MNIST classification using deep learning

Thomas Grivaz

School of Computer and Communication Sciences

Semester Project

May 2016

Responsible
Prof. Pierre
Vandergheynst
EPFL / LTS2

Supervisors
Nathanaël Perraudin
Nauman Shahid
EPFL / LTS2

Contents

Introduction

The goal of this project was to recognize handwritten digits by implementing a deep neural network on `Matlab`. Deep learning has become increasingly prevalent to do complex visual pattern recognition tasks that involve a large number of training examples.
The project aims to build a feed-forward neural network from scratch and then train it on a hands-on dataset. The first part of the project was dedicated to learn and implement the gradient descent algorithm on smaller convex problems. Afterwards, the efforts were focused on starting to build the architecture of the neural network and understanding the underlying concepts and mechanisms. Then the backpropagation algorithm was implemented as well as other functions related to training. Finally, once the neural network was fully implemented and functional, we trained it on a large dataset and learned how to tune some hyperparameters, their respective effect and purpose and to optimize them in order to reach an optimal performance. This report summarizes our findings and results.

The project involved a lot of theoretical work to be at ease with the mathematical concepts related to neural networks but was also practically oriented as it required implementing and testing a functional model. We also worked with a real dataset and were confronted with the same type of problems that a data scientist would have on the field.

Chapter 1

Theory

In this chapter we formalize the general theory used for the project. This will serve as a reference for the model we use in chapter 2.

1.1 Statistical Classification

In machine learning and statistics, classification is the problem of identifying to which of a set of categories a new observation belongs. We assign to a vector $\mathbf{x}$, known as an *instance*, where each component of the vector is called a *feature*, one of K discrete *classes* C_k where k =1,...,K. This task is performed on the basis of a model that has been trained on a dataset containing observations whose category membership is known [1]. This is an instance of supervised learning. A good classifier should be able to generalize, meaning that it should perform well (i.e. correctly classify) on data it has never encountered before, based on the probability of samples belonging to each of the classes.

1.2 From Linear to Nonlinear Classification

We will introduce here some motivational theory as to why using neural networks for classification. A large number of algorithms for classification can be phrased in terms of a linear function, also called a *discriminant function* that assigns a score to each possible category k by combining the feature vector of an instance with a vector of weights, using a dot product [1]. This type of score function has the following form:

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0 \tag{1.1}$$

where $\mathbf{w}$ is called the *weight vector*, and w_0 is called the *bias.* For linear discriminants, the decision surfaces are represented by hyperplanes. Without loss of generality we consider the case of two classes for this example. An input vector $\mathbf{x}$ is assigned to class C_1 if $y(\mathbf{x}) \geqslant 0$ and to class C_2 otherwise. The corresponding decision boundary is therefore defined by the relation $y(\mathbf{x}) = 0$, which corresponds to a (D - 1)-dimensional hyperplane within the D-dimensional input space [2].

A more powerful approach than using discriminant functions involves modeling the conditional probability distribution $p(C_k|\mathbf{x})$ in an inference stage, and then subsequently using this distribution to make optimal decisions. Using Bayes' theorem, we are able to compute the posterior probabilities by means of an *activation function* that maps the output in the range (0,1). The generalization of our previous model becomes :

$$y(\mathbf{x}) = f(\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0) \tag{1.2}$$

Where f is the activation function. The decision surfaces correspond to $y(\mathbf{x}) = constant$, so that $\mathbf{w}^T\mathbf{x} + w_0 = constant$ and hence the decision surfaces are linear functions of $\mathbf{x}$, even if the function $f(.)$ is nonlinear [2].

The two models discussed above address only linearly separable problems meaning that there must exist hyperplanes that fully separate set of points based on their respective class. Problems where classes are linearly separable in the original input space $\mathbf{x}$ are very rare. We can define more complex decision boundaries by making a fixed nonlinear transformation of the inputs using a vector of basis function $\boldsymbol{\phi}(\mathbf{x})$. As long as classes are linearly separable in the new feature space $\boldsymbol{\phi}(\mathbf{x})$ we can still classify well, so why would we need nonlinear classifiers ?

