained with generative techniques and then refined with supervised training.

However ,the use of ReLu has several problems in deep architectures. These problems appear when lots of neurons activate as a zero because the zeros are propagated through the network and so it is the derivative. This means we do not learn anything in early layers.

3.5.2 Hyperparameter Search

Hyperparameters in machine learning are very important. In a neural network it can have high influence in the performance. Changing the number of hidden layers or the number of neurons per layer can suppose a decrease of 20-30 errors in MNIST task, for example. For that reason how hyperparameters are set is also a case of study.

The classical way of doing this was by grid search, that is, we define a space of possible hyperparameters and train a neural network with every hyperparameter and choose the best one. For sure, this is really slow. [Bergstra and Bengio, 2012] verified experimentally that performing a random search is a better approach in high dimensional feature spaces. The key idea is defining distributions over the different parameters and sample from them different the hyperparameters.

3.5.3 Batch Normalization

Batch normalization (BN) [Ioffe and Szegedy, 2015] has emerged as an important modification in the topology of the network when improving the results and the cost of training a neural network. We will pay special attention to this concept because it has shown to be a really good improvement.

BN is a simple concept, it consists in normalizing the inputs to all the layers in the network. Remember we talk about the advantages of data normalization so this go further to try to avoid several problems due to the fact that the data in the hidden layers can change widely.

Consider a network like the one in figure 1, BN for an specific batch $\mathcal{B}^l = \{B_1, B_2, ..., B_N\}$ in a layer $h^l \in \mathbb{R}^k$ is defined as:

$$h^l_{BN_i} = \gamma^l \cdot \frac{h^l_i - \mathbb{E}[\mathcal{B}^l]}{\sqrt{Var[\mathcal{B}^l]}} + \beta^l \, , \gamma \, and \, \beta \in \mathbb{R} \tag{8}$$

As we can see, it is a zero mean 1 standard deviation normalization for each batch. This try to reduce the problem call internal covariate shift (ICS). When training a very deep architecture, changes during training change the distributions in each layer. This implies a slow down in the optimization time and require lower learning rates and careful parameter initialization. Saturating nonlinearities are also included as part of this phenomenon, for example, with sigmoid if we have big parameters the preactivation results in very similar outputs of the sigmoid no matter if the differences in this high big values for each samples are big between them: we do not learn anything. Another problem related to activation functions and, following the same example, if we have data in the saturated part of the activation function the gradient is closed to zero so we suffer from gradient vanishing. People try to solve this with regularization. For that reason, BN acts also as a regularizer and in some cases eliminates the use of Dropout . BN makes possible the use of higher learning rates and be less careful about parameter initialization.

To give a bit of freedom to the learning requirements in each feature space, the parameter γ and β are included. They scale and shift the normalized value. We have the same two parameters for all the dimensions in each layer to reduce the number of parameters to learn. In some activation functions like ReLu, the γ is not needed because this activation function is only influenced by a shift. If we have a set of samples and multiply all of them by the same value, the result of the ReLu would be scaled by the same factor. These parameters can undo the normalization but that is something that is learnt during the optimization. Note that this normalization implies a limitation in what the layer can represent so this two parameters permit the network to learn anything but with the advantages of BN.

3.5.4 Convolutional Networks

Convolutional Networks are a type of FC neural network. Mathematical convolution is defined as:

$$y(s) = \sum_{t \in \mathcal{T}} x(t) \cdot h(s - t) \tag{9}$$

In our field of study, x and y represent the input and output images respectively and h is what we call the kernel. In our field of study s represent a vector of two independent variables (the position in the images) so the convolution is a

2D convolution.

Convolution is a very typical operation in signal processing due to the linearity of lots of digital and analog systems. Another typical operation is the subsampling, that is, the representation of a signal with a subset of the points of that signal.

This two basic operations are the main operations in convolutional networks. Remember the AlexNet in figure 2. In this figure we can see the three basic parts of a convolutional network: the convolution layers, the pooling layers and the fully connected part. We will go over these parts. In addition to this we will talk about new layers and the most important convolutional networks topologies proposed. One of the big advantages of convolutional networks (apart from the fact that they implement a convolution that is used to extract relevant features from images and signals in general) is the fact that all the weights are shared for all the pixels in an image. In a FC network if we have a $28 \cdot 28$ image we will have 784 different weights for each neuron in the next layer. In a convolutional network if we have only one kernel of $3 \cdot 3$ pixels we will only have 9 weights to learn in that layer for each of the images that conform the input to that layer. This means that the weights can be well trained in deep architectures although we have the gradient vanishing problem present.

3.5.4.1 Convolution Layer

The convolution layer is where the convolution takes place. The convolution layer is defined with the number of kernels to learn, that is, the number of h in equation 9 and the size of these kernels, for example, a $3 \cdot 3$ kernel is a square kernel with 9 pixels. Additionally, the number of kernels to learn defines the number of output images. The size of the input map, that is, the number of input images changes the convolution from a 2D convolution to a 3D convolution. To understand how the convolutional layer works lets look at figure 7. As we can see we have N output images that correspond to the number of convolutions to perform. Each convolution is a 3D convolution so we have 3D kernels whose shapes are Kr Kc and M.

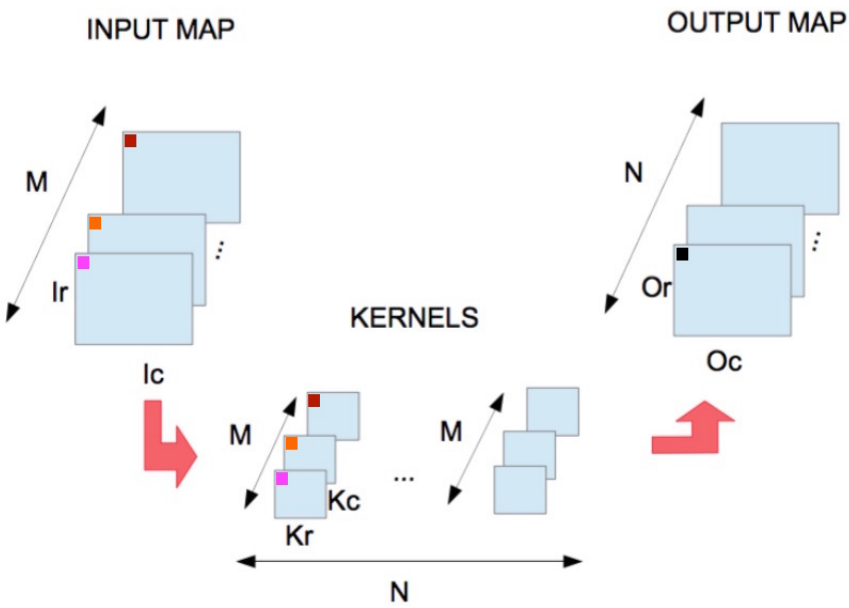


Figure 7: Graphic representation of the Convolution Layer operation

To end with the convolutional operator lets write the 3D convolution operation for the black pixel in the output map in figure 7. Let's call the output image O_1, the set of input images $\mathcal{I} = \{I_1, I_2, ..., I_N\}$ and the corresponding set of kernels for O_1, $\mathcal{K} = \{K_1, K_2, ..., K_N\}$. Ω represents the domain and it depends on the shape of the kernel.

$$O_1(r_1, c_1) = \sum_{u,v \in \Omega} I_1(u,v) \cdot K_1(u - r_1, v - c_1) + \\ + I_2(u,v) \cdot K_2(u - r_1, v - c_1) + \ ... \ + I_m(u,v) \cdot K_m(u - r_1, v - c_1) \tag{10}$$

3.5.4.2 Pooling Layer

Pooling layer is the layer where the subsampling takes place. It takes as input and image of shape $N \cdot M$ and outputs and image of shape $K \cdot P$ with $K < N$ and $P < M$. Each pixel in the new image has a relation with the pixels of the input image. Depending on this relation we can find different pooling layers. We will classify the pooling layers depending on how we combine the pixels of the input image. Each pixel in an output image is compute from the neighbors pixels in the input image.

- MaxPooling: The output pixel is the one with the biggest value in the input image neighbor.

- AveragePooling: The output pixel is an average of the neighbor pixels of the input image.

Usually each output pixel is compute from non-overlapped input neighbors.

The goal of the pooling layer is reduce the computation cost, allow the possibility of extracting features at different scales and capture high level features. These operations give also invariance to affine transformations such as translations.

3.5.4.3 Fully Connected: Reshape Layer

As we can see in figure 2 at the end of the network there is a fully connected part. This fully connected part is to connect the extracted features in the convolutional part to the class to perform the classification. Remember we are talking about discriminative training so our whole purpose is to extract features in any way to have a good classification error.

The reshape layer takes as input an image of arbitrary shape, $M \cdot K$ and perform a flattening to create a vector in $\mathbb{R}^{M \cdot K}$. It does this for each image, $\mathcal{I} = \{I_1, I_2, ..., I_N\}$, in the map and concatenate all the vectors so at the end we have a vector of shape $(1, N \cdot M \cdot N)$. This vector defines the input shape to the fully connected part.

3.5.4.4 Other layers

In this section we will talk about new layers that has been appeared recently: cat layer, agregation layer, $1 \cdot 1$ kernel size.

The most confusing is the $1 \cdot 1$ kernel (11K). A convolution with a 11K its only multiplying each pixel by a number so we are not performing any kind of correlation. However, remembering equation 10 a 11K will make a weighted sum of the pixels of the images of the input map at every position and the result will be an image that have a combination of all the previous images in one. What we are doing is a linear combination of the same feature in different images in a map.

The cat layer is a simple layer that takes as input different maps of the same shapes but obtained from different convolutions and create a new map that include all the previous maps. We will see an example of application of this layer after.

The most recently layer is the aggregation layer. The aggregation layer performs something similar to convolving with a 11K kernel. Suppose we have a set of inputs map computed from different convolutions. This maps must have images with same shapes and each map must have the same number of images. The aggregation layer performs the next operation:

Lets have a set of input maps $\mathcal{IM}_1 = \{I_1, I_2, ..., I_k\}$, $\mathcal{IM}_2 = \{I_1, I_2, ..., I_k\}$, ..., $\mathcal{IM}_p = \{I_1, I_2, ..., I_k\}$ and let $I_1 = \mathcal{IM}_1(1), I_2 = \mathcal{IM}_1(2)...$ After the agregation layer we will have an output map defined as $\mathcal{OM} = (\mathcal{IM}_1(1) + \mathcal{IM}_2(1) + ... + \mathcal{IM}_k(1), \mathcal{IM}_1(2) + \mathcal{IM}_2(2) + ... + \mathcal{IM}_k(2), ..., \mathcal{IM}_1(p) + \mathcal{IM}_2(p) + ... + \mathcal{IM}_k(p))$

3.5.4.5 Main Convolutional Topologies

To end with convolutional network we will highlight the most important topologies and networks that are showing to behave well in different problems and are the winners of ILSVRC competitions.

We have already talked about AlexNet that was the first convolutional network. Some people used the convolutional output of this network as input to other classifiers. It is a network that has good data representation.

The ILSVRC2014 winner was the GoogleNet [Szegedy et al., 2014a]. This network introduced the cat layer and also combined gradient in SGD. It computed cost at different levels of the network and minimize the sum of all this costs. It also introduces the 11K.

The ILSVRC2014 second position was the OxfordNet (Vgg) [Simonyan and Zisserman, 2014]. It was the best single model. This model implemented only $3 \cdot 3$ kernel shape convolutions. To have an effect similar to what a $5 \cdot 5$ kernel shape convolution do what they did is put two or more $3 \cdot 3$ kernel shape convolution layers. But, the most important thing of this networks is that the way of creating the topology of the network has shown to be a good point of start in the creation of topologies for other networks in other tasks.

Finally the ILSVRC2015 winner is the microsoft network, ResidualNet [He et al., 2015]. This network introduced the agregation layer.

3.6 Generative networks

The final part of our state of the art revision of neural networks for computer vision is dedicated to generative learning.

Neural networks for classification are classically trained in a discriminative way. However, when going deeper in the topology we comment that the learning process suffer from gradient vanishing and saturated activations. We tried to avoid the saturated activations with regularization but it was with the appearance of the rectifier linear unit when we could start going deeper in the topology without suffering from gradient vanishing and saturated activation effect.

Before this, the problem of the gradient vanishing made that the early weights in the network where very little modified, that is, we were not learning. To

solve this problem researchers started to explore the use of unsupervised data to initialize properly the weights from the network so when learning the discriminative task this little modifications in the early weights were helpful for the task.

Using the unsupervised data results in changing the paradigm of learning. We could not perform discriminative learning and change the techniques to start learning generative neural networks (GNN). There were several GNN networks to perform this initialization and in this section we will explore the two most important.

This generative models trained with only unsupervised data could only be able of explaining the data structure by definition. This section is dedicated to show how this generative networks are trained to represent data and how are then used to create a discriminative model. We will start with Restricted Boltzmann machines and Deep Boltzmann Machines and then talk about Deep Belief Networks. We will end this section talking about autoencoders.

3.6.1 Restricted Boltzmann Machines

Restricted Boltzmann Machines (RBM) is an energy based model, see ***Appendix 3*** for an explanation, whose energy function is given by equation 11:

$$E(x, h) = -b^T \cdot x - c^T \cdot h - h^T \cdot W \cdot x \tag{11}$$

Where b and c represent the biases and W represent the matrix of weights connecting the visible and hidden units. This equation can be graphically interpreted as in figure 8 where each connection represents a number from matrix W and the biases are added to each linear combination but they are not present in the figure.

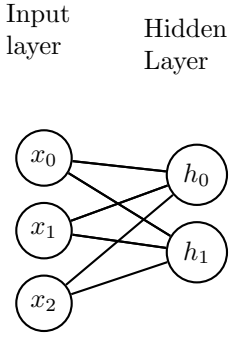


Figure 8: RBM topology

The derivative of the log likelihood is easy to obtain. Our final expression for the gradient wrt a weight, w_{ij}, of the free energy function in an RBM for one training sample X is given by equation 12:

$$\frac{\partial F(x)}{\partial w_{ij}} = \sum_{\forall h} p(h|x) \cdot x_i \cdot h_j \tag{12}$$

And the expression of the gradient of the log probability is:

$$\frac{\partial \log p(x)}{\partial w_{ij}} = \sum_{\forall h} p(h|x) \cdot x_i \cdot h_j - \frac{1}{|\mathcal{X}|} \sum_{\forall (x',h) \in \mathcal{X}} x'_i \cdot h_j \tag{13}$$

The next step is on how we compute those expectations. There is more technical information in [Hinton, 2012] [Bengio, 2009] and the deep learning tutorial (http://deeplearning.net/). Basically we have to sample h from $p(h|x)$ and pairs (x', h) from $p(x', h)$.

We will briefly expose an introduction on how we can do that, with a wider and more technical explanation in the above references. The expectation over the data can be easily computed from the training data and taking in consideration the conditional independence between the layers in the RBM. Given a training data X the probability of the dimensions in a hidden sample is conditional independent, that is:

$$p(h|x) = \prod_{j=1}^{k} p(h_j|x) \tag{14}$$

The conditional probability can be computed using the joint probability distribution $p(x, h)$ given by an energy based model with energy function given by equation 11, and its corresponding partition function, and the marginal distribution $p(x)$. We can find a good explanation in [Bengio, 2009] which end up showing that RBM fits with equation 14 and that in the RBM this conditional expectation is given by:

$$p(h_j|x) = \frac{e^{h_j \cdot (c_j + W_{\cdot j} \cdot x)}}{\sum_{\forall \tilde{h}_j} e^{\tilde{h}_j \cdot (c_j + W_{\cdot j} \cdot x)}} \tag{15}$$

where depending on the problem can take well known defined functions. In a binary case this function boils down to the sigmoid function. Note that the graphical representation of the RBM is that because of this conditional expectation (note that there are no connections between the dimensions of a layer, that is a path from x_i to x_j). With some tricks we end up showing that the first term of the gradient of the log likelihood have an analytic form and is given by equation 16. Using equation 15 we can compute the gradient of this part. This means that for RBM we do not need to sample to compute this expectation.

$$\sum_{\forall h} p(h|x) \cdot x_i \cdot h_j = p(h_j = 1|x) \cdot x_i \tag{16}$$

We end up taking a training sample X_m. Compute the posterior probability of a sample $H_j = 1$ that depends on the parameters c_j and the j column from matrix W. This column represents the weights of the linear combination of the elements of X for computing the posterior probability of $H_m(j)$. We do this for each sample X and end up computing $\mathbb{E}_x[x_i \cdot p(h_j = 1|x_i)]$.