Linear models are convenient to use because they are simple, have useful analytical and computational properties but their practical applicability is limited by the curse of dimensionality [3] in the sense that it is harder to obtain the same statistical relevance in higher dimensions. In order to deal with problems that are still not linearly separable in transformed feature spaces, we need to step from linear to nonlinear classification by adapting the basis functions to the data.
One can address this by first defining basis functions that are centred on the training data points and then selecting a subset of these during training, this is the approach taken by Support Vector Machines (SVM), more infos can be found in [4].
We can also fix the number of basis functions in advance but allow them to be adaptive i.e. make use of basis functions $\boldsymbol{\phi}(\mathbf{x}, \boldsymbol{\theta})$ that are parametrized by $\boldsymbol{\theta}$ that we will also learn during training. This is the approach taken by neural networks.

1.3 Multi-Layer Perceptrons

Artificial neural networks are a family of models which mimic brain function. A human brain contains a huge amount of nerve cells called neurons. These neurons are interconnected between each other forming complex networks. Artificial neural networks are similar to biological neural networks in the performing by its units (neurons) of functions collectively and in parallel, rather than by a clear delineation of subtasks to which individual units are assigned [5].

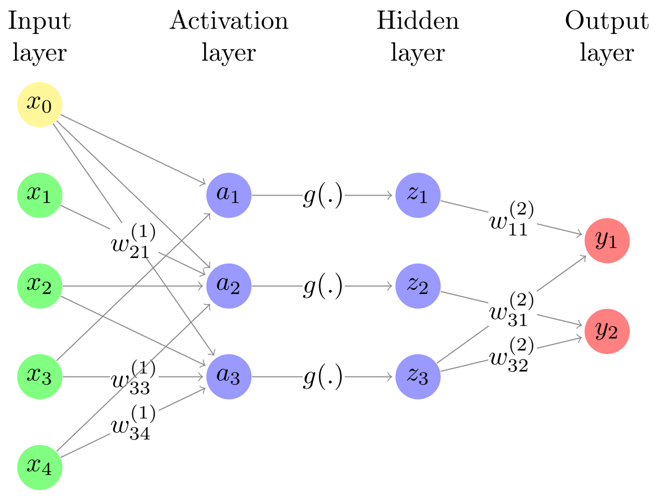


Figure 1.1: Network diagram of a two layers feed-forward neural network. Inputs, activation, hidden units and outputs are represented by nodes, x_0 is the bias parameter. The nonlinear transfer function and adaptive weights are represented by links between the nodes (some of them were not labeled for the sake of clarity). Picture by the author.

The type of neural network we used for our project is the feed-forward neural network, also called the *multi-layer perceptron*, such neural networks contain no feedback loops, as we will see later on. For the remaining of this report, the terms neural network and multi-layer perceptron will be used interchangeably.
A neural network can be seen as a series of functional transformations. We first define the *activations*, which are linear combinations of the input variables in the form:

$$a_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_j^{(1)} \tag{1.3}$$

Where $j = 1, ..., h_1$, the subscript (1) denotes that the parameter belongs to the first 'layer' of the network. We refer to the parameters $w_{ji}^{(1)}$ as *weights* and the parameters $b_j^{(1)}$ as *biases*. Next we pass each activation through a differentiable nonlinear *activation function* $g(.)$ which yields:

$$z_j^{(1)} = g(a_j^{(1)}) \tag{1.4}$$

These parameters are called *hidden units*. Typical functions for $g(.)$ are sigmoidal functions such as the logistic function or the 'tanh' function. Following our notation, $z_j^{(1)}$ is the output of j-th unit in the first layer. These units are then again linearly combined to feed the next layer of activations:

$$a_k^{(2)} = \sum_{k=1}^{M} w_{kj}^{(2)} z_j^{(1)} + b_k^{(2)} \tag{1.5}$$

And then the same process goes on for as many layers as we want, most of the models we use in practice involve one or two layers of adaptive weights. The activations of the last layer are called *output unit activations*. For classification, the output units activations are then transformed using an appropriate function to yield the network outputs y_k. The type of function we use depends on the nature of our problem, in the case of binary classification, each output activation unit is transformed using a logistic function so that:

$$y_k = \frac{1}{1 + \exp(-a_k)} \tag{1.6}$$

For multiclass classification, we use a softmax function of the form:

$$y_k = \frac{\exp(a_k)}{\sum\limits_j \exp(a_j)} \tag{1.7}$$

The various stages of the 'forward propagation' through the network can be seen on figure 1.1, which displays a two layers network.

As any machine learning algorithm, we need to define how costly our mistakes are. During training, the error function $E(\mathbf{w})$ quantifies the discrepancy between our predictions (the output unit activations) and the training set targets. Next we learn weights $\mathbf{w}$ by minimizing $E(\mathbf{w})$ using gradient descent and the backpropagation algorithm, as we will see later.

1.3.1 Vectorization and Matlab implementation

In this small section we will establish how the model formalism was translated into `Matlab` code. We begin by vectorizing the formulas seen before. We first define the input matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$, n being the number of samples and p the number of features. For the first layer we obtain :

$$\mathbf{A}^{(1)} = \mathbf{X}(\mathbf{W}^{(1)})^{\mathrm{T}} + \mathbf{b}^{(1)}, \quad \mathbf{Z}^{(1)} = g(\mathbf{A}^{(1)})$$

With $\mathbf{W}^{(1)} \in \mathbb{R}^{h_1 \times p}$ and $\mathbf{b}^{(1)} \in \mathbb{R}^{h_1}$, h_1 being the number of hidden units for the first layer. The corresponding `Matlab` code is:

Code 1.1: `nn_fwd.m`, lines 19-21

```
net.a{1} = X*net.w{1}';
net.z{1} = g(net.a{1});
net.z{1} = [ones(m,1), net.z{1}];
```

Where the appended column of ones acts as the bias parameter. The recurrence relationships defined in (1.3) and (1.4) are equivalent to:

$$\mathbf{A}^{(i)} = \mathbf{Z}^{(i-1)}(\mathbf{W}^{(i)})^{\mathrm{T}} + \mathbf{b}^{(i)}, \quad \mathbf{Z}^{(i)} = g(\mathbf{A}^{(i)})$$

Where $\mathbf{W}^{(i)} \in \mathbb{R}^{h_i \times h_{i-1}}$. Which is implemented as :

Code 1.2: `nn_fwd.m`, lines 25-29

```
 for i = 2 : n - 1
    net.a{i} = net.z{i-1}*net.w{i}';
    net.a{i} = [ones(m, 1), net.a{i}];
    net.z{i} = g(net.a{i});
end
```

The rest of the computations are straight forward, we pass the last layer of activations through the nonlinear output function to obtain the output unit activations, we then compute the error and the loss.

1.4 The backpropagation Algorithm

As stated before, our goal is to minimize the loss function $E(\mathbf{w})$. To this end, we need to compute the gradient $\nabla E(\mathbf{w})$ that will be used for gradient descent. Our main issue here is that our loss function is defined in terms of activations and not directly in terms of weights, applying the chain rule for partial derivatives gives us :

$$\frac{\partial E}{w_{kj}^{(i)}} = \frac{\partial E}{\partial a_k^{(i)}} \frac{\partial a_k^{(i)}}{\partial w_{kj}^{(i)}} \tag{1.8}$$

This is the intuition behind the backpropagation algorithm. To update the weights, we will first compute the derivatives with respect to the activations in a 'top-bottom' fashion and we will then use these derivatives to obtain the gradient $\nabla E(\mathbf{w})$ with respect to each layer of adaptive weights.