The second term has to be an approximated term. An easy way is with a finite set, $\mathcal{X} = \{(X_1, H_1), (X_2, H_2), (X_3, H_3), ..., (X_N, H_N)\}$. In this case there is no analytic solution for the binary case for this expectation and needs to be approximated.

$$\mathbb{E}_{x',h}[x_i \cdot h_j] = \frac{1}{|\mathcal{X}|} \sum_{\forall (x',h) \in \mathcal{X}} x'_i \cdot h_j \tag{17}$$

Sampling the pairs from the distribution $p(x', h)$. The problem is how we sample the data to form that set. This data has to be sampled according to the probability distribution of our model and that means x needs also to be sampled. [Hinton, 2012] said we can do that using Gibbs sampling. In [Hinton, 2002] an algorithm called Contrastive Divergence is proposed to train RBM. Note that the same conditional probability expression holds for $p(x|h)$ changing c_j in the expression by b_j and other little modifications, see [Bengio, 2009]. Instead of taking and arbitrary value of x and generate h for computing this expectation we can also select a training vector X and compute H using $p(h|x)$ and then reconstruct X using $p(x|h)$ (note that using the original X does not make sense because it has to be generated by our model). Reconstructing X by projecting a training sample to the hidden space and then reconstruct has shown to accelerate the learning process. All this sampling methods are used for getting unbiased samples of the distributions. The underlying idea is generating our set by a MCMC exploration based on gibbs sampling which end up giving a set of observed and hidden samples which approximate the distribution $p(x', h)$. As long as we sample towards ∞ we approximate the real distribution. The MCMC has already converged when we start at a sample X from the training data and the empirical and model distribution are closed [Bengio, 2009]. Once we have computed this expectation we have the gradient and we can apply SGD.

Finally, deep boltzmann machines [Salakhutdinov and Hinton, 2009] are the same as RBM but with more hidden layers. We have briefly shown how a very popular unsupervised method is defined. This can be used to create a network that is the used for supervised learning. We will talk about this in the final section.

3.6.2 Deep Belief Networks

Deep belief networks [Hinton and Salakhutdinov, 2006] are a simple way of creating deep architectures making use of what we explained about RBM. Instead of training a deep boltzmann machine we train different RBM and then stack

them together.

To better understand this part lets look at figure 9. The whole training process is divided in three different parts:

- First several RBM are trained in this way. The first RBM is trained and then all the training data is projected to the latent space. The next RBM is trained with this projected data. We continue this process and generate the desired deep architecture.
- Second step is unrolling the deep architecture, that is, put the same architecture we have trained but with the projections in the opposite way (transpose the projections matrices). This part is known as the decoder.
- Last step is fine-tunning, this fine-tunning is done forcing the network to reconstruct the input at the output.

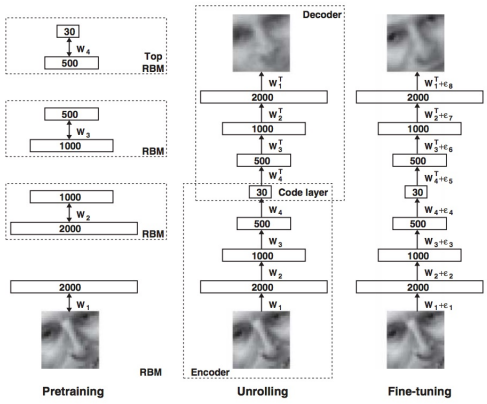


Figure 9: Deep belief network. From left to right we observe the pre-training, unrolling and fine-tunning steps, [Hinton and Salakhutdinov, 2006]

3.6.3 Autoencoders

Autoencoders are the last two parts of the DBN (the second part is the architecture and the third is how we learn). With the appearance of the Rectifier Linear Unit (remember it helps avoid gradient vanishing) we could directly train the network by making it reconstruct the input. The process of learning is the same as the fine-tunning step, we learn to reconstruct inputs. Without rectifier linear unit a way of training each layer was the same as in DBN. We train a pair of layer, then project all the data and train the next layer.

In addition, a good way of improve generalization is by modify the input adding it noise but force to reconstruct the original input. This is called denoising autoencoders. In this case the cost function compares the original signal x with a reconstruction signal $\bar{x}$ computed from a corrupted version of the input $\widetilde{x}$. The underlying idea is that the representations should be robust to noise. If a dog, for example, is partially occluded we should still recognize a dog.This was proposed by [Vincent et al., 2008] and we can find an extension in [Vincent et al., 2010]. In multilayer denoising autoencoders we could train the whole network or train pairs of layers with corrupted versions but then project the clean data (not the corrupt), corrupt the projection and learn the next two pairs of layers (this will be useful for very deep architectures).

Normally in autoencoders and DBN the weights of the encoder and decoder are tied. This means that the projection matrix of layer l should be the transpose of the corresponding layer $L-l$, that is in a 5 hidden layer network $W^1 = W^{6\intercal}$.

3.6.4 From Generative To Discriminative Training

It is easy to use this generative networks to create a classifier. No matter if we have a RBM, a DBN or an autoencoder we proceed in the same way.

Imagine the MNIST problem, which is a classification task in ten different numbers. We create one of these topologies with the middle layer make from ten neurons. We then train the generative model as explained and then use only the encoder part. With the use of supervised data we know perform fine-tunning changing the training criteria (probably use cross-entropy) but using the weights already trained.

We will see after how these techniques outperformed classical supervised techniques.

4 Resources

This short chapter is dedicated to talk about the resources involved in the work. On one side we will talk about the databases used in the experimentation. On the other, we will describe the hardware and software computation resources.

4.1 Databases

We have use two databases: MNIST and CIFAR10. There is another version of CIFAR which is the CIFAR100. We use these two databases because they are the ones used by [Rasmus et al., 2015] in their experimentation. This permits comparison between our results and their results. We thought in trying it with the imageNet database. However we will see after why was this unfeasible.

MNIST [Lecun and Cortes] is a database for HTR (*handwritten text recognition*) task. It is composed of $28 \cdot 28$ pixel images, that is, vector space of $\mathbb{R}^{784}$. Their content is all the alphanumeric numbers. They are gray-scale images with 8-bit quantification. In the database they are 60000 images for training and 10000 for test. We can see an example of a test MNIST in image 10. Wider description could be find in [Lecun and Cortes].

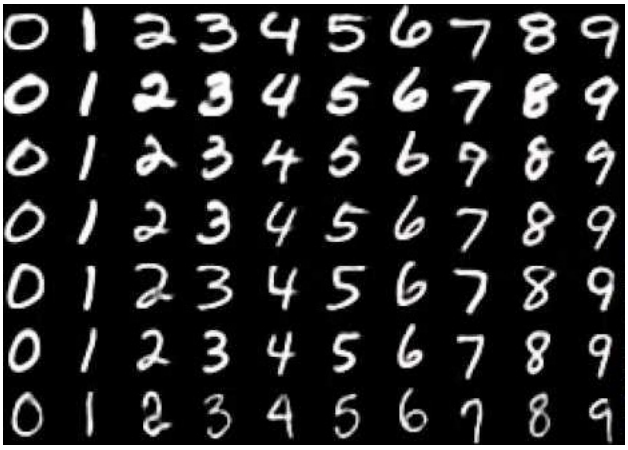
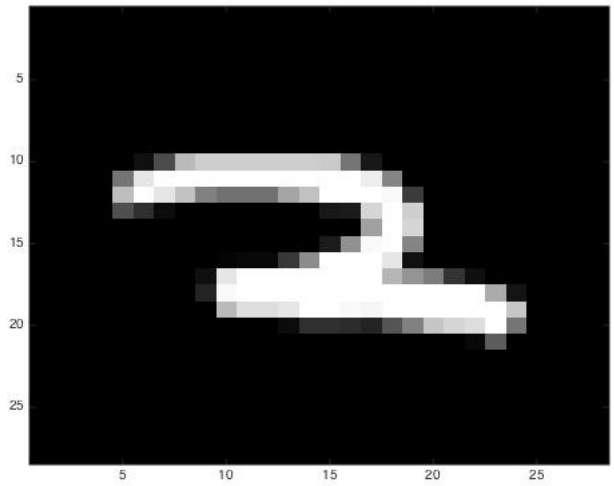


Figure 10: Example of MNIST figures

In figure 11 we can find an example of an MNIST image at scale. The purpose is to show the degree of detail present in an image. It should be clear that if for a human a computer vision task is harder, it would be also for a machine.



(a) Example of the detail of a MNIST image. Pixels are the monotic gray boxes that we see in the image

(b) Example of how we see a number in a 15,4 inch $(2880 \cdot 1800)$ computer screen

Figure 11: A sample from MNIST database

CIFAR10 [Krizhevsky, 2009] is a database of $32 \cdot 32$ RBG pixel images. This means we have a vector space of $\mathbb{R}^{3072}$. It is make from 60000 images with 50000 for training and 10000 for test. It has 10 classes with 6000 images per class. The classes we can find are the next ones: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck.

As before we present an example of CIFAR-10 image in the next figure:

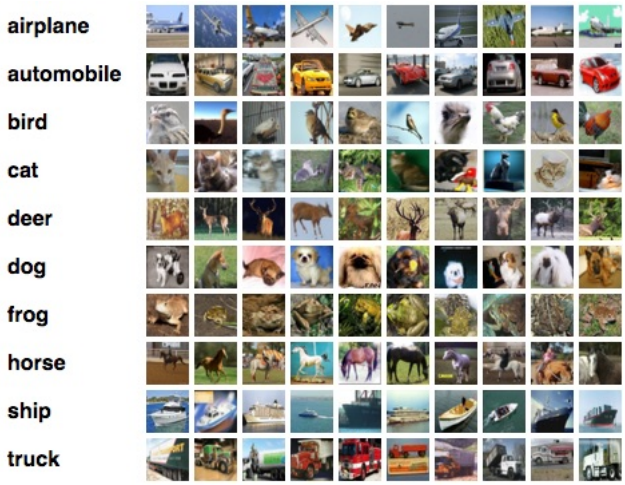
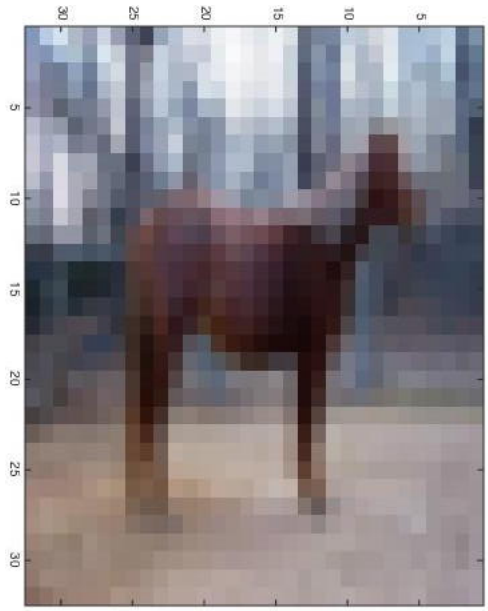


Figure 12: Example of CIFAR10 images



(a) Example of the detail of a CIFAR10 image (in this case is a horse). Pixels are the monotic color boxes that we see in the image

(b) Example of how we see a horse in a 15,4 inch $(2880 \cdot 1800)$ computer screen

More information about this database can be find in chapter 3 of [Krizhevsky, 2009].

4.2 Computation Resources

This subsection is dedicated to explain the computation resources involved in the experimentation. First we will talk about the software resources and after about the hardware resources.

4.2.1 Software Resources

We have use the code provided by Rasmus available at `https://github.com/arasmus/ladder`. This code is made with blocks, which is wrapper on top of Theano [Theano Development Team, 2016], and fuels [van Merriënboer et al., 2015]. Blocks is used to basically create the computation graphs and fuels is used to provide the data to the machine learning algorithm.

We use the bleeding edge Blocks Version that depending on the operative system changes littetly. For example in ubuntu 16 we use theano 8.0 version and in ubuntu 14 theano 7.0 version. We could find the requirements for the installation on the website. What is important is that the blocks version we use is different from the author's version so we have to first update the code. This also influences the results as we will see later. Neural network weights are randomly initialized so different versions can lead to different initialization and to different results. That is why we have done our own experiments with the baseline model. These results are quite different from the ones in the paper.

As we said, Blocks is used to facilitate the creation of computation graphs which is done with Theano. This computation graphs are made in C code. Theano only made easy the creation of this computation graphs. Is an abstract representation of this computation graphs. Working with theano is similar to the creation of finite state machines in which each state could represent a variable, an update of a variable... This means Theano is symbolic. For example when creating the input to a neural network, we create the variable specifying the type and the shape of that vector, but we cannot see examples of this variable (for example a vector representing an image) until we start running the algorithm represented by the computation graph. We draw the computation graph but we cannot use it until it is compiled in C code.

Theano is able of working with GPU and CPU. For the use of GPU we installed the CUDA 7.5 and the cuDNN. Cuda 7.5 is an API for programming NVIDIA GPUs. It provides libraries, compilers... depending on the OS. cuDNN (cuda deep neural network) is a library for accelerate computing in deep learning task. For example it makes convolutions faster using the different operation modules available in the GPU. This was all installed over a 64 bit linux operating system.

We use 14.04 version of the Ubuntu distribution and the Ubuntu 16.04. In this operative system we use CUDA 8.0.

4.2.2 Hardware Resources

The most important hardware resource for this task is the GPU. We have use an NVIDIA GeForce GTX 980. This GPU has 2048 CUDA Cores (see CUDA computing platform) and 4GB internal memory. The rest of specifications can be found at `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications`.

The CPU used is a Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz which has 4 kernels with 2 threads per kernel. The memory RAM used is a DDR3 type with 1600 MHz clock speed. The total amount of RAM installed is 16 GB. The CPU and RAM memory are not so important for the computing task but having good resources at this level allows faster memory transfer between the GPU and the general RAM memory, which normally suppose a big overhead in computing tasks, as exposed in the deep learning tutorial.

We had another available computer with a GPU NVIDIA GeForce GTX 750 Ti. This was also used during experimentation until we notice that the results where not exactly the same. This was notice with the first experiments so we could fix quickly and perform the rest of experiments in the first computer.

We ended up using another computer with a GPU NVIDIA GeForce GTX 1080, 16 GB DDR4 2133 MHz clock speed memory ram and a Intel(R) Core(TM) i3-6100 CPU @ 3.70GHz which has 4 kernels with 2 threads per kernel. This GPU has 2560 CUDA Cores and 8GB memory. We can find a comparison between the 980 and the 1080 and the full specs for the 1080 at `http://www.geforce.com/hardware/10series/geforce-gtx-1080`.

5 Mathematical Foundations

The objective of this chapter is to explain the two mathematical frameworks in which our work is based. We are not going to make an exhaustive analysis of this two themes but to try to explain and draw important conclusions to understand why this works well and why mixing this to things have sense. For more detailed information there are good papers to find the information.

The first section would be dedicated to latent variable models and how can we use them to make a good neural network model that work really well with supervised and unsupervised data at the same time. This section will occupy most of the chapter and we will make use of [Bishop, 1999] and [Bishop, 2006] to explain this models. The final part of the chapter will be dedicated to adversarial noise and why this kind of noise works really well in supervised models.

5.1 Latent Variable Models

A latent variable model is a way of defining probabilistic models in which we have a set of observed variables and a set of hidden or latent variables. By defining joint distributions over this set of variables we are capable of creating a model of the observed variables by marginalization over the hidden variables [Bishop, 1999].

There are several reasons to focus our attention in latent variable models. We will talk about a set of samples, $\mathcal{X} = \{X_1, X_2, ..., X_N\}$, drawn from a probability distribution we want to infer. The more expressive is our model, the more we can say about the underlying probability distribution of our data set. We have, for sure, to take care about overfitting but from now on we will think that our data samples are drawn from really complex probability distributions.

Let's think in the MNIST problem where we have images of $28 \cdot 28$ pixels. This means we have 784 dimensions we have to model. If we think in a very simple model, a gaussian model, we will have to estimate in the worst case $784 + 784^2$ parameters for each gaussian explaining each number. Maybe we can assume that the components of the images are independent and only have $2 \cdot 784$ parameters but, for sure, this assumption does not make sense in written data because the position of a pixel in a number is related to the positions of the other pixels in the surroundings. This means we have to estimate $10 \cdot 615440$ parameters with a data set made from 60000 images. These parameters will not be well estimated when using maximum likelihood estimation, for example. Latent variable models are a way of catching the important information in the underlying probability distribution but in a lower dimensional space and that means we have to learn a less number of parameters. In this case we will have to learn lower dimensional gaussian distributions and the relations between the

hidden and visible variables. In the RBM the CD-1 algorithm is used for learning these relations.

Another important characteristic of latent variables models is that we can make more powerful models because they permit representing data at different levels so each level can center in capturing different characteristics of the data. Neural networks are in some way latent variable models. Finally, as we said, we can define joint distributions over the latent and visible data and obtain the probability distribution of the visible data by marginalization. This means we can decompose this joint distribution using bayes rule to have simple models that catch prior information about the variables and the relations between the different variables. For sure, modeling these simples models is much more easy than trying to model directly the joint distributions.

Refreshing our notation we will use the x for the input data and the h for the hidden data. In this context x would be the visible units and h will represent the latent units. Equation 18 represent the basic mathematical operation we have to perform to compute the probability distribution of the visible data from a joint distribution. Depending on how $p(x|h)$ and $p(h)$ is defined, we will have analytic solution for this summation or we will have to use iterative methods such as EM algorithm [Dempster et al., 1977].

$$p(x) = \int_{h \in \mathcal{H}} p(x, h)dh = \int_{h \in \mathcal{H}} p(x|h) \cdot p(h)dh \tag{18}$$

We should take in consideration that we can have latent variable models at different levels in which we could have $p(h)$ being dependent on other latent variables that could be dependent or not between them. Equation 18 catches all this information.