We begin by evaluating the *errors* or *residuals* for all the output unit activations:

$$r_j^{(n)} = \frac{\partial E}{\partial a_j^{(n)}} \tag{1.9}$$

Where n is the number of layers. Now using the chain rule and going one layer down we obtain:

$$r_q^{(i)} = \frac{\partial E}{\partial a^{(i+1)}} \frac{\partial a^{(i+1)}}{\partial a_q^{(i)}} = r^{(i+1)} \frac{\partial}{\partial a_q^{(i)}} \sum_{k'} w_{k'}^{(i+1)} g(a_{k'}^{(i)}) = r^{(i+1)} w_q^{(i+1)} g'(a_q^{(i)}) \tag{1.10}$$

Using (1.5) we can write:

$$\frac{\partial a_k^{(i)}}{\partial w_{kj}^{(i)}} = z_j^{(i-1)} \tag{1.11}$$

Now substituting (1.11) and (1.9) into (1.8) we finally obtain:

$$\frac{\partial E}{w_{kj}^{(i)}} = r_k^{(i)} z_j^{(i-1)} \tag{1.12}$$

Thus in order to obtain the weight derivatives, we first have to compute all the activations by doing a 'forward pass' through the network. Once we can compute our loss function, we do a 'backward pass' by recursively computing each residual. The last step is just multiplying hidden units and residuals to obtain the weight derivatives. These steps are displayed on figure 1.2.

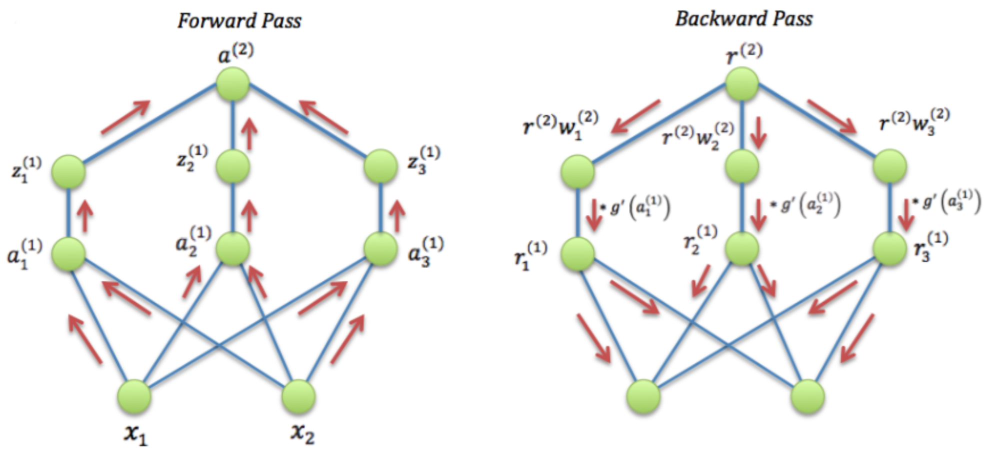


Figure 1.2: Forward pass and backward pass of error backpropagation for a two layers neural network. Picture taken from [6]

1.4.1 Vectorization and Matlab implementaion

We now again transform the sums into matrix-vector notation. We can directly write (1.10) as:

$$\mathbf{r}^{(i)} = (\operatorname{diag} g'(\mathbf{a}^{(i)}))\mathbf{r}^{(i+1)}\mathbf{W}^{(i+1)}$$

With $g'(\mathbf{a}^{(i)})$ depending on the choice of the activation function. The code is as follows:

Code 1.3: `nn_bwd.m`, lines 21-37

```
  for i = (n-1) : -1 : 1

    % compute g'(a_i)
    switch(net.act)
        case 'logistic'
            delta = net.z{i} .* (1 - net.z{i});
        case 'tanh'
            delta = sech(net.z{i}).^2;
    end
    % include bias term for the last layer
    if (i+1) == n
        net.r{i} = (net.r{i+1} * net.w{i+1}) .* delta;
    else
        net.r{i} = (net.r{i+1}(:,2:end) * net.w{i+1}) .* delta;
    end

end
```

We can now write (1.12):

$$\nabla_{\mathbf{W}^{(i)}} E = (\mathbf{r}^{(i)})^{\mathrm{T}}\mathbf{z}^{(i-1)}$$

Which corresponds to :

Code 1.4: `nn_bwd.m`, lines 39-48

```
  % use derivative to compute adjustements to be made to weights
net.dW{1} = net.r{1}(:,2:end)' * [ones(size(X,1), 1),X];
for i =1 : (n-1)
    % include bias term for last layer
    if (i+1) == n
        net.dW{i+1} = net.r{i+1}' * net.z{i};
    else
        net.dW{i+1} = net.r{i+1}(:,2:end)' * net.z{i};
    end
end
```

Note that similarly to code 1.1, we multiply the residuals by the input matrix to obtain the derivatives of the first layer.

1.5 Gradient Descent Optimization

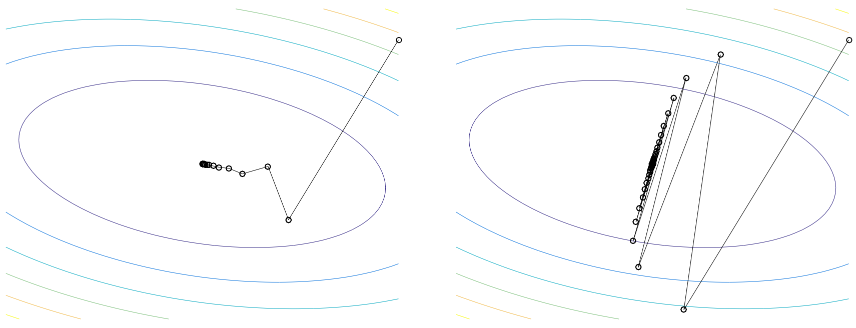
Now that we are capable of deriving the gradient of the weights of our model, we use gradient descent optimization to find a (hopefully global) mininimum of the function $E(\mathbf{w})$. Gradient descent works as follows : at each iteration, we update the weights $\mathbf{w}$ on the basis of the gradient of $E(\mathbf{w})$, since the gradient points to the direction of the largest increase of a function and since we want to minimize our error, we take steps in the opposite direction of the gradient until we find a minimum.

1.5.1 Simplest gradient descent

The simplest form of gradient descent just involves going in the direction of the steepest descent. Thus at iteration $k+1$ we update the weights :

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \nabla_{\mathbf{w}^{(k)}} E \tag{1.13}$$

Where η is the *learning rate* or *time step*. Depending on the properties of $E(\mathbf{w})$, under sufficient regularity assumptions, when the initial estimate $\mathbf{w}^{(0)}$ is close enough to the optimum and when the learning rate η is sufficiently small, this algorithm achieves linear convergence although η is somewhat tricky to setup and can lead to disastrous results depending on the value.



(a) $\eta = 0.2$ (b) $\eta = 0.3$

Figure 1.3: Gradient descent optimization on a 2D quadratic function

Two cases are illustrated on figure 1.3. On figure 1.3a the learning rate is set to $\eta = 0.2$, we can observe that the algorithm converges within a decent amount of iterations and the shape of the evolution of $\mathbf{w}$ is pretty smooth.

On figure 1.3b, $\mathbf{w}$ strongly oscillates from one iteration to another, this problem is know as *zig-zagging*. If E($\mathbf{w}$) is highly steep around a point $\mathbf{w}^{(k)}$, for a learning rate too large, we will go to the other side of the ravine which leads to slow overall progress, the algorithm doesn't converge in the worst case. The momentum method helps to cope with this issue.

1.5.2 The momentum method

The momentum method uses a convex combination of the steepest descent direction and the previous update step. Gradient descent with momentum method works as follows:

$$\triangle\mathbf{w}^{(k)} = -\eta(1-\mu)\nabla_{\mathbf{w}^{(k)}}E + \mu\triangle\mathbf{w}^{(k-1)} \tag{1.14}$$

Where η is the learning rate, μ is the momentum term, $\triangle\mathbf{w}^{(k)} = \mathbf{w}^{(k+1)} - \mathbf{w}^{(k)}$ is the update done at step k. The addition of the momentum term smoothes out oscillatory behavior while leaving systematic useful components in place, this guaranties a faster convergence than simple gradient descent [7]. With this method you can also have a larger learning rate while mitigating the risk of divergence.