We now have to model the two terms in the product. Normally, $p(x|h)$ is given by a relation between x and h. This relation can be expressed like in equation 19.

$$x = f_{\mathcal{P}}(h) + u \tag{19}$$

Where $f(\cdot)$ represent a function of the latent variables and u is h independent noise process but that could depend on other variables. This mapping from the latent variable to the visible variable is probabilistic due to the fact that u is a stochastic process so this could model $p(x|h)$ where the mean is given by the projection of h and the variance is related to the noise process. Note that if the components of u are uncorrelated (for example a white gaussian noise) and given all the components of h, all the components of $x \in \mathbb{R}^k$ are independent. We could factorize the probabilistic model as in equation 20.

$$p(x|h) = \prod_{i=1}^{k} p(x_i|h) \tag{20}$$

On the other hand $p(h)$ represent prior information about the latent variables, that is information only dependent on the structure of this latent variables and their relations. For example if we do not know any information about the latent variables our prior information could be expressed in terms of a non-informative probability distribution like the one in equation 21.

$$p(h) = \frac{1}{|\mathcal{H}|} \tag{21}$$

5.1.1 Going Deeper in Latent Variable Models

To end with this briefly introduction to latent variables we would talk about the learning a generative process. To this point we should remark that a latent variable model is defined if we have $p(u)$, the mapping of equation 19 and the prior distribution $p(h)$.

5.1.1.1 Generative Process

The generative process is quite simple. We drop a value h with a probability distribution given by $p(h)$. We then have to compute x. We use equation 19. As we can see this equation represents the distribution $p(x|h)$ with mean given by $f_{\mathcal{P}}(\cdot)$ and standard deviation given by the noise process. This noise process could be $u = \mathcal{N}(0, \sigma \cdot I)$. If we have a noise of this class we could factorize the conditional distribution as in equation 20.

5.1.1.2 Learning Process

A typical way of learning supervised or unsupervised generative models such as the latent variable models presented in this section is by maximum likelihood estimation. Given a dataset $\mathcal{X} = \{X_1, X_2, X_3, ..., X_N\}$ the likelihood is given by:

$$\prod_{i=1}^{N} p(X_i) = \prod_{i=1}^{N} \int_{h \in \mathcal{H}} p(X_i|h) * p(h) dh \tag{22}$$

Where we can use the prior $p(h)$ to compute the posterior distribution $p(h|\mathcal{X})$ using bayes rule. The EM algorithm make use of these distributions to learn the set of parameters from $f_{\mathcal{P}}(\cdot)$ and the parameters from the noise process. If we have a linear function for computing the mean of $p(x|h)$ our set of parameters could be $\mathcal{P} = \{W, b, \sigma\}$. This learning process is divided in an inference process (inference of the latent variables given the inputs $p(h|x)$), that is computing the posterior probability of the latent variables given the observed variables, and a learning process that is maximizing the underlying probability distribution of the latent variables to fit better the observations.

In the next section we will show how the latent variable models can be used to create a neural network topology that uses supervised and unsupervised data at the same time. We should remark that, as we have seen, that latent variable model are generative models so we will now see how we can create a neural network with a generative part that will take the basic idea of latent variable models and this generative part uses what the supervised part finds suitable for the task at hand to find new features that correlates well with the already found features.

5.2 Ladder Networks

This section is dedicated to explain ladder networks. Ladder networks (LN) are a kind of autoencoder. The main particularity of LN are the lateral connections between the different layers in the encoder and the decoder, see figure 14. We will go over the different parts of this model topology, but for the moment we should look at the three different parts of this autoencoder, from left to right: the corrupted encoder, the decoder and the clean encoder.

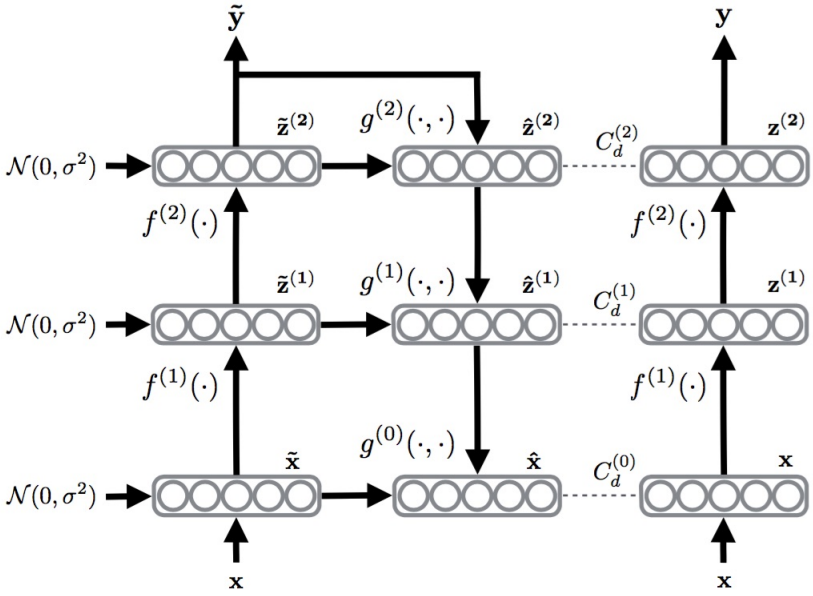


Figure 14: Ladder Network Topology: encoder and decoder. Figure obtained from [Rasmus et al., 2015]

The explanation will be based on [Rasmus et al., 2015] where we can find the state of art model and on [Valpola, 2015] where we can find the mathematical foundations for the model.

5.2.1 Semi-Supervised Learning

The first part of our explanation is dedicated to establish and refresh what is the objective of the learning process we will explain, which is basically mixing supervised and unsupervised data at the same time.

When using unsupervised data we can only handle information about the data representation. We are not capable of improving the task we want to perform because unsupervised data, by definition, does not have information about the task. This means that unsupervised learning will learn relevant features for representing the data.

On the other hand, supervised learning is used for performing a task. This means that the features we learn are those features that are discriminative for the task at hand. We could have the same data set but two different tasks and we could see that the network is learning different features.

As we said before, researchers has been using unsupervised data due to the fact that there is more unsupervised data than supervised data and because there is much more information in the input than in the targets which define the task. The targets, for example in MNIST, only have information about what number is a specific sample, but the input, that is the image, have all the pixels which contain the information about the shape, gray levels, correlations.... The problem is that mixing supervised and unsupervised data was done in a way that first unsupervised data was used to create a network that represents data and then refine the features learnt by unsupervised learning to fit with the task, this means selecting the most discriminative features that unsupervised learning has learnt. For example if we want to difference between young and older people by face pictures we could learn that a face have a nose, eyes, the nose is below the eyes in the unsupervised pre-training. Probably, when the supervised part takes place we will miss the information relative to the position of the nose and learn how big or pointed is a nose.

After the supervised part takes place, the unsupervised features cannot help anymore in the learning process due to the different paradigms each learning process try to achieve. While unsupervised learning tries to extract good features to reconstruct, supervised learning uses certain features for the task at hand. If we try to mix both of them what would happen is that the unsupervised part will select features that maybe are not necessary for the discriminative task and probably the learning process will not converge.

So the objective is to try to find a new paradigm of learning so mixing unsupervised and supervised learning at the same time have sense. The key point is, how can the unsupervised learning helps the learning process when the supervised part has started?. The unsupervised part should be capable of finding new features that correlate with the already learnt features found by the unsupervised

part that have shown to be important to the task, due to the supervised part. A good example given by [Rasmus et al., 2015] is imagine that the supervised part has found that an eye is important to recognize a face in an image. The unsupervised part should be capable of finding other features like the nose or the mouth that helps the generalization. The supervised part should establish if this new features are good or not for the task. So in some way we are doing what other people did when pre-training with unsupervised data but in this case this pre-training is being done at the same time we select the features.

So the paradigm changes in this way. Before the supervised part takes places, unsupervised learning should find all the features that carry as much information about the input as possible. After supervised learning starts and shows preference for some features, unsupervised learning should find features that correlate with the features supervised learning has shown a preference. This means that unsupervised learning should be capable of **discarding information**. The unsupervised model that can deal with this task is hierarchical latent variable models, which are an extension of latent variable models that we will study in the next section. We will now see why this model can discard information.

5.2.2 Hierarchical Latent Variable Models

Let's refresh our latent variable model from equation 19.

$$x = f_{\mathcal{P}}(h) + u \tag{23}$$

In this model we have a mapping from the latent variable given by the function $f_{\mathcal{P}}(\cdot)$ and this mapping is governed by some parameters, $\mathcal{P}$. On the other hand u represent a noise process. We said this could represent $p(x|h)$.

$$p(x|h) = p(u) = p(x - f_{\mathcal{P}}(h)) \tag{24}$$

As we can see, the latent variable models are stochastic and this property is given by u. With the same projection $f_{\mathcal{P}}(h)$ we could have different observed variable. These models are trained by defining a cost which takes in account the reconstruction error between x and $f_{\mathcal{P}}(h)$ (we will give after a reference for an example) or by probabilistic modeling (like what the EM does). When training this models the probability $p(u)$ represent the variability of the possible observed data for a given latent variable. This is why we can express the probability model as in equation 24. We see that the noise is the reconstruction error and should be a probabilistic process due to the fact that the observations are given by a probability distribution, that is, they are not deterministic.

This latent variable models are data representation models. The objective is to let this models discard information. A one latent variable model have lots of trouble in discarding information because it needs to keep all the possible

information to keep the reconstructing error low. They are also not capable of focusing on abstract invariant features because for example for reconstructing a face we need information about position, orientation, shape... that are not abstract invariant features.

This can be solved with hierarchical latent variable models.

$$p(h^l|h^{l+1}) \tag{25}$$

Where now the relation between variables is given by:

$$h^l = f^l_{\mathcal{P}}(h^{l+1}) + u^l \tag{26}$$

As we can see each latent variable of each level can focus on different features. We will refer as lower levels to the levels close to the input and higher to the opposite. Now, lower levels can focus on lower level features such as position and higher level on invariant features such as having a nose is important. Each level is capable of discarding information as long as it is needed. This means that when supervised learning takes place it would be easy to extract the relevant features for the task at hand, minimizing the influence of the objective of unsupervised learning. Some information that is not needed for the supervised task in level l could be represented in some way with the other levels, so the unsupervised objective can also be achieved. Note that each variable is a combination of a function of the latent variable and a noise process, this means that at each level of the hierarchy the latent variable can add stochastic information and that is why we can discard information in a level but be able of having all the necessary information to keep the reconstruction error low. It should be clear how powerful is this model.

The problem of this kind of models is in computing the posterior probability of the latent variable and the parameters. We should remember that the aim of the learning process is learn the projections to the subspaces that have good representation of the data. We have to model $p(h|x)$ which is the posterior of the latent variables. We can do this with the likelihood information of the data and prior information.

$$p(h|x) = \frac{p(x|h) * p(x)}{p(h)} \tag{27}$$

This operation is mathematically intractable so we have to approximate its solution. Sometimes there are closed solutions based on the EM algorithm, for example in gaussian mixture models. As we said we can also learn this models by minimizing the reconstruction error.

5.2.3 From Autoencoder to Hierarchical Latent Variable Model

In this subsection we will see how to create a neural network topology based on an autoencoder that implements the underlying idea of hierarchical latent

variable model.

Remembering autoencoders, this kind of neural network have an encoder part where the projections between the subspaces are given by:

$$h^{l+1} = f^l_{\mathcal{P}}(h^l) \tag{28}$$

and a decoder part given by:

$$\bar{h}^l = g^{l+1}_{\mathcal{P}}(\bar{h}^{l+1}) \tag{29}$$

where f and g represent the same kind of function but for clarity we will use f for encoder and g for decoder (also known as the reconstruction function). Moreover, these functions are of the form:

$$f(x) = \mathcal{A}(W \cdot x + b) \tag{30}$$

where W represent the weights of the projection, b is the zero order term and $\mathcal{A}$ is the non-linear projection. In this model $\mathcal{P} = (W, b)$ are the set of parameters to be learnt. We should note that now h represent hidden deterministic variables, in this case we do not have stochastic variables.

We could think that these models are somehow similar to latent variable model in equation 26. The main difference is that while autoencoders are deterministic projections between latent and observed variables, latent variable models have a stochastic relation. Note that in an autoencoder for a given h there is only one possible x while for a latent variable model there are infinite possible x each one with different probability. The noise process u is what give the latent variable the possibility of adding new information in comparison to standard autoencoders. This means that autoencoders resemble in some way one single latent variable model were discarding information is not possible.

What [Valpola, 2015] propose for making the autoencoder be similar to latent variable model is combine information from the bottom-up path. Remember equation 18. The latent variable model combines prior information over the latent variable and likelihood information of the observed variable. On the other hand an autoencoder only have top-bottom information so the solution is to add a connection between the encoder and the decoder to have an influence from the bottom-up path. Now the reconstruction function is:

$$\bar{h}^l = g^{l+1}_{\mathcal{P}}(\bar{h}^{l+1}, h^l) \tag{31}$$

This means that now the higher levels does not need to represent everything needed in the reconstruction. The function $g_{\mathcal{P}}(\cdot)$ can learn to combine encoded abstract information in higher levels with detailed features to reconstruction. Higher layer can then focus on more invariant features suitable for the task and the model can now **discard information**, that was the proposed objective for

this section.

This is a good way of simulating latent variable models with a neural network. In this case $\bar{h}^{l+1}$ will represent in some way the prior information and $\bar{h}^{l}$ given $\bar{h}^{l+1}$ (and its given because we are on level l) is the likelihood information. The function $g_{\mathcal{P}}(\cdot)$ will represent the mapping expressed in equation 18 where the h^l will imitate what the noise process u does, that is, adding extra information. We should note that this model is not really a latent variable model because given all the mapping functions, f and g, for all the layers the reconstruction is always the same. What [Valpola, 2015] does is take the underlying idea of latent variable models and try to create a neural network that implements this idea which is basically the **independent representation capacity**.

For sure a trivial solution that the reconstruction function can learn is just copy h^l to the output. We will now see how can we learn this model to avoid this little problems for example adding noise to the bottom-up path. To end with figure 15 shows a comparison between the three models.

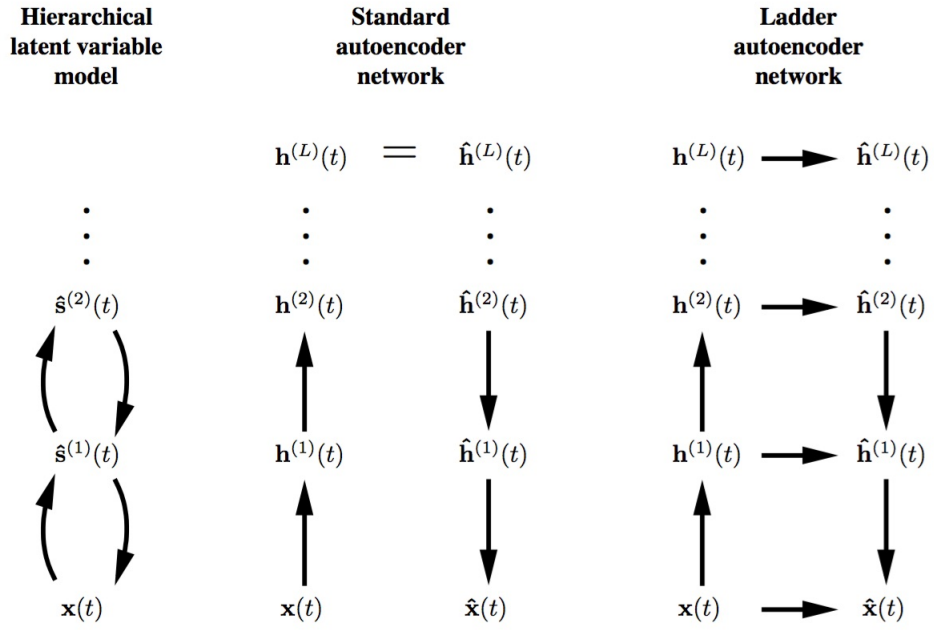


Figure 15: Comparison of hierarchical latent variable model, autoencoder and the ladder autoencoder network. Figure obtained from [Valpola, 2015]

5.2.4 The Learning Scheme

We have seen how can we modify the autoencoder topology to have an autoencoder that implements the underlying idea of a latent variable model. The aim of this subsection is how can we now train this topology to learn good representations. The objective is proposing a learning scheme in which the learning process is distributed rather than guided by propagating errors computed from a single error term.

There are several techniques for learning the parameters in latent variable model for example the EM algorithm [Dempster et al., 1977]. There are other proposed techniques based on reconstruction error. These is quite complicate so for deeper information look at [Valpola, 2015] section 3.1. We can find how to define the cost of reconstructing the input to the network in a latent variable model. What is important is that in latent variable models the cost is make up from independent cost measuring reconstruction errors at each level of the hierarchy.

For the next part we should remember that the LN is not a latent variable model but takes the idea from it. What we find in section 3.1 from [Valpola, 2015] is that the mapping functions depends on a prior distribution of h and the noise distribution $p(u)$ in a latent variable model. It makes sense because in some way a reconstruction depends on a projection with some variability and denoising means learning which kind of perturbation we have, so when projecting we are able to correct it and reconstruct the original signal. We will introduce the term denoised for reconstructed variables, that is, $\bar{h}^l$ and the function $g_{\mathcal{P}}(\cdot)$ would be the denoising function.

How can we do that in neural networks is really easy. We have already talked about denoising autoencoders. A denoising autoencoder is a good way of making the autoencoder learn robust representations of the data. What the autoencoder learn is basically denoising functions. The way we incorporate this to the LN is by simply corrupting the input. We will go a bit deeper in the denoising process and why it is needed when describing the denoising function used in the model.

Once we have the topology created we should imitate the way latent variable models are learnt that is learning to reconstruct at each level of the hierarchy that is each denoising functions is taught to learn to denoise a level. The way we will distribute the learning process is with the next cost function:

$$C = C^0 + \sum_{l=1}^{L} \omega^l * C^l \tag{32}$$

Where ω^l ponderate the different costs. On the other hand:

$$C^l = ||h^l - \bar{h}^l||_2 \tag{33}$$

Finally:

$$\bar{h}^l = g_{\mathcal{P}}(\widetilde{h}^l, \bar{h}^{l+1}) \tag{34}$$

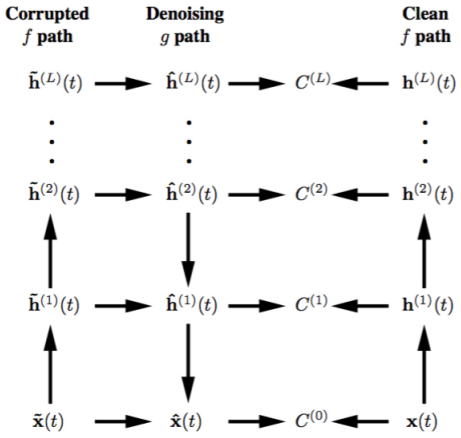


Figure 16: Computed cost in ladder networks. Figure obtained from [Valpola, 2015]

Now we can apply stochastic gradient descent to minimize the single cost C wrt to the parameters. Figure 16 represents a good scheme of this training process. There is one more thing to make this model work which is normalization. We will go over this latter. To end with, note that having a cost made up from cost at different levels is also useful to create very deep architectures.

We have just seen how the model we will use is justified. In the next section we will go deeper in how can we fit this model with supervised learning, how the different operations of the model represented in figure 14 are defined and which parameters will be optimized.

5.3 Supervised Ladder Network

We have seen a model to learn probability distributions $p(x)$. This model learns data representation by a function that reconstructs data from corrupted versions in level l and reconstructed versions in $l+1$. This model minimizes a cost function which is made from the evaluation of the euclidean distance of the denoised variables and the clean variables at each level l.

Due to the fact that this model is based in latent variable model it fits well with supervised data [Valpola, 2015]. The only difference is if the tags are observed or not. Due to the fact this model is based on latent variable models should fit well with semi-supervised learning. We will now see how this is done. The

information is taken from [Rasmus et al., 2015].

The big contribution of semi-supervised learning is the ability of using the features that supervised learning finds suitable to refine which features of the input data, at different levels, are helpfull for the discriminant task performance. We will now see how the supervised part is incorporated in the cost function and the model.

The discriminative cost is given by the cross-entropy of the outputs at level L compared to the target. The pre activation of this layer is the softmax function which ensures a normalized output, that is $t_i \in [0, 1]$ and $\sum_{i=1}^{K} t_i = 1$, that can be interpreted as the posterior probability of the target.

The cost is the cross entropy that can be also written in probabilistic terms as:

$$C_s = -\frac{1}{N} * \sum_{i=1}^{N} \log P(\widehat{T}_i = T_i \,|\widetilde{X}_i) \tag{35}$$

that is the probability of the mismatch between the output and the target given the input. As we see there is noise addition to the input to improve generalization and prevent the reconstruction function to learn the identity (note that $\widehat{T}_i$ is just the output of level L of the unsupervised part). The total cost is minimized using stochastic gradient descent and is the sum of the supervised and unsupervised cost. We will go over the model explaining each part focusing on those we have not seen.

Let's refresh the figure in which the model was exposed:

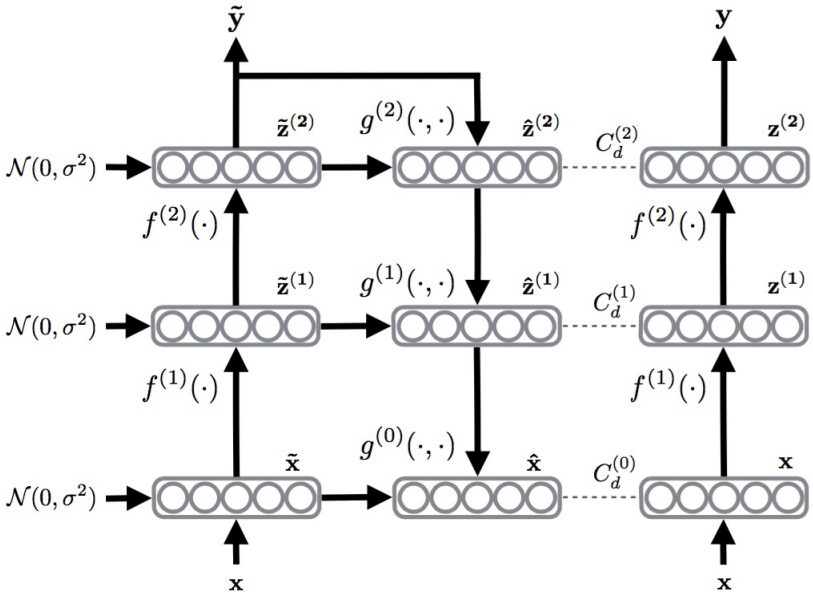


Figure 17: Figure obtained from [Rasmus et al., 2015]

As we see the supervised part takes the noisy output from unsupervised part, as we said. The different cost C_d represents the unsupervised cost for training the ladder network. What we minimize using SGD is $C = C_u + C_s$ where C_u is given by equation 32.

The reason for adding noise at each level is first the denoising principle we explained (it was only necessary to add noise in the input) and avoiding the use of dropout. Adding noise at each level acts in the same way as using dropout. We should remark that the weights from the corrupted encoder and the clean encoder are the same. The clean path is only used for computing the unsupervised cost but note that the weights from this clean path are affected by the noise when optimization takes places and that is the reason why we do not need to use dropout in the clean path.

The last two things to talk about are the use of batch normalization and the definition of the denoising function. The mapping function $f_{\mathcal{P}}(\cdot)$ is given by a linear projection and activation function. First we will go over the denoising functions.

5.3.1 Denoising Function

We have seen how can we implement a neural network based on the principle of latent variable model, that is, let our latent variable add information for the

reconstruction. This is implemented with the lateral connections of the exposed topology in combination with the reconstruction in the above level.

We have also seen that we need to corrupt the input to avoid the denoising function just copy the projected signal at level l. By this corruption we force using the information in the above level to reconstruct and learn perturbation invariant mappings $f_{\mathcal{P}}(\cdot)$. This denoising principle could be view in two ways. We have seen we can learn denoising functions by minimizing the euclidean distance between the original signal and the reconstructed signal. However, it is interesting to remark that this can also be done in a probabilistic way, where the observation is the corrupted variable $\widetilde{h}$ and the latent variable is the objective we want to reconstruct which has a prior probability distribution. We could use the EM that divides the process in two steps: inference (update the posterior probability) and learning (readjust the underlying probability to fit the observations). Denoising source separation (DSS) [Särelä and Valpola, 2005] gives deeper information on how can we do this using the EM. In our LN the inference process is done using the principle of denoising, that is minimizing the mismatch between the variables. Remember we could approach this principle minimizing the reconstruction error or by probabilistic modeling. A denoising function could be written like (note that now it only takes one variable as input):

$$z = g_{\mathcal{P}}(\widetilde{z}) \tag{36}$$

Where the underlying latent variable model is given by:

$$\widetilde{z} = W \cdot z + n \tag{37}$$

Where the observation is the corrupted value and n is a noise process. This denoising function $g_{\mathcal{P}}(\cdot)$ depends on the prior distribution of z and noise process which gives the variance. All this can be used to compute the posterior distribution $p(z|\widetilde{z})$. A good way of learning this model is with denoising autoencoder [Vincent et al., 2008] [Vincent et al., 2010]. This has shown to regularize.

An example of a denoising function is given the purple line in figure 18.

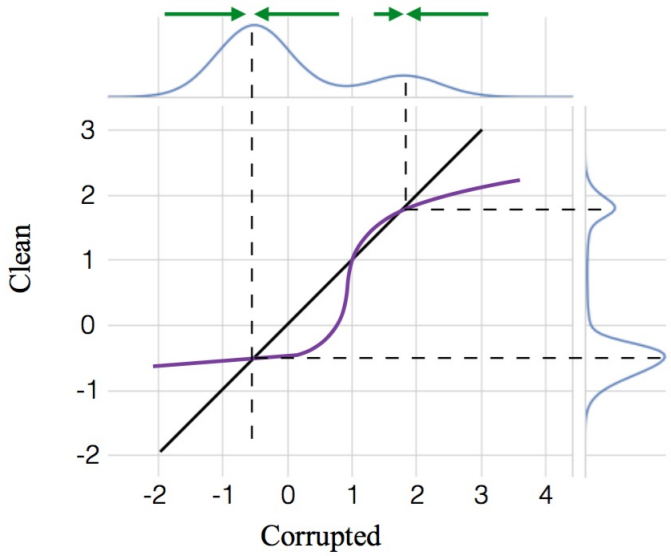


Figure 18: Optimal denoising function for bimodal distribution. Figure obtained from [Rasmus et al., 2015]

We see how the denoising function is able of modify the data distribution to extract the injected noise. We will now see how the denoising functions are defined in this model and the reasons for doing it like this. We should have clear that the denoising principle can be view the same way as a latent variable model where the observations are the corrupted signals $\widetilde{h}$ and the latent variable are h. Our reconstruction function implements both the denoising approach and the stochastic principle of latent variable models where they can incorporate information that let discard information in other levels.

In our model the denoising function is defined as:

$$\bar{h}^l = q_{\mathcal{P}}(\widetilde{h}^l) = (\widetilde{h}^l - \mu) * \nu + \mu \tag{38}$$

And for the fact that is linear wrt to the latent variable the optimal denoising is the gaussian distribution [Valpola, 2015]. It is clear that $\bar{h}^l$ incorporates prior information about h^l through $\widetilde{h}$ and $q_{\mathcal{P}}(\cdot)$ represent the inverse mapping from latent h^l to the observed variable $\widetilde{h}^l$, that is we represent $p(\widetilde{h}|h)$. The noise u process is incorporated in the random perturbation added to the observed variable.

Let refresh our model denoising function given by:

$$\bar{h}^l = g_{\mathcal{P}}(\widetilde{h}^l, \bar{h}^{l+1}) \tag{39}$$

We see that the denoising function mixes the two principles we have been discussing in the above sections. Note that adding noise is also necessary to force the denoising function to use the information in the above layer because if not it will only copy the encoder output. In [Rasmus et al., 2015] there is more information on how this function is parametrized and explanation on other denoising functions. What is important is that the assumption of the latent variable being gaussian is relaxed by making it being gaussian conditional independent, that is:

$$p(z^l|z^{l+1}) = \prod_{i=1}^{K} p(z_i^l|z^{l+1}) \tag{40}$$

In the denoising function from equation 38 the dependence of ν and μ with the latent layer is given by a batch normalized projection followed by a non-linearity with some trainable parameter. This gives the model a freedom to learn a wide variety of denoising functions.

Minimizing the reconstruction error forces the encoding mappings to learn good features that fit with the prior distribution given by the layer at level $l + 1$.

Once we explained all the model we should end up justifying why this fits well with supervised learning and why we should define the denoising functions with non linearity. Note that given the latent variables, if the observed variables are true gaussian independent distributions, our denoising function would end up being linear. In this case it is possible that the variable are not gaussian independent and this means that the distribution of h^l can be modulated by h^{l+1} with a variety of possible mapping functions. This means that supervised learning has an indirect influence on the representations learned by the unsupervised decoder: any abstractions selected by supervised learning will bias the lower levels to find more representations which carry information about the same abstractions [Rasmus et al., 2015].

5.3.2 Batch Normalization

This model also used batch normalization as explained in the state of the art, with the same parameters β and γ. They use batch normalization in all the layers adding β and γ when necessary. For example as we commented ReLu only need the bias parameter β.

Batch normalization serves for two purposes. The first one is reducing the covariate shift [Ioffe and Szegedy, 2015]. The other one is because the input layer need some kind of normalization to prevent the encoder outputs constant values, which are the easiest to denoise. Note that we minimize $||\bar{z} - f(x)||_2$, so if $\bar{z} = f(x) = z =$ constant we found the trivial solution. We will be learning something like:

$$h^{l+1} = \mathcal{A}\{f_{\mathcal{P}}(h^l)\} = \mathcal{A}\{W \cdot h^l + b)\} = \mathcal{A}\{0 \cdot h^l + b)\} \tag{41}$$

However, if we normalized the projection we avoid this problem. Normalizing the output of $f(x)$ prevents this effect because it sets the variance in a layer to 1. A constant value is the same as having a distribution with a mean and a 0 variance. This also implies a normalization in the decoder to have same distributions in the input to the denoising function.

To end with the next image shows how the learning process is done:

Algorithm 1 Calculation of the output $\mathbf{y}$ and cost function C of the Ladder network

Require: $\mathbf{x}(n)$
- # Corrupted encoder and classifier
- $\tilde{\mathbf{h}}^{(0)} \leftarrow \tilde{\mathbf{z}}^{(0)} \leftarrow \mathbf{x}(n) + \texttt{noise}$
- **for** l = 1 **to** L **do**
  - $\tilde{\mathbf{z}}^{(l)} \leftarrow \texttt{batchnorm}(\mathbf{W}^{(l)}\tilde{\mathbf{h}}^{(l-1)}) + \texttt{noise}$
  - $\tilde{\mathbf{h}}^{(l)} \leftarrow \texttt{activation}(\boldsymbol{\gamma}^{(l)} \odot (\tilde{\mathbf{z}}^{(l)} + \boldsymbol{\beta}^{(l)}))$
- **end for**
- $P(\tilde{\mathbf{y}} \mid \mathbf{x}) \leftarrow \tilde{\mathbf{h}}^{(L)}$
- # Clean encoder (for denoising targets)
- $\mathbf{h}^{(0)} \leftarrow \mathbf{z}^{(0)} \leftarrow \mathbf{x}(n)$
- **for** l = 1 **to** L **do**
  - $\mathbf{z}^{(l)}_{\text{pre}} \leftarrow \mathbf{W}^{(l)}\mathbf{h}^{(l-1)}$
  - $\boldsymbol{\mu}^{(l)} \leftarrow \texttt{batchmean}(\mathbf{z}^{(l)}_{\text{pre}})$
  - $\boldsymbol{\sigma}^{(l)} \leftarrow \texttt{batchstd}(\mathbf{z}^{(l)}_{\text{pre}})$
  - $\mathbf{z}^{(l)} \leftarrow \texttt{batchnorm}(\mathbf{z}^{(l)}_{\text{pre}})$
  - $\mathbf{h}^{(l)} \leftarrow \texttt{activation}(\boldsymbol{\gamma}^{(l)} \odot (\mathbf{z}^{(l)} + \boldsymbol{\beta}^{(l)}))$
- **end for**
- # Final classification:
- $P(\mathbf{y} \mid \mathbf{x}) \leftarrow \mathbf{h}^{(L)}$
- # Decoder and denoising
- **for** l = L **to** 0 **do**
  - **if** l = L **then**
    - $\mathbf{u}^{(L)} \leftarrow \texttt{batchnorm}(\tilde{\mathbf{h}}^{(L)})$
  - **else**
    - $\mathbf{u}^{(l)} \leftarrow \texttt{batchnorm}(\mathbf{V}^{(l+1)}\hat{\mathbf{z}}^{(l+1)})$
  - **end if**
  - $\forall i : \hat{z}^{(l)}_i \leftarrow g(\tilde{z}^{(l)}_i, u^{(l)}_i)$ # Eq. (2)
  - $\forall i : \hat{z}^{(l)}_{i,\text{BN}} \leftarrow \frac{\hat{z}^{(l)}_i - \mu^{(l)}_i}{\sigma^{(l)}_i}$
- **end for**
- # Cost function C for training:
- $\text{C} \leftarrow 0$
- **if** $t(n)$ **then**
  - $\text{C} \leftarrow -\log P(\tilde{\mathbf{y}} = t(n) \mid \mathbf{x}(n))$
- **end if**
- $\text{C} \leftarrow \text{C} + \sum_{l=0}^{L} \lambda_l \left\| \mathbf{z}^{(l)} - \hat{\mathbf{z}}^{(l)}_{\text{BN}} \right\|^2$ # Eq. (3)

Figure 19: Ladder Network Algorithm. Figure obtained from [Rasmus et al., 2015]

5.3.3 Extension to convolutional networks

Extending this model to convolution networks is nothing more than implement the deconvolution in the decoder. The denoising function is also shared in each layer.

Finally, as the encoder have some pooling layer the decoder should implement an upsampling. This downsampling is compensated by copying on the decoder side.

5.3.4 Models Hyperparemeters

This are the different parameters used by the authors in their experiments. We will expose those from the three models we are exploring: MNIST convolutional, MNIST fully connected and CIFAR10 convolutional. The different hyperparameters are: network topology, activation functions, parameters of the gaussian noise, the weights from the unsupervised cost, type of denoising function, batch size, learning rate decay, adding γ parameter to softmax, number of

epochs, whitening with ZCA. All the activation functions have the β parameter from the BN added except the linear activation function.