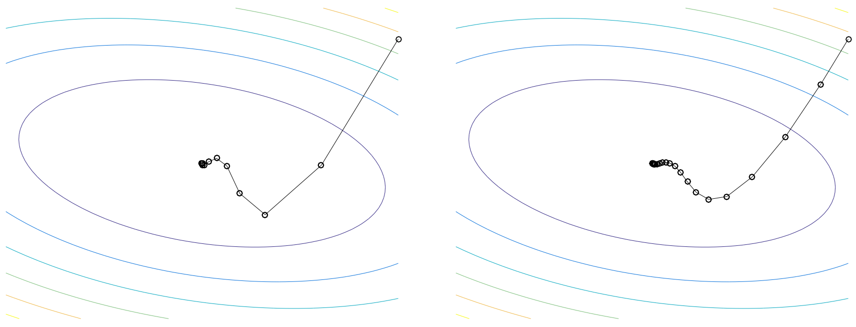

(a) $\mu = 0.3$ (b) $\mu = 0.5$

Figure 1.4: Gradient descent with momentum method

As we can observe from figure 1.4, adding a momentum term really smoothes out the previous shape of the descent, it also speeds up the convergence.

1.5.3 Adaptive learning rate

In addition to the momentum method, there exist a few techniques to locally adapt the learning rate $\eta_i^{(k)}$.

Bold driver

A useful batch method for adapting the global learning rate η is the bold driver method. It works as follows: after each iteration, you compute the relative loss value:

$$\triangle E(\mathbf{w})^{(k)} = E(\mathbf{w})^{(k)} - E(\mathbf{w})^{(k-1)}$$

If this quantity is negative, the learning rate becomes $\eta^{(k+1)} = c_1 * \eta^{(k)}$ with $c_1 > 1$ (you typically increase η by 15-20% . Conversely if the relative loss value is positive (our error has increased), the learning rate becomes $\eta^{(k+1)} = c_2 * \eta^{(k)}$ with $c_2 < 1$. A good rule of thumb is to reduce η by 50%. Depending on the nature of the loss function and whether the design matrix is ill-conditioned or not, this method can speed up the convergence rate of gradient descent by 50% or lead to a worse convergence rate [8].

Local rate adaptation

We can also use an online weight update that uses a local, time-varying learning rate for each weight :

$$\mathbf{w}_i^{(k+1)} = \mathbf{w}_i^{(k)} - \eta_i^{(k)} \nabla_{\mathbf{w}_{i^{(k)}}} E \tag{1.15}$$

The idea is to adapt these local learning rates by gradient descent, while simultaneously adapting the weights. At iteration k, we would like to change the learning rate before updating the weight such that the loss $E^{(k+1)}$ is reduced. Minimizing the loss with respect to $\eta_i^{(k)}$ gives us :

$$\frac{\partial E}{\partial \eta_i^{(k)}} = \frac{\partial E}{\partial w_i^{(k+1)}} \frac{\partial w_i^{(k+1)}}{\partial \eta_i^{(k)}} = -\nabla_{w_i^{(k+1)}} E \, \nabla_{w_i^{(k)}} E$$

Ordinary gradient descent using the meta learning rate α (a new global parameter) gives us :

$$\eta_i^{(k+1)} = \eta_i^{(k)} + \alpha \nabla_{w_i^{(k+1)}} E \, \nabla_{w_i^{(k)}} E$$

Note that this method can sometimes lead to negative learning rates.

Chapter 2

Analysis

In this chapter we describe our methodology and results of the training of our implemented neural network.

2.1 Dataset Description

The dataset we trained our model on is the MNIST database of handwritten digits [9]. It is a large database that is commonly used for training various image processing systems and machine learning models. It contains 60,000 training images and 10,000 testing images. Each observation of the dataset is a 28×28 pixels grayscale image, some examples are featured on figure 2.1. For the remaining of our analysis, 48,000 samples of the training set was kept for training while the remaining 12,000 served as a validation set, the full test set was also used.