5.3.4.1 MNIST Fully Connected

- network topology: encoder $\rightarrow$ 1000-500-250-250-250-10
- activation function: rectifier linear unit
- gaussian noise distribution: $\mathcal{N}(0, 0.3)$
- denoising function: gaussian
- batch size: 100
- number of epochs: 150
- learning rate: 0.002
- learning rate decay: linear wrt the number of epochs starting from epoch 100.
- unsupervised weight costs, ω_l: 1000 1 0.01 0.01 0.01 0.01 0.01
- γ in softmax: yes
- whitening with ZCA: no

5.3.4.2 MNIST Convolutional

- network topology: encoder $\rightarrow$ See figure 20
- activation function: rectifier linear unit
- gaussian noise distribution: $\mathcal{N}(0, 0.3)$
- denoising function: gaussian
- batch size: 100
- number of epochs: 150
- learning rate: 0.002
- learning rate decay: linear wrt the number of epochs starting from epoch 100.
- unsupervised weight costs, ω_l: 0 0 0 0 0 0 0 0 0 1
- γ in softmax: yes
- whitening with ZCA: no

5.3.4.3 CIFAR10 Convolutional

- network topology: encoder → See figure 20
- activation function: leaky rectifier linear unit
- gaussian noise distribution: $\mathcal{N}(0, 0.3)$
- denoising function: gaussian
- batch size: 100
- number of epochs: 70
- learning rate: 0.002
- learning rate decay: linear wrt the number of epochs starting from epoch 60.
- unsupervised weight costs, ω_l: 0 0 0 0 0 0 0 0 0 0 0 0 4
- γ in softmax: no
- whitening with ZCA: yes

Model

ConvPool-CNN-C	Conv-Large (for CIFAR-10)	Conv-Small (for MNIST)
Input 32 × 32 or 28 × 28 RGB or monochrome image		
3 × 3 conv. 96 ReLU	3 × 3 conv. 96 BN LeakyReLU	5 × 5 conv. 32 ReLU
3 × 3 conv. 96 ReLU	3 × 3 conv. 96 BN LeakyReLU	
3 × 3 conv. 96 ReLU	3 × 3 conv. 96 BN LeakyReLU	
3 × 3 max-pooling stride 2	2 × 2 max-pooling stride 2 BN	2 × 2 max-pooling stride 2 BN
3 × 3 conv. 192 ReLU	3 × 3 conv. 192 BN LeakyReLU	3 × 3 conv. 64 BN ReLU
3 × 3 conv. 192 ReLU	3 × 3 conv. 192 BN LeakyReLU	3 × 3 conv. 64 BN ReLU
3 × 3 conv. 192 ReLU	3 × 3 conv. 192 BN LeakyReLU	
3 × 3 max-pooling stride 2	2 × 2 max-pooling stride 2 BN	2 × 2 max-pooling stride 2 BN
3 × 3 conv. 192 ReLU	3 × 3 conv. 192 BN LeakyReLU	3 × 3 conv. 128 BN ReLU
1 × 1 conv. 192 ReLU	1 × 1 conv. 192 BN LeakyReLU	
1 × 1 conv. 10 ReLU	1 × 1 conv. 10 BN LeakyReLU	1 × 1 conv. 10 BN ReLU
global meanpool	global meanpool BN	global meanpool BN
		fully connected 10 BN
10-way softmax		

Figure 20: Convolution topology. Figure obtained from [Rasmus et al., 2015]. From left to right we find the convolution topology in which the models are based [Springenberg et al., 2014], the CIFAR10 convolution network and the MNIST conbvolution network.

5.3.5 Results of the model

We now present the results reported by [Rasmus et al., 2015]. These results include not only those achieved by the ladder network but also some achieved by other researchers. Basically we have included the result from adversarial [Goodfellow et al., 2014] which is a supervised technique and DBM + dropout finetunning which is a fine tunning unsupervised technique [Srivastava et al., 2014]. The results reported are the mean of ten experiments with different parameter initialization. The first table shows MNIST fully connected (FC) results. We see the results achieved depend on the number of labeled samples. We see that this model reaches state of the art result in all the tasks exposed under the same conditions. It is also interesting to see that we do not need an unsupervised cost made from all the levels as we see in the results reported in the Γ-model and the only bottom-level cost from table 1. This means that reconstructing at each level is not so important (it was for latent variable model) but the main power of these models are in the $g_{\mathcal{P}}(\cdot)$ function which implements the denoising process and simulates the behavior of latent variable models.

FC MNIST			
# of labels	100	1000	All labels
DBM, Dropout [Srivastava et al., 2014]			0.79
Adversarial [Goodfellow et al., 2014]			0.78
Γ-model (Ladder with only top-level cost) [Rasmus et al., 2015]	3.06%	1.53%	0.78%
Ladder, only bottom-level cost [Rasmus et al., 2015]	1.09%	0.9%	0.59%
Ladder full [Rasmus et al., 2015]	1.06%	0.84%	**0.57%**

Table 1: Results for fully connected MNIST task. In red is state of the art result.

The next table shows the result with MNIST convolutional. We should remark that this results are really good considering we are only using 100 labels.

Convolutional MNIST			
# of labels	100	1000	All labels
EmbedCNN [Weston et al., 2012]	7.75%		
SWWAE [Zhao et al., 2015]	9.17%		0.71%
Baseline: Conv-Small, supervised only [Rasmus et al., 2015]	6.43%		0.36%
Conv-FC [Rasmus et al., 2015]	0.99%		
Conv-Small, Γ-model [Rasmus et al., 2015]	0.89%		
BN Maxout Network in Network [Chang and Chen, 2015]			0.24%
Drop-Connect [Wan et al., 2013]			**0.21%**

Table 2: Results for Convolutional MNIST task. In red is state of the art result.

Finally, we show the results reported for CIFAR10.

Convolutional CIFAR10		
# of labels	4000	All labels
All-Convolutional ConvPool-CNN-C [Springenberg et al., 2014]		9.31%
Spike-and-Slab Sparse Coding [Goodfellow et al., 2012]	31.9%	
Baseline: Conv-Small, supervised only [Rasmus et al., 2015]	23.33%	9.27%
Conv-Small, Γ-model [Rasmus et al., 2015]	20.40%	
BN Maxout Network in Network [Chang and Chen, 2015]		6.75%
Fractional Max Pooling [Graham, 2014]		**3.47%**

Table 3: Convolutional CIFAR10 results. In red is state of the art.

5.4 Adversarial Noise

The last thing to explain is adversarial noise. [Szegedy et al., 2014b] showed that the neural networks are sensible to adversarial examples.

Adversarial examples are perturbations that can be computed like

$$\widetilde{X} = X + \tau * \text{sgn}\{\frac{\partial C}{\partial x}\Big|_X\} \tag{42}$$

in neural networks. This noise modifies widely the output of a model. They simulate, through the parameter τ, perturbations like the precision error in 8-bit quantification that should not modify the output of a model. The particularities of the noise such as using the $\text{sgn}\{\cdot\}$ function or the $||\tau||$ can be find in [Goodfellow et al., 2014].

It is clear that the magnitude of the derivative of the function could be such as little as the precision error but bigger enough to increase the cost function which implies increasing the minimum classification error. Note that the added noise is added directly in the direction in which the cost function increases and this means that the cost would be increased. Let's put an example. For making the calculations easier to write, derive and understand we will use a very simple regression model with only one input data X. This model would be the same as the one exposed in appendix 2 given by figure 21, that is a 0 hidden layer with linear activation function that implements $f : \mathbb{R} \to \mathbb{R}$.

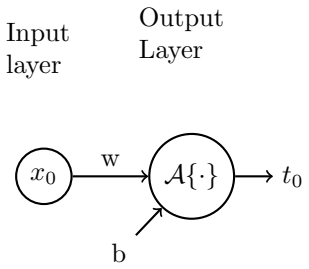


Figure 21: Model: $t = w * x + b$

We will use the sum of squared errors as cost function. For this case this error function can be written like:

$$C = \frac{1}{2} * \sum_{i=1}^{N} ||\widehat{T}_i - T_i||^2 = \frac{1}{2} * (\widehat{T} - T)^2, t \in \mathbb{R} \tag{43}$$

The derivative of the cost function respect to the input is easy to compute (note that $t = w * x + b$ and we change from upper case to lower case):

$$\frac{\partial C}{\partial x} = -1 * (\widehat{t} - t) * w = w^2 * x - w * \widehat{t} \tag{44}$$

We now corrupt the input with adversarial noise, that is:

$$\widetilde{x} = x + u = x + w^2 * x - w * \widehat{t} \tag{45}$$

And now the prediction of the model changes to:

$$t = w * \widetilde{x} = w * x + b + w^3 * x - w^2 * \widehat{t} \tag{46}$$

Let's check now the error function. We denote with subindex b to the error before adding adversarial noise and with subindex a to the error after. We scale both errors by 2 with, for sure, does not change the result.

$$C_b = (\widehat{t} - w * x - b)^2 \tag{47}$$

$$C_a = (\widehat{t} - w * x - b - w^3 * x + w^2 * \widehat{t})^2 \tag{48}$$

We should note several things. The first thing is to particularize these expressions to our sample X, that is changing lower case to upper case. When doing this X and $\widehat{T}$ are constants. We have also a fix value for w and b. We can see that $C_a > C_b$ as long as $\widetilde{T} \neq W * X$ or $w = 0$. To ensure this we have to ensure that:

$$(\widehat{t} - w * x - b - w^3 * x + w^2 * \widehat{t})^2 > (\widehat{t} - w * x - b)^2 \tag{49}$$

that due to the square value and that X, $\widehat{T}$, b and w are constants is the same as verifying that :

$$|\widehat{t} - w * x - b - w^3 * x + w^2 * \widehat{t}| \overset{?}{>} |\widehat{t} - w * x - b| \tag{50}$$

Note that we could rewrite this expression as:

$$|a + b| \overset{?}{>} |a| \tag{51}$$

To demonstrate this inequality we could think in using the triangle inequality given by 52 and the fact that $|a| \leq |a| + |b|$. However, this only holds when the sign of both operands is the same and this is something that cannot be

ensure because this problem is unconstrained and the derivative is a polynomial function which is also unconstrained.

$$|a+b| \leq |a| + |b| \tag{52}$$

We could try to check how the sign of this function is governed. Rewriting our expression like:

$$\widehat{t} - w * x - b - w^3 * x + w^2 * \widehat{t} = \widehat{t} - w * x - b + w^2 * (\widehat{t} - w * x) \tag{53}$$

And now it is clear that the sign of the expression is governed by $\widehat{t} - w * x$ and biased by b. Now our inequality changes to:

$$|\widehat{t} - w * x - b + w^2 * (\widehat{t} - w * x)| > |\widehat{t} - w * x - b| \tag{54}$$

which is true for the fact that both elements of the function have the same sign (w^2 is always positive) and both terms are biased the same. Again is a monotic transformation and this does not change the result. We now apply the triangle inequality and demonstrate what we wanted. This is expressed in equation 55.

$$|a+b| = |a| + |b| > |a| \tag{55}$$

Note that this is always true except for the two particularities $\widetilde{T} = W * X$ or $w = 0$. To end with Let's take a look at the two particularities. $w = 0$ does not seem interesting because that means the cost is the same no matter what the value from X is. The other value is exactly the value we want our model to predict. This means that if our model predicts the exact value we cannot add any more information. However, this is something tricky and we will see why.

The first thing we should note is that we minimize the cost wrt to the parameters and not wrt input so maybe we could have a 0 value adversarial noise but we are not in a singularity from the cost we minimize. We should check if with other cost functions this same thing happens. Another important characteristic is that we are only considering one sample from the distribution. Think that the only way this effect shown could happen is by having a population make from the same sample because our model will only predict exactly one of the samples of the minibatch.

The objective of this was showing that the cost is really increased. However, the objective of adding adversarial noise is improving generalization. Generalization is something really hard to formalize for the fact that the new data is not present in our training process and that it is not easy to show how adding adversarial noise affects the cost function respect to the parameters. In this example w and b were always fixed. Adding adversarial noise only guarantees that the cost is higher than before but we should check which direction of the cost function is being affected and how is affected. This is represented in figure 22. It should be clear that as long as the dimensions are increased the possibilities grows highly.

In [Goodfellow et al., 2014] there are some hypotheses for explaining why they generalize well.

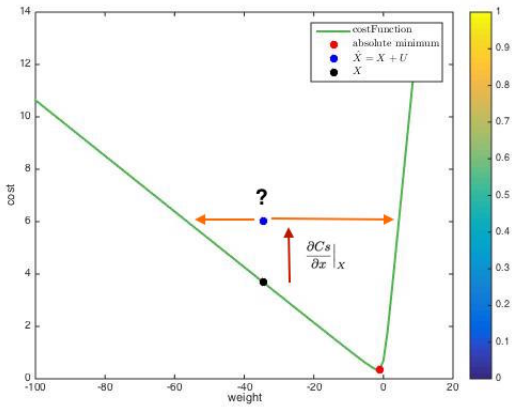


Figure 22: Example of how adversarial noise influence the cost function wrt to the parameter space.

To end with we will show an example of corruption with adversarial noise. The next figures show how the image is perturbed with adversarial noise and an uncorrelated noise drawn from an isotropic gaussian distribution $q = \mathcal{N}(0, I)$. The noise is computed as $u = \tau * \text{sign}(q)$ to have the same power as the adversarial noise.

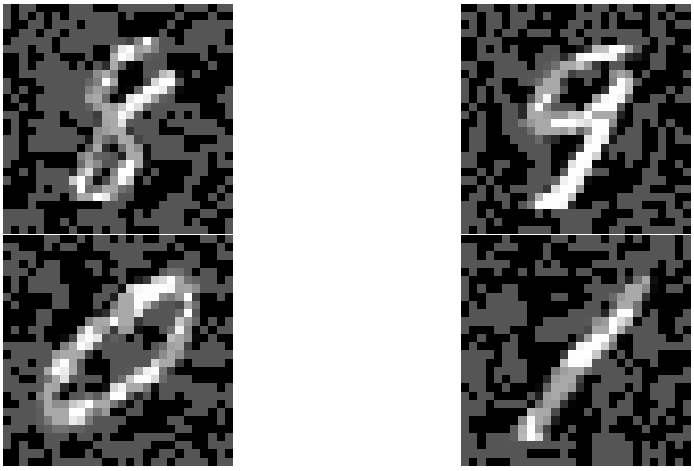


Figure 23: Adversarial noise addition to example from MNIST database with $\tau = 0.25$

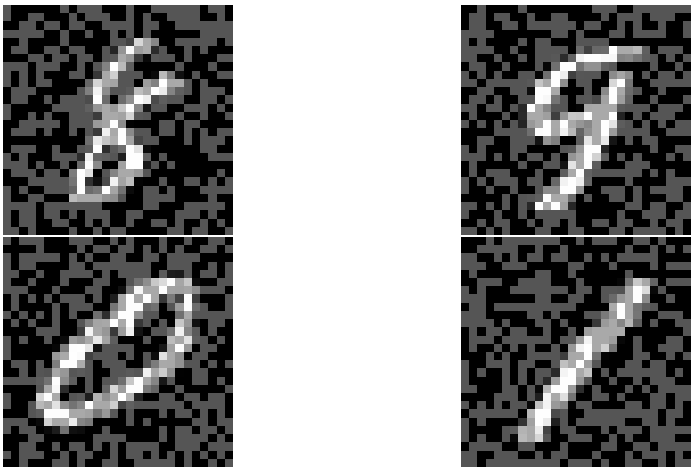


Figure 24: Gaussian noise addition to example from MNIST database with $\tau = 0.25$

As we could see there is not a big difference between the corrupted images with one noise and with the other. However, if we evaluate the system with adversarial noise and the gaussian distributed noise we find a 6.87% of error for the adversarial noise addition, 0.61% error for gaussian distributed noise addition and 0.51% with the original test set.

These results are very interesting because we can visually think that the added noise corrupts in the same way the images. Think the effort you have to make to recognize the different numbers in the image. However, as we can see the addition of adversarial noise implies a higher error rate. This is what an adversarial example is, a data with a corruption lower than a quantification error but that causes big changes in the output of the model.

[Goodfellow et al., 2014] says that with a value of $\tau = 0.25$ they cause a shallow softmax classifier to have an error rate of 99.9 and that is the reason for choosing this value when performing our test. Note that in the MNIST database where the images are basically binary, adding a value of 0.25 in all the pixels would not change the result if we apply a threshold at 0.5. However, if that noise is in the gradient direction we increase the error.

What we have seen is that the supervised ladder network is not robust to adversarial noise so our work would be incorporate this class of noise to try to reach better results than the ones reported in [Rasmus et al., 2015].

6 Experiments and Results

In this chapter we will explain the experimental procedure followed with the intermediate results. As we have said, it is not easy to show why adversarial noise improve neural network generalization. This means that the experiments done are in someway heuristic, and as we will see there is no exact reason for using a magnitude order for the adversarial power.

On the other hand, due to the high computation cost of our task (we will see later an example), we were not able to do an exhaustive search of the new hyperparameters in our problem. Moreover, we could not perform a bunch of experiments for each hyperparameter searched. Remember we said that [Rasmus et al., 2015] performed ten different experiments with different SGD parameter initialization in their experiments and the given result was the mean of the ten experiments. When searching hyperparameters we compared the results in [Rasmus et al., 2015] to ours with only one seed. This seed fix the parameter initialization. When we thought we had a good model, we performed ten experiments and do the average. We had to do it in this way for two reasons. First, as we said is the computation cost. What the authors do is for each hyperparameter do ten experiments with a validation set and do the mean. The second is because using a validation set for hyperparameter search and comparing with the result of the test could give unreal estimations (we will have also to do ten experiments with the baseline model and a validation set for each hyperparameter. Ten experiments of the CIFAR10 database, for example, took a whole week to finish). Note that checking the hyperparameter with only one network and use the best to perform 9 more experiments is in some way the same as doing an unbiased test, because we do not check the influence of that hyperparameter in other experiments so in the end we are doing the same as using a validation set, that is, choosing a hyperparameter without knowing how will work in different model. As we will see adversarial noise give worse results in some networks. In addition, we did all the experiments for the fully connected 60000 labeled MNIST task. The best result for this was applied to the other task. This is done in this way firt because of the computation cost and then because we try to avoid as much as we can an heuristic method. This means that if in the MNIST task we add adversarial noise computed with a sign function, that should be also done with CIFAR. The value of the τ parameter can change because the cost functions are dependent on the data set.

We shall comment several curious things about the baseline result (we will refer to [Rasmus et al., 2015] results as this). When trying to improve a result of 0.59% we have to take care of each little detail because it is quite hard to do better than this, even more when we have lack of computation resources for the task and when discriminating these errors is also difficult for the human eye (remember the photo of Cifar database it was far for being a well defined horse). Any changes in the hardware or software computing platform can also modify

the result. This was the reason for using the same GPU for hyperparameter search and use any of them for a specific experiment, as we will see later. This also helps demonstrate that our technique is unbiased. We will see how the same experiment in one computer give 0.58% and the same in other give 0.64%.

The experimental procedure is done in the next way. First we perform 10 experiments with the baseline model, with seeds from one to ten (in the work it is not said which seeds are used). These results are shown in table 4. The state of the art result will be shown in red color.

Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result %	0,53	0,55	0,54	0,62	0,56	0,53	0,58	0,56	0,61	0,57	**0,565**

Table 4: Baseline Experiment

The first thing we can see is that the average result is different from what [Rasmus et al., 2015] exposed. The reason could be the GPU used, the blocks version, the operative system... In blue we show the result to what we will compare our searching. For example using the same blocks version as the authors we obtained for the first seed a result of 0,51%. We use a different blocks version and we notice here that the blocks version influence the result. In a CPU this result was 0.54% (we think in using the CPU of another computer to perform the average results and keep the searching experiments in the GPU). We see how the software or hardware computing platform can influence the results and when trying to keep such a lower result down we can not permit having one more error due to these facts. We ended doing everything in the commented blocks version and with only one GPU.

6.1 Supervised Gaussian Noise Addition

Due to the fact that adversarial noise is supposed to affect the supervised task of the semi-supervised model we first compute the result of the proposed model without the unsupervised part and with the addition and suppression of the gaussian noise in each layer. The results are showed in the next table:

	Noise: every layer $\mathcal{N}(0, 0.3)$	**Without Noise**
Result %	0.82	1.10

Table 5: Baseline Experiment with only supervised learning

As we see, adding noise improves generalization. We can see that the addition of the unsupervised learning decreases in 29 the errors of the network. With these results we continue with the search of adversarial noise.

6.2 In Search of Adversarial Noise

This section is dedicated to search for adversarial noise. We think in performing several experiments. [Goodfellow et al., 2014] uses the sgn$\{\cdot\}$ function for training. We explore also not to use this function and we call this using directly the gradient (UDG). We also try to see if we should only add adversarial noise or we should add also the gaussian noise. Finally, we ended in trying to normalize the UGD to have unity norm and having the same power as the sgn$\{\cdot\}$ computed noise.

All this experiments where done using different τ factors. Because we did not have enough computation we decided to explore different orders of magnitude instead of different values. For example we train using a set factors of the form $\tau = \{0.25, 0.025, 0.0025, 0.00025\}$.

Before showing the different experiments it is necessary to show clearly how the different adversarial noises, u, where computed. Corrupting a sample X with u_{SGN} adversarial noise, [Goodfellow et al., 2014] is defined as:

$$\widetilde{X} = X + \text{sgn}\{\frac{\partial Cs}{\partial x}\Big|_X\} \tag{56}$$

Corrupting a sample with u_{UDG} noise is defined as:

$$\widetilde{X} = X + \frac{\partial Cs}{\partial x}\Big|_X \tag{57}$$

Corrupting a sample with u_{1N}, that is, making the noise having unity norm. Due to the fact that the derivatives are vectors what we did is the next thing. Let U be $\frac{\partial Cs}{\partial x}\Big|_X$.

$$\widetilde{X} = X + \frac{U}{||U||} \tag{58}$$

Where $||U|| = \sqrt{\sum_{i=1}^{K} U^2(i)}$. The last type of noise u_{qN}, that is, controlling the norm of the power is defined as:

$$\widetilde{X} = X + \sqrt{k} * \frac{U}{||U||}, X \in \mathbb{R}^k \tag{59}$$

where q is the new norm which has been fixed to $\sqrt{k}$ to have the same power as the u_{SGN}. All this noise can have an uncorrelated noise n also added. So a corrupted sample is expressed in terms of $\widetilde{X} = X + u + n$.

Once the different noise are defined, the next tables shows the results. We perform two main experiments. The first one with only adversarial noise, that is $\widetilde{X} = X + u$, and the other with both noises, that is $\widetilde{X} = X + u + n$. For each

experiment we corrupted a sample using a factor, τ. Due to the high computation cost we explore this factor in different magnitude orders. The value started from 0.25.

Only Adversarial				
τ	**0.25**	**0.025**	**0.0025**	**0.00025**
u_{UDG}	1.05	1.15	1.12	1.08
u_{SGN}	0.91	0.77	0.91	1.12
u_{1N}	0.86	1.03	1.16	1.17
u_{28N}	1.45	0.85	0.79	0.86

Table 6: Supervised training with adversarial noise. Results from the different models

Adversarial and Gaussian $\mathcal{N}(0, 0.3)$				
τ	**0.25**	**0.025**	**0.0025**	**0.00025**
u_{SGN}	1.09	0.85	0.85	0.76
u_{1N}	0.82	0.77	0.78	0.85
u_{28N}	1.45	0.76	0.78	0.77

Table 7: Supervised training with adversarial and gaussian noise. Results from the different models

The first thing we see is that using adversarial noise gives better results. We also see how the u_{UDG} is not useful and it is logical because this kind of noise changes its power when the cost function changes. Moreover, it could be very low when we are ending the training process. On the other hand we see that using the normalized norm and the u_{SGN} give both good results. In the next experiments we notice some interesting properties in the use of adversarial noise. Our next experiment was using the best supervised configuration we found and incorporate the unsupervised learning to see if we could outperform the results reported by [Rasmus et al., 2015]. From the two options we decided to use adversarial and gaussian noise in the input for the fact that in general it gave better results.

The next tables shows the experiment with unsupervised learning. In this case we perform an experiment using different magnitud orders. For the best one we choose other factor values. With the norm we tried factors starting from 0.25. For the sign we started directly from 0.00025 and search for other values. We did that because of what we observed in the previous results. Remark that the unsupervised data only have gaussian noise addition.

Sign Adversarial Noise											
τ	**0.00015**	**0.00025**	**0.00035**	**0.00045**	**0.00015**	**0.00025**	**0.00035**	**0.00045**	**0.00045**	**0.00045**	**0.00045**
Epochs	150	150	150	150	170	170	170	170	170	200	220
Learning Rate Decay	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.58	0.5	0.45
u_{SGN}	0.63	0.53	0.62	**0.49**	0.61	0.58	0.64	0.54	0.52	0.62	0.55

Table 8: Semi-supervised learning with gaussian and adversarial noise computed with the sign function.

Norm Adversarial Noise							
τ	**0.00015**	**0.00025**	**0.00035**	**0.00045**	**0.00025**	**0.0025**	**0.025**
Gauss Noise std	0.25	0.25	0.25	0.25	0.3	0.3	0.3
u_{28N}	0.59	0.56	0.61	0.61	0.54	0.59	0.61

Table 9: Semi-supervised learning with gaussian and adversarial noise normalizating the result.

We started using the u_{28N} because with this τ values the magnitude order of the factor was not modified (for example $0.025 \cdot 28 = 0.7$ which is a magnitude order explored when using 0.25). We also check other values of gaussian noise std to see what happened but the result were not better. We tried to modify the number of epochs and the learning rate annealing but we did not get better results. We see that the better result was the 0.49 and this was the model chose. We should remark that due to the fact that the first baseline result was obtained with and old blocks versions the baseline for this task was 0.51 and not 0.53 so we choose this as it was the only result lower from the baseline. In the next experiment we notice that the hardware and software platform influence.

Our next experiment was training 10 models with this architecture and see what happens. The results are presented in the next table. As we see the result are only little better so we could continue checking new experiments and try to outperform new tasks. Table 10 shows the results. As we see the seed one give a result of 0.50 vs the 0.49 we obtained when using the other GPU. From this moment we restrict all of ours experiments to the same computing platform and to the same blocks version.

Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	0.50	0.52	0.59	0.53	0.55	0.55	0.56	0.64	0.59	0.54	**0.557**

Table 10: This table shows the result for the MNIST problem with the addition of adversarial noise.

6.2.1 Adding Noise to Unsupervised Data

The last thing to explore is the addition of adversarial noise to the unsupervised data. We did not talk about this when explaining adversarial noise because this is something we propose in this work. Remember adversarial noise, by definition, was a technique for supervised learning. We now need a tag for each

sample which we do not have.

What we do is use the argmax of the output of the softmax for each unsupervised sample and compute the adversarial noise with this virtual tag. Note that as long as the network is well-trained this output will tend to be the real true value.

For understanding why this could be useful we should think that, although the adversarial noise is focus in obtaining better performance when mininizing the cost function what it really does is modifying the data space. It should move the data towards the decision threshold. For the purpose of justifying our next step we will have a problem given by the next figure:

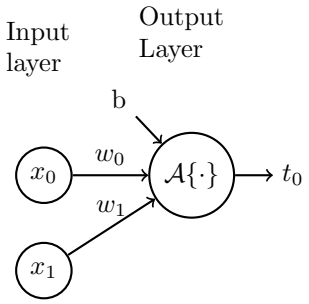


Figure 25: 2D input data space with linear activation function

This represent a regression problem with linear activation function (this justifies why only using a scalar bias). The weights and bias have fixed values ($w_0 = -20$, $b = -40$ and $w_1 = -40$). The error function is minimum squared error and the associated outputs are 1 for one class and 0 for the other. The decision threshold is set to 0.5. This means we train the model to assign one class to tag 1 and the other to tag 0 and perform a classification with a 0.5 threshold. Note that this will make more sense using sigmoid activation function but the aim of this problem is not imitating a realistic problem but showing the influence of the adversarial noise. The data is drawn from to 2-D gaussian distribution.

The next figure shows the data representation space and the decision threshold, that is, all the points where the model have a 0.5 value.

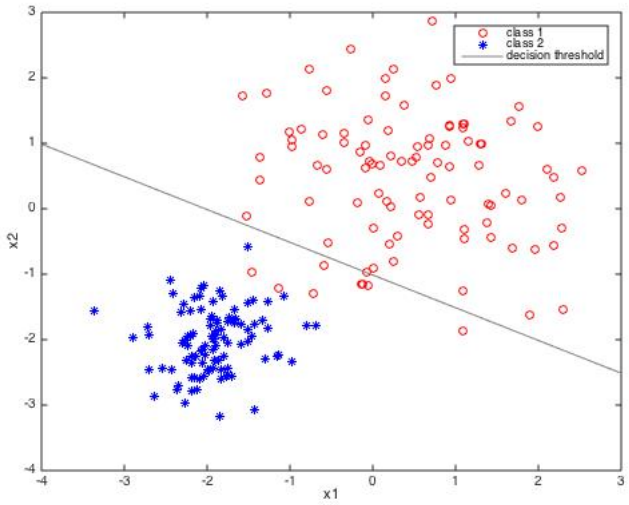
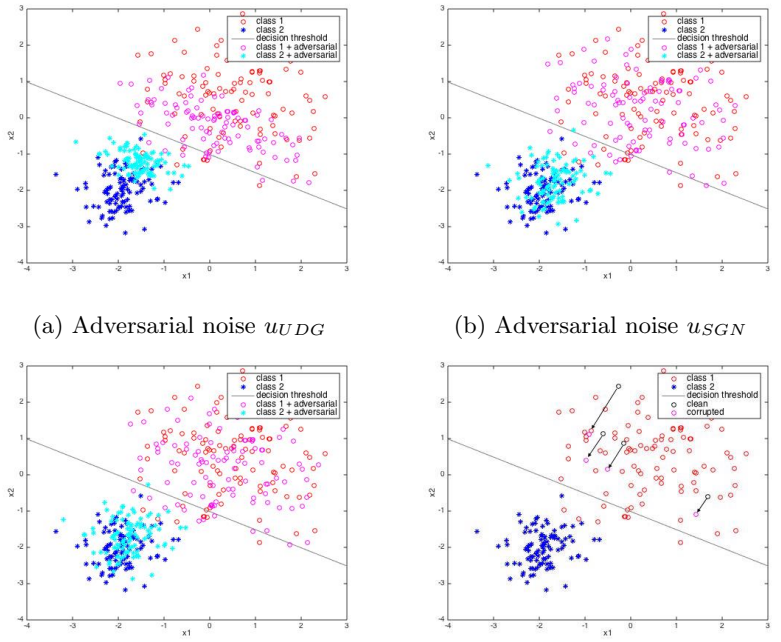


Figure 26: Feature space and decision threshold

The next figure shows what happens when we corrupt the samples with the three kinds of noise we have exposed. The adversarial factor $\tau = 0.00025$ for u_{UDG} and $\tau = 0.25$ for u_{1N} and u_{SGN}. Figure 27d shows how the noise corrupts separate samples. This corruption is represented with arrows.



(a) Adversarial noise u_{UDG}

(b) Adversarial noise u_{SGN}

(c) Adversarial noise $u_{\sqrt{2}*N}$

(d) Corrupting few samples with u_{UDG}

Figure 27: Adversarial data space influence

We can observe several interesting things. We see in u_{UDG} how the size of the corruption depends on how far is the sample from the target (which is what the cost measures). This is the reason for normalizing this kind of noise. When we are close to an optimum this noise does not affect. We showed this in the above results. The u_{SGN} influences the same as the u_{1N}. For the fact that both noises have similar power the only difference is in the direction of the perturbation as we can see in the figure. This means that the use of the sign or not seems to be for computation reasons but we should get similar performance using both noises. However, we decided to try only the normalized noise. We thought using the normalized noise was the proper way to proceed for the fact that unsupervised learning tries to reconstruct so it seems useful to corrupt in the worst direction towards the other data class.

Our next step is corrupting unsupervised path with adversarial noise computed from the argmax of the softmax. It should be clear that although we do not have the tag from a sample it does not matter in which way we corrupt that sample because that is the same of corrupting with gaussian noise, which is a corruption in a random direction.

Unsupervised Data Adversarial Norm									
Power	$4.5 * 10^{-9}$	$4.5 * 10^{-8}$	$4.5 * 10^{-7}$	$4.5 * 10^{-6}$	$4.5 * 10^{-5}$	$4.5 * 10^{-4}$	$4.5 * 10^{-3}$	$4.5 * 10^{-2}$	$4.5 * 10^{-1}$
Result	0.57	**0.48**	0.53	0.60	0.59	0.59	0.57	0.58	0.59

Table 11: Unsupervised adversarial noise addition

With the best value $\tau = 4.5 \cdot 10^{-8}$ we have 2 fewer errors. We now perform a global experiment using 10 seeds to see if we get better results than before. These results are presented in the next table.

Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	0.48	0.59	0.59	0.58	0.60	0.63	0.56	0.59	0.50	0.54	**0:560**

Table 12: This table shows the result for the MNIST problem with the addition of adversarial noise to labeled and unlabeled data.

We see that the results are not better than the baseline ones. However, we see that some models such as seed 1,8,9 or 10 get better results than the experiment without using unsupervised adversarial noise addition. For this reason and for what we explained before about why adding adversarial noise to unsupervised data should be useful we decided to include it in our other experiments.

We will now present the results for the rest of the experiments. We will use the same architecture for everyone: adding sign noise to supervised data and norm noise to unsupervised data. We check for the noise power hyperparameter for each new experiment. We should remark that the noise power for 100 and 1000 FC MNIST label experiment should be as closed to the all labels experiments. This would mean that our experiments are far from being heuristic results. We see that they are not exactly the same and we will after discuss the possible reason. We show the hyperparameter search in the next tables. We present in bold the best hyperparameter. For the MNIST1000 label we tried the best one and the one which was similar to the fully labeled model because it was closed to the best result and we wanted to see if their was a relation between the powers of the noise as we explained.