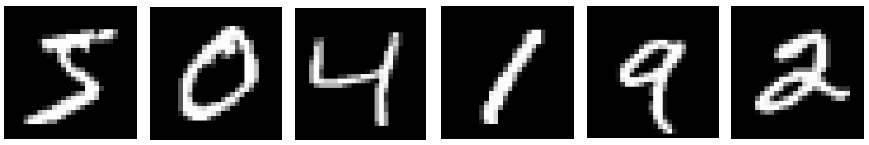


Figure 2.1: Examples of MNIST digits

2.2 Preprocessing & Features Extraction

The MNIST dataset has pixel values in the range $[0, 255]$. Thus a simple rescaling to shift the data into the $[0, 1]$ range was done in the first place. Even though the relative scales of pixels are already approximately equal, having small initial values can help to set up the weights accordingly, indeed if input variables are of order unity, we expect that the network weights should also be of order unity [10] and we avoid saturation. Secondly, a reshaping of the observations was made so that the data is more convenient to use, each digit image has been rescaled to a $28 \times 28 = 784$ dimensional vector and subsequently stored in the data matrix.

Besides simple rescaling, feature standardization was tried i.e. we independently set each dimension of the data to have zero-mean and unit variance. Feature standardization is most of the time a good practice to have as it can make training faster and reduce the chances of getting stuck in local minima. However, it seems that in the case of the MNIST dataset, standardization on top of normalization loses a lot of information and negatively affects the performance of the model. Indeed we observed during the training phase that it was nearly impossible to get a test error below 10% with this settings. For most of the state-of-the-art models previously trained on MNIST no preprocessing of this type was done either [11] which supports the idea that standardization should not be done in our case. My personal guess is that since a lot of pixel values are set to 0 (black) while some of them have non null values, apply a standardization column wise in fact reduces possible discriminative "power" of a feature.

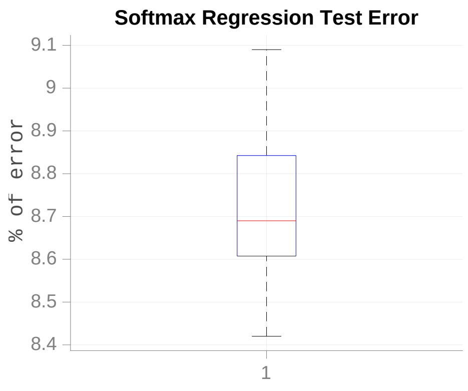
PCA was also tried before training but resulted in a drastical error increase. Moreover, applying dimensionality reduction techniques here is not necessarily relevant since the number of training examples is such that we most likely avoid the curse of dimensionality [12], eventually this would have helped with the running time but the tradeoff in accuracy is not worth it at all.

2.3 Baseline Model

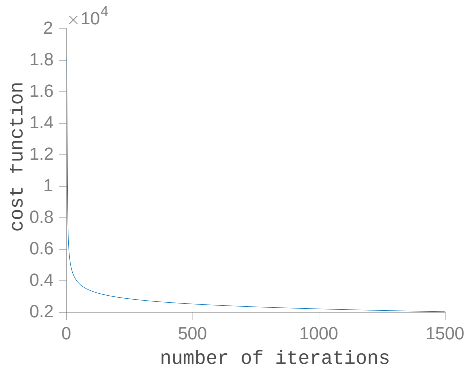
A softmax regression model was considered as a baseline. It is a generalization of logistic regression to classification problems. The cost function is of the form:

$$E(w) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{j=1}^{k} 1\{y^{(i)} = j\}\log\frac{e^{w_j^T x^{(i)}}}{\sum\limits_{l=1}^{k} e^{w_l^T x(i)}}\right]$$

We then minimize this function using regular gradient descent. For our problem, we used only 10,000 training cases and 2,000 testing cases chosen uniformly at random without replacement to keep a descent running time since it's the bottleneck here. The all zeros matrix was chosen as the starting point of the algorithm. Regarding the hyper parameters for gradient descent, a time step of $\alpha = 1 \times 10^{-4}$ was chosen as it was the biggest one that guaranteed convergence, a momentum term of $\mu = 0.5$ was added and we restricted the descent to 1500 iterations. We then computed the testing errors for five consecutive runs. Results obtained are displayed on figure 2.2.