Supervised Adversarial Factor	**Unsupervised Adversarial Factor**	**Result**
$4.5 * 10^{-5}$	$4.5 * 10^{-9}$	1.06
	$4.5 * 10^{-7}$	1.06
	$4.5 * 10^{-6}$	1.06
	$4.5 * 10^{-5}$	1.03
	$4.5 * 10^{-4}$	1.02
	$4.5 * 10^{-3}$	0.99
	$4.5 * 10^{-1}$	1.01
$\mathbf{4.5 * 10^{-4}}$	$4.5 * 10^{-9}$	0.97
	$\mathbf{4.5 * 10^{-7}}$	**0.94**
	$4.5 * 10^{-6}$	0.99
	$4.5 * 10^{-5}$	0.97
	$4.5 * 10^{-4}$	0.95
	$4.5 * 10^{-3}$	0.95
	$4.5 * 10^{-1}$	1.23
$4.5 * 10^{-3}$	$4.5 * 10^{-9}$	0.99
	$4.5 * 10^{-7}$	0.98
	$4.5 * 10^{-6}$	0.99
	$4.5 * 10^{-5}$	1.08
	$4.5 * 10^{-4}$	0.97
	$4.5 * 10^{-3}$	0.97
	$4.5 * 10^{-1}$	1.15
$\mathbf{4.5 * 10^{-2}}$	$4.5 * 10^{-9}$	1.03
	$4.5 * 10^{-7}$	0.98
	$4.5 * 10^{-6}$	0.97
	$4.5 * 10^{-5}$	1.08
	$\mathbf{4.5 * 10^{-4}}$	**0.91**
	$4.5 * 10^{-3}$	1.10
	$4.5 * 10^{-1}$	1.04

Table 13: MNIST 100 label Fully Connected hyperparameter search. In bold is the chosen hyperparameter

Supervised Adversarial Factor	**Unsupervised Adversarial Factor**	**Result**
$\mathbf{4.5 * 10^{-5}}$	$4.5 * 10^{-9}$	0.95
	$4.5 * 10^{-7}$	0.85
	$4.5 * 10^{-6}$	0.96
	$\mathbf{4.5 * 10^{-5}}$	**0.83**
	$4.5 * 10^{-4}$	0.95
	$4.5 * 10^{-3}$	0.86
	$4.5 * 10^{-1}$	1.07
$4.5 * 10^{-4}$	$4.5 * 10^{-9}$	0.93
	$4.5 * 10^{-7}$	0.93
	$4.5 * 10^{-6}$	0.99
	$4.5 * 10^{-5}$	0.94
	$4.5 * 10^{-4}$	0.97
	$4.5 * 10^{-3}$	0.97
	$4.5 * 10^{-1}$	1.03
$4.5 * 10^{-3}$	$4.5 * 10^{-9}$	0.94
	$4.5 * 10^{-7}$	0.92
	$4.5 * 10^{-6}$	1.01
	$4.5 * 10^{-5}$	0.96
	$4.5 * 10^{-4}$	1.00
	$4.5 * 10^{-3}$	0.92
	$4.5 * 10^{-1}$	1.04
$4.5 * 10^{-2}$	$4.5 * 10^{-9}$	1.09
	$4.5 * 10^{-7}$	1.02
	$4.5 * 10^{-6}$	0.94
	$4.5 * 10^{-5}$	0.97
	$4.5 * 10^{-4}$	0.89
	$4.5 * 10^{-3}$	0.97
	$4.5 * 10^{-1}$	1.15

Table 14: MNIST 1000 label Fully Connected hyperparameter search. In bold is the chosen hyperparameter

Supervised Adversarial Factor	Unsupervised Adversarial Factor	Result
$\mathbf{4.5 * 10^{-5}}$	$4.5 * 10^{-9}$	0.64
	$\mathbf{4.5 * 10^{-7}}$	**0.58**
	$4.5 * 10^{-5}$	0.64
	$4.5 * 10^{-3}$	0.63
	$4.5 * 10^{-1}$	0.69
$4.5 * 10^{-4}$	$4.5 * 10^{-9}$	0.70
	$4.5 * 10^{-7}$	0.65
	$4.5 * 10^{-5}$	0.62
	$4.5 * 10^{-3}$	0.66
	$4.5 * 10^{-1}$	0.72
$4.5 * 10^{-3}$	$4.5 * 10^{-9}$	0.61
	$4.5 * 10^{-7}$	0.62
	$4.5 * 10^{-5}$	0.68
	$4.5 * 10^{-3}$	0.66
	$4.5 * 10^{-1}$	0.70
$4.5 * 10^{-2}$	$4.5 * 10^{-9}$	0.74
	$4.5 * 10^{-7}$	0.71
	$4.5 * 10^{-5}$	0.70
	$4.5 * 10^{-4}$	0.61
	$4.5 * 10^{-1}$	0.73

Table 15: MNIST 100 label Convolutional hyperparameter search. In bold is the chosen hyperparameter

Supervised Adversarial Factor	Unsupervised Adversarial Factor	Result
$4.5 * 10^{-5}$	$4.5 * 10^{-9}$	20.21
	$4.5 * 10^{-7}$	20.27
	$4.5 * 10^{-5}$	20.18
	$4.5 * 10^{-3}$	19.69
	$4.5 * 10^{-1}$	48.15
$\mathbf{4.5 * 10^{-4}}$	$4.5 * 10^{-9}$	19.88
	$\mathbf{4.5 * 10^{-7}}$	**19.15**
	$4.5 * 10^{-5}$	19.96
	$4.5 * 10^{-3}$	20.52
	$4.5 * 10^{-1}$	32.35
$4.5 * 10^{-3}$	$4.5 * 10^{-9}$	19.61
	$4.5 * 10^{-7}$	19.66
	$4.5 * 10^{-5}$	19.42
	$4.5 * 10^{-3}$	19.51
	$4.5 * 10^{-1}$	38.22
$4.5 * 10^{-2}$	$4.5 * 10^{-9}$	20.11
	$4.5 * 10^{-7}$	25.18
	$4.5 * 10^{-5}$	20.23
	$4.5 * 10^{-4}$	25.21
	$4.5 * 10^{-1}$	28.04

Table 16: CIFAR10 4000 label Convolutional hyperparameter search. In bold is the chosen hyperparameter

6.3 Results

We will present the results for CIFAR10 4000 labels, MNIST 100 labels Convolutional, MNIST 100 labels FC, MNIST 1000 labels FC and MNIST fully labeled FC. For each experiment we have perform a 10 seed random weight initialization.

6.3.1 MNIST fully labeled Fully Connected

Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result %	0.53	0.55	0.54	0.62	0.56	0.53	0.58	0.56	0.61	0.57	**0.565(±0.0310)**

Table 17: Baseline Experiment

Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	0.50	0.52	0.59	0.53	0.55	0.55	0.56	0.64	0.59	0.54	**0.557(±0.0406)**

Table 18: Adversarial noise supervised $\tau = 0.00045$ unsupervised $\tau = 0.0$ MNIST FC fully labeled

Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	0.48	0.59	0.59	0.58	0.60	0.63	0.56	0.59	0.50	0.54	**0.560(±0.0467)**

Table 19: Adversarial noise supervised $\tau = 0.00045$ unsupervised $\tau = 0.000000045$ MNIST FC fully labeled

6.3.2 MNIST 100 labels Fully Connected

The next table shows the results for each seed and the average result.

Baseline Model											
Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	1.01	0.88	1.06	0.92	1.02	0.93	0.98	2.22	0.95	1.11	**1.108(±0.3967)**

Table 20: Baseline Results MNIST FC 100 labels

Adversarial Noise Addition											
Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	0.94	0.88	1.05	1.00	2.26	0.97	0.98	2.24	1.03	0.90	**1.225(±0.5428)**

Table 21: Adversarial noise supervised $\tau = 0.00045$ unsupervised $\tau = 0.00000045$ MNIST FC 100 labels

Adversarial Noise Addition											
Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	0.91	0.87	0.96	0.94	0.99	1.07	0.96	2.23	0.92	0.99	**1.084(±0.4063)**

Table 22: Adversarial noise supervised $\tau = 0.045$ unsupervised $\tau = 0.00045$ MNIST FC 100 labels

6.3.3 MNIST 1000 labels Fully Connected

The next two tables shows the result for MNIST 1000 label FC.

Baseline Model											
Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	0.93	0.96	1.00	1.03	0.99	0.85	0.98	0.89	0.88	1.01	**0.952(±0.0614)**

Table 23: Baseline Results MNIST FC 1000 labels

Adversarial Noise Addition											
Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	0.83	1.00	0.99	0.97	0.85	0.98	1.02	0.85	0.90	0.93	**0.932(±0.0702)**

Table 24: Adversarial noise supervised $\tau = 0.000045$ unsupervised $\tau = 0.000045$ MNIST FC 1000 labels

6.3.4 MNIST 100 labels Convolutional

Baseline Model											
Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	0.72	0.56	1.28	0.58	0.65	1.27	0.71	1.99	0.64	0.57	**0.897(±0.4708)**

Table 25: Baseline Results MNIST Convolutional 100 labels

Adversarial Noise Addition											
Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	0.65	0.61	0.63	0.69	0.61	0.70	0.69	2.10	0.71	0.57	**0.7960(±0.4605)**

Table 26: Adversarial noise supervised $\tau = 0.000045$ unsupervised $\tau = 0.000045$ MNIST FC 1000 labels

6.3.5 CIFAR10 4000 labels Convolutional

Baseline Model											
Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	20.35	20.33	20.46	19.60	21.01	20.47	19.65	19.89	19.88	20.32	**20.196(±0.4360)**

Table 27: Baseline Results CIFAR Convolutional 4000 labels

Adversarial Noise Addition											
Seed	1	2	3	4	5	6	7	8	9	10	**Average**
Result	19.15	19.81	19.95	19.67	21.49	20.5	19.43	19.86	20.35	20.02	**20.02(±0.65)**

Table 28: Adversarial noise supervised $\tau = 0.000045$ unsupervised $\tau = 0.000045$ CIFAR10 Convolutional 4000 labels

7 Discussion

In this work we have trained the ladder autoencoder network model with adversarial noise to get state of the art results. On one side we add adversarial noise to the labeled data to improve generalization as exposed in previous works where the adversarial noise addition showed a good performance. On the other, we use the same idea of adversarial noise to add it to the unlabeled data, showing that it can help the unsupervised learning because it modifies the data space pushing the samples towards the most sensible direction our model has to discriminate. We have reached state of art in the same tasks where the previous ladder network had outperformed the MNIST and CIFAR10 classification tasks.

The most relevant improvement has been done in the MNIST Convolutional task, where we have 10 errors less and in the CIFAR10. In the MNIST FC we have only 2 errors less. The reason for this could be that the FC model with that architecture cannot be more expressive than it is. The addition of adversarial noise shows a better performance but not so relevant for this fact. In the CIFAR10 we take the error rate 0.17% which means we have 17 error less. This is no so significant as the MNIST Convolutional for the fact that the error rate in CIFAR10 was higher (20%) than MNIST Convolutional (0.8%). One of the things we have not try is checking the no addition of unsupervised adversarial noise in the rest of the models. In the FC MNIST fully labeled we have not reached better results than the model with only supervised adversarial noise, but could be for the exposed reason of the lack of expressivity of the model. However, the results are better than the baseline model and that ensures that adding adversarial noise to the unsupervised part have sense. Moreover, if we look at the tables of the hyperparmeter searching for the other models, we see that we do not get better performance as the power of unsupervised adversarial noise get to 0.

Another interesting thing we have noticed is that the addition of adversarial noise have a strange influence in the different models (the different parameter initialization). Looking at one model we see how sometimes the good models achieved without adversarial noise are worsened when this noise is added. On the other hand, poor models cannot be improved with adversarial noise addition. This have sense because if the parameters initialization is not good we cannot do better. We have only seen a model (the MNIST Convolutional) in which we have highly improved the poor models of the experiment. However, this were not the worst model of the experiment and we should check other parameter initialization to see if this hypothesis is true.

We did not find an exact correlation with the power of adversarial noise in the different tasks. We thought that the power in the FC MNIST problem would be the same or at least similar between the three experiments, but it was not. This is something to explore in the future because adversarial noise depends

on the supervised cost function and this cost function is different with different number of labeled data.

Future work will be focus on exploring new possible techniques for semi-supervised learning. One of the keys that cannot let us conclude if adversarial noise improve the performance of the ladder network is the high variability of the hyperparemeters of this model. The authors said they did exhaustive hyperparameter search. For example it does not make sense adding some parameters in BN to the softmax in CIFAR10 task and not adding them in the MNIST. The weights of the unsupervised cost are also something that need lots of searching. The key could be searching new hyperparmeters of the ladder network for the adversarial noise addition. However, the aim of this work was showing if adversarial noise can achieved better results given a state of art model, and we prove it can. We let this for future work. Another thing to explore is if we can substitute the gaussian noise for adversarial noise. As we said the worst that adversarial noise can do is a gaussian perturbation. The last thing we let for the future is adding adversarial noise in each layer of the network and find relations of the power with the dimensionality of the layers. In this work it was only added to the input.

References

R. Battiti. Accelerated backpropagation learning: Two optimization methods. *Complex systems*, 3(4):331–342, 1989.

Y. Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, Jan. 2009. ISSN 1935-8237. doi: 10.1561/2200000006. URL `http://dx.doi.org/10.1561/2200000006`.

J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, Feb. 2012.

C. Bishop. Latent variable models. In *Learning in Graphical Models*, page 371–403. MIT Press, January 1999. URL `https://www.microsoft.com/en-us/research/publication/latent-variable-models/`.

C. M. Bishop. Neural networks for pattern recognition, 1995.

C. M. Bishop. Pattern recognition and machine learning, 2006.

J. Chang and Y. Chen. Batch-normalized maxout network in network. *CoRR*, abs/1511.02583, 2015. URL `http://arxiv.org/abs/1511.02583`.

K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *J. Mach. Learn. Res.*, 7:551–585, Dec. 2006. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=1248547.1248566`.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In G. J. Gordon and D. B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, pages 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011. URL `http://www.jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf`.

I. J. Goodfellow, A. Courville, and Y. Bengio. Spike-and-slab sparse coding for unsupervised feature discovery. *arXiv preprint arXiv:1201.3382*, 2012.

I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. C. Courville, and Y. Bengio. Maxout networks. *ICML (3)*, 28:1319–1327, 2013.

I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

B. Graham. Fractional max-pooling. *CoRR*, abs/1412.6071, 2014. URL `http://arxiv.org/abs/1412.6071`.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`.

G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Comput.*, 14(8):1771–1800, Aug. 2002. ISSN 0899-7667. doi: 10.1162/089976602760128018. URL `http://dx.doi.org/10.1162/089976602760128018`.

G. E. Hinton. A practical guide to training restricted boltzmann machines. In G. Montavon, G. B. Orr, and K.-R. Müller, editors, *Neural Networks: Tricks of the Trade (2nd ed.)*, volume 7700 of *Lecture Notes in Computer Science*, pages 599–619. Springer, 2012. ISBN 978-3-642-35288-1. URL `http://dblp.uni-trier.de/db/series/lncs/lncs7700.html#Hinton12`.

G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

A. Krizhevsky, I. Sutskever, and G. 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. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

Y. Lecun and C. Cortes. The MNIST database of handwritten digits. URL `http://yann.lecun.com/exdb/mnist/`.

A. Rasmus, H. Valpola, M. Honkala, M. Berglund, and T. Raiko. Semi-supervised learning with ladder network. *CoRR*, abs/1507.02672, 2015. URL `http://arxiv.org/abs/1507.02672`.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. doi: 10.1038/323533a0.

R. Salakhutdinov and G. E. Hinton. Deep boltzmann machines. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 1, page 3, 2009.

J. Särelä and H. Valpola. Denoising source separation. *J. Mach. Learn. Res.*, 6: 233–272, Dec. 2005. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=1046920.1058110`.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL `http://arxiv.org/abs/1409.1556`.

J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014. URL `http://arxiv.org/abs/1412.6806`.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, Jan. 2014. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=2627435.2670313`.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014a. URL `http://arxiv.org/abs/1409.4842`.

C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2014b.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

H. Valpola. From neural pca to deep unsupervised learning. *Adv. in Independent Component Analysis and Learning Machines*, pages 143–171, 2015.

B. van Merriënboer, D. Bahdanau, V. Dumoulin, D. Serdyuk, D. Warde-Farley, J. Chorowski, and Y. Bengio. Blocks and fuel: Frameworks for deep learning. *CoRR*, abs/1506.00619, 2015. URL `http://arxiv.org/abs/1506.00619`.

P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390294. URL `http://doi.acm.org/10.1145/1390156.1390294`.

P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *J. Mach. Learn. Res.*, 11:3371–3408, Dec. 2010. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=1756006.1953039`.

L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus. Regularization of neural networks using dropconnect. In S. Dasgupta and D. Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1058–1066. JMLR Workshop and Conference Proceedings, May 2013. URL `http://jmlr.org/proceedings/papers/v28/wan13.pdf`.

J. Weston, F. Ratle, H. Mobahi, and R. Collobert. Deep learning via semi-supervised embedding. In *Neural Networks: Tricks of the Trade*, pages 639–655. Springer Berlin Heidelberg, 2012.

J. Zhao, M. Mathieu, R. Goroshin, and Y. Lecun. Stacked what-where auto-encoders. *arXiv preprint arXiv:1506.02351*, 2015.

8 Appendix

8.1 Appendix 1: Activation Functions

This appendix shows the principle activation functions used in neural network activations.

Linear

$f(z_i) = z_i$

Step

$$f(z_i) = \begin{cases} 0, & if\ z_i \leq 0 \\ 1, & else \end{cases}$$

Sigmoid

$f(z_i) = \frac{1}{1+\exp -z_i}$

Hyperbolic Tangent

$f(z_i) = \frac{\exp z_i - \exp -z_i}{\exp z_i + \exp z_i}$

Softmax

$f(z_i) = \frac{\exp z_i}{\sum\limits_{\forall j} \exp z_j}$

Rectifier Linear Unit

$f(z_i) = \max(0, z_i)$

Leaky Rectifier Linear Unit

$$f(z_i) = \begin{cases} z_i, & if\ z_i \leq 0 \\ 0.1 \cdot z_i, & else \end{cases}$$

8.2 Appendix 2: Cost Functions

This appendix briefly explain the machine learning paradigm.

A machine learning problem is a problem in which we try to learn a function, f from a set of examples, $\mathcal{X}$. This function f maps the input x to an output t.

The way we learn the function is by defining a cost function, C, which represent a relation between the output t of f and the desired output $\widehat{t}$ of our function. This relation is normally a measurement of the mismatch between $\widehat{t}$ and t. The objective is to minimize or maximize (depending on how C is defined), that is, finding singular points in C which respect to the parameters of f. This means we try to find the parameters that maps x to t in the way t is as close as possible to $\widehat{t}$. This shows how important is the definition of the cost function. Different cost functions can lead to different improvements in the learning process for the same data set and network topology.

Cost Functions

We describe some popular cost functions. Each one depends on how the t is required to be or the kind of problem we approach: classification, regression... Depending on the outputs constraints some cost functions are valid or not. We will define this function over a set of pairs $\mathcal{X} = \{(X_1, T_1), (X_2, T_2), ..., (X_N.T_N)\}$ in a problem with an output of K dimensions (classification problem of K classes). Why using this cost functions (which for sure can be derived from statistical principles) is widely exposed in chapter 6 from [Bishop, 1995].

Categorical Cross-Entropy

Categorical Cross-Entropy is defined for outputs, t, that must be in the 0-1 range and must sum 1.

$$C = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\widehat{T}_i(k) * \log(f(X_i(k))) + (1 - \widehat{T}_i(k)) * \log(1 - f(X_i(k))) \quad (60)$$

Minimum Squared Error

For unconstrained outputs.

$$C = \frac{1}{2} * \sum_{i=1}^{N}\sum_{k=1}^{K}(\widehat{T}_i(k) - f(X_i(k)))^2 \quad (61)$$

We can also have the normalized version which is a particularization of the root mean square:

$$C = \frac{1}{2 * |\mathcal{X}|} * \sum_{i=1}^{N} \sum_{k=1}^{K} (\widehat{T}_i(k) - f(X_i(k)))^2 \tag{62}$$

This is useful for the purpose of evaluating the performance of a model in predicting new data, where the test set can have different shapes [Bishop, 1995]. For the purpose of learning a function both errors are the same, and so is the root mean square, due to the fact that we add monotic transformations.

Minimum Classification Error

The minimum classification error function is not really used in optimization. MCE is just counting the errors. This kind of function is non differentiable so it cannot be minimized using gradient guided methods.

We now show three examples of cost functions. This cost functions are evaluated over a 0-hidden layer neural network in a $\mathbb{R}^1$ input space. This allows plotting the cost functions as a function of the parameters. We use the sigmoid activation function as the output which ensures that the output is the posterior probability in a two class problem. We need this in order to compute the categorical cross entropy. We use a 0.5 threshold for classifying. This also allows the output to be represented with only one neuron, necessary for the purpose of only having to parameters so we can represent the cost function. The problem consist of a data set drawn from different gaussian distributions. The network topology is given by figure 28. The aim of these examples are showing characteristics of the cost functions which shows why optimization is really difficult some examples.

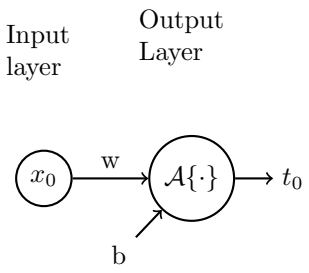
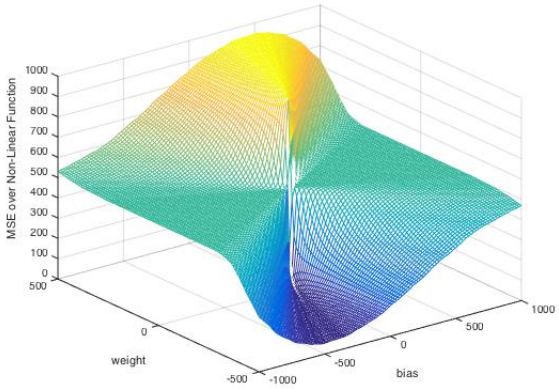


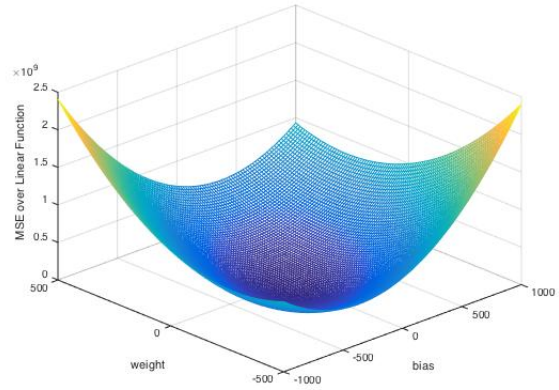
Figure 28: 2D input data space with linear activation function

Remark that the below model is also evaluated without the activation function, that is, a linear model over the parameters. Only MSE is evaluated for this model because now the output is unconstrained. It is well known that a function with the form $f(x, y, z, ..., t) = (a^2 * x + b * y + c * z + ... + d^3 * t)^2$, where the expression inside the parenthesis is linear wrt the independent variables of the function, is concave (or convex). This means that the MSE cost function evaluated at a neural network without activation function should have

this shape. When adding the activation function should not. Figure 29 shows this two examples.



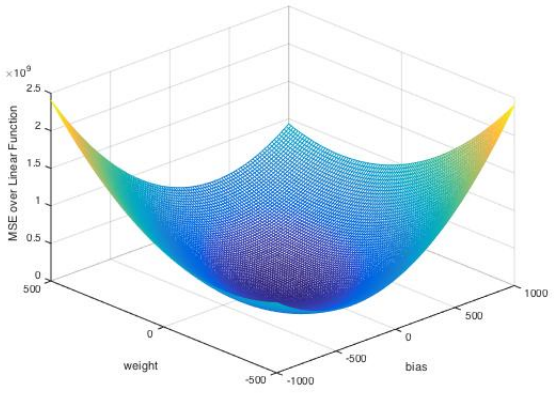
(a) Model: $t = \frac{1}{1+\exp(w*x+b)}$



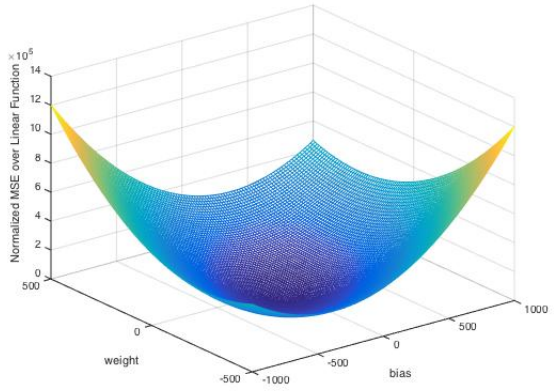
(b) Model: $t = w * x + b$

Figure 29: MSE cost Function

The next figure shows the normalized MSE versus the MSE. It should be noted that the shape is the same changing the value of the cost. The optimal point is the same in both functions.

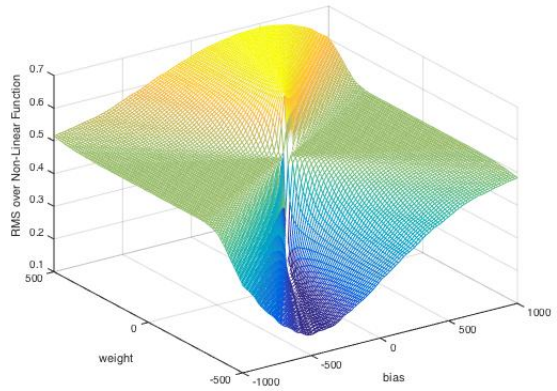


(a) MSE

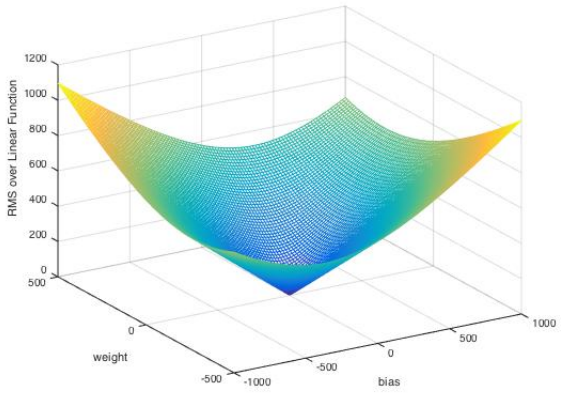


(b) Normalized MSE

The next figure shows the RMS with and without activation function. It should be noted that the optimal point is at the same point as it was in figure 29. When adding monotone transformation or scale parameter the optimization does not change, but the computation can be increased (for example when adding the derivative of a root). In this case the shape of the function changes due to the fact that the square root is a non linear operation.



(a) Model: $t = \frac{1}{1+\exp(w*x+b)}$



(b) Model: $t = w * x + b$

Figure 31: RMS cost Function

Finally we show the cross-entropy in which we do not have any concave or convex shape due to the fact that the cost function does not meet the requirements to have this shape and due to the fact that the projection is not strictly linear wrt to the parameters.

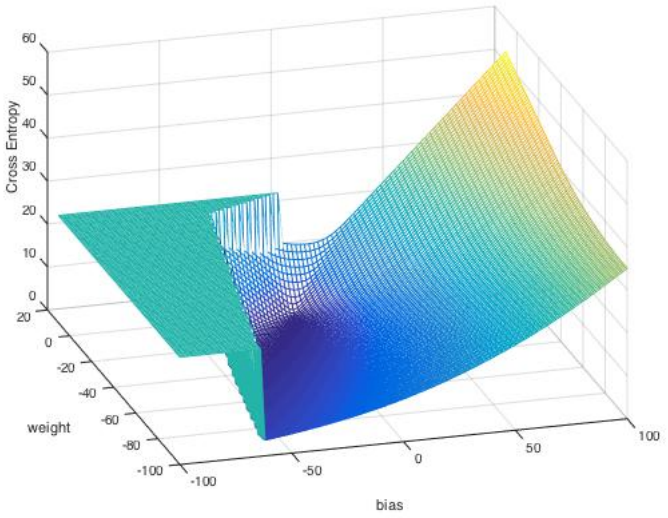


Figure 32: Cross Entropy over the model

All the improvements done in the SGD that are exposed in the state of the art are oriented to solve some problems that can be seen in these pictures. We can see that when having concave functions the result of the optimization is the same, for a fixed dataset, no matter where we start the iterative algorithm. The problem is that having this shape implies having simple models. Using other cost functions (which implies different things, see [Bishop, 1995]) also changes this shape. When having other shapes, we see that we can have other points where the gradient is 0 (see figure 29 (a)), or changes in the slope of the function which can make difficult the search of the optimum (see figure 4).

Requirements for back propagation

The back propagation algorithm [Rumelhart et al., 1986] is an algorithm used to minimize cost functions C in Neural Networks. It is an Stochastic Gradient Descent minimization. To apply back propagation the cost function must satisfy this requirements:

- Cost Functions must be differentiable.
- All the mappings in f must be differentiable (we really require the activations, $\mathcal{A}$ to be differentiable).
- The cost function must only be computed from elements in the output layer.

8.3 Appendix 3: Energy Based Models

The purpose of this appendix is to briefly explain energy based models for beginners in probability theory.

The concept of energy model come from thermodynamics physics. In this kind of systems the more the stable is the system the lower is the total amount of energy. The explanation will be based in [Hinton, 2012].

An energy based probability model is defined as:

$$p(x) = \frac{e^{-E(x)}}{\sum\limits_{\forall x} e^{-E(x)}} \tag{63}$$

As we can see low probability vectors would have big energy and viceversa. One of the typical ways of learning generative models with unsupervised data is maximizing the likelihood of the function wrt parameters. Maximizing this function is achieved by taking the derivative of $p(x)$ wrt to the parameters $\mathcal{P}$. Let's call $Z = \sum\limits_{\forall x} e^{-E(x)}$ the partition function.

On the other hand it is very typical to add hidden variables to the model to make the model much more powerful. These hidden variables typically represent subspaces that condense important information to represent data. Models are trained to achieve this property.

We could rewrite equation 63 as:

$$p(x) = \frac{\sum\limits_{\forall z} e^{-E(x,z)}}{\sum\limits_{\forall x}\sum\limits_{\forall z} e^{-E(x,z)}} \tag{64}$$

And now $Z = \sum\limits_{\forall x,z} e^{-E(x,z)}$

The derivation of equation 64 is quite simple. Let's rewrite equation 64 to have a similar form to equation 63.

$$p(x) = \frac{e^{-F(x)}}{\sum\limits_{\forall x} e^{-F(x)}} \tag{65}$$

Where $F(x) = -\log \sum\limits_{\forall h} e^{-E(x,h)}$ is the free energy function (also coming from thermodynamics). The log likelihood derivation of equation 65 is:

$$\frac{\partial \log p(x)}{\partial \mathcal{P}} = \frac{Z \cdot [e^{-F(x)}]' - e^{-F(x)} \cdot Z'}{Z \cdot e^{-F(x)}} = -\frac{\partial F(x)}{\partial \mathcal{P}} + \frac{1}{Z} \cdot \sum_{\forall x'} \frac{\partial F(x')}{\partial \mathcal{P}} \cdot e^{-F(x')} \tag{66}$$

Equation 66 is the derivation for only one training vector. If we have N training vectors, $\mathcal{X} = \{X_1, X_2, .., X_N\}$ the expression for the derivation of the log probability is rewritten as:

$$\sum_{\forall X} \frac{\partial \log p(x)}{\partial \mathcal{P}} = \sum_{\forall X} - \frac{\partial F(x)}{\partial \mathcal{P}} + \sum_{\forall X} \frac{1}{Z} \cdot \sum_{\forall x'} \frac{\partial F(x')}{\partial \mathcal{P}} \cdot e^{-F(x')} \tag{67}$$

Taking in consideration equation 65:

$$\sum_{\forall X} \frac{\partial \log p(x)}{\partial \mathcal{P}} = -\sum_{\forall X} \frac{\partial F(x)}{\partial \mathcal{P}} + -\sum_{\forall X} \sum_{\forall x'} \frac{\partial F(x')}{\partial \mathcal{P}} \cdot p(x') \tag{68}$$

So the second term of the sum in right side of equation 68 is the expectation of the derivative of $F(x)$. Multiplying by $\frac{1}{N}$ would not change the result for the fact that we perform a monotic operation. This would give us the average log likelihood given by:

$$\frac{1}{N} \sum_{\forall X} \frac{\partial \log p(x)}{\partial \mathcal{P}} = -\frac{1}{N} \sum_{\forall X} \frac{\partial F(x)}{\partial \mathcal{P}} + \frac{1}{N} \sum_{\forall X} \sum_{\forall x'} \frac{\partial F(x')}{\partial \mathcal{P}} \cdot p(x') \tag{69}$$

Considering that the expectation of the derivative does not depend on X that summation would be equal to N so we could finally rewrite our expression like:

$$\mathbb{E}_x[\frac{\partial \log p(x)}{\partial \mathcal{P}}] = -\mathbb{E}_x[\frac{\partial F(x)}{\partial \mathcal{P}}] + \mathbb{E}_{x'}[\frac{\partial F(x')}{\partial \mathcal{P}}] \tag{70}$$

These expectations are over the training data, $\mathbb{E}_x[\cdot]$ and over the model distribution, $\mathbb{E}_{x'}[\cdot]$.

The next step is dependent on how the energy function is defined. According to the result of the derivative we have to sample data from the model distribution (which can be done from the training data) and use the training data to compute the expectation over the training set (with sampling needed when adding hidden variables). To end with, let's take a look to the derivative of the free energy function. Note that our model is defined in terms of the energy function so it seems reasonable to have our derivative in terms of the energy function. Taking in consideration the conditional probability:

$$p(h|x) = \frac{p(x,h)}{p(x)} = \frac{e^{-E(x,h)}}{\sum\limits_{\forall h} e^{-E(x,h)}} \tag{71}$$

The log likelihood after deriving, the free energy function is:

$$\mathbb{E}_x[\frac{\partial \log p(x)}{\partial \mathcal{P}}] = -\mathbb{E}_x[\sum_{\forall h} p(h|x) \frac{\partial E(x,h)}{\partial \mathcal{P}}] + \mathbb{E}_{(x',h)}[\frac{\partial E(x',h)}{\partial \mathcal{P}}] \tag{72}$$

And this means that for computing the gradient we have to sample h for each training sample X to model the conditional expectation of the gradient in the first term of the sum and we need to sample pairs of (x', h) from the model

distribution to compute the expectation of the second term. The expectation of the second term is approximated as:

$$\mathbb{E}_{(x',h)}[\frac{\partial E(x',h)}{\partial \mathcal{P}}] = \sum_{(x',h)} p(x',h) \cdot \frac{\partial E(x',h)}{\partial \mathcal{P}} \approx \frac{1}{|\mathcal{Q}|} \sum_{\forall (x',h) \in \mathcal{Q}} \frac{\partial E(x',h)}{\partial \mathcal{P}} \quad (73)$$

Where our set $\mathcal{Q} = \{(X_1, H_1), (X_2, H_2), ..., (X_N, H_N)\}$ are pairs sampled from the model distribution.