(a) Boxplot of test error



(b) Evolution of cost function

Figure 2.2: Baseline model results

We estimated that the average test error is 8.7% (±0.4). The variance is satisfactory as we stay within a 1% range here. Regarding the evolution of the cost function, we can observe that while the problem is highly non-convex, gradient descent behaves smoothly with this parameters, the cost decreases rapidly for the 50 first iterations then reaches convergence in a local minimum. Other starting points were tried but the all zeros one gave the most satisfying results.

2.4 Neural Network Parameter Tuning

Now that we have our baseline model, we train our implemented neural network on the data and see how well it performs compared to the baseline and other existing models in the literature. This section summarizes the experiments and results for the various hyperparameters inherent to the neural network. Note that parameters were optimized in a "suboptimal" way, indeed because of our time and computational resources, we tuned the parameters one after the other with a trial and error method. Certain parameters can affect others thus finding the best values for parameters separately can lead to a result that isn't as good as if you would have searched in the parameter space in a combinatorial way. Optimal techniques involve a lot of computational time and more advanced algorithms, some examples can be found in [13].

For our project we proceeded in the following way: for each value of a parameter, we performed a 4-fold cross-validation i.e. 75% of the training set was randomly retained as training while the remaining 25% were used for validation, then we computed the generalization error on the full test set. The whole process is repeated for the four different slots.

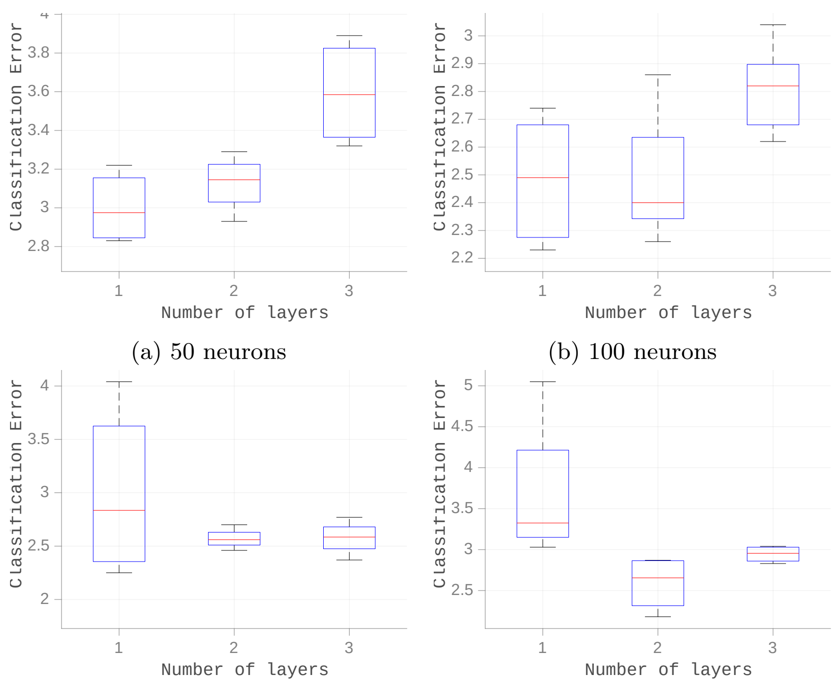
2.4.1 Initialization

As our nonlinear link function we chose a logistic function, a hyperbolic tangent was also considered but entailed problems with derivatives. Another sigmoid function, defined as $f(x) = 1.7159 \tanh(\frac{2}{3}x)$ suggested in [14] and supposedly better than a regular logistic function was also tried but did not give satisfactory results. Since we had to classify 10 classes here, the output and loss functions chosen were the softmax and negative log-likelihood, respectively.
Regarding weight initialization, it is important to note that one should avoid setting initial weights to a constant value as it will make the neurons learn the same thing. Indeed if they compute the same output, then the gradient computed will also be the same during backpropagation and the updates made to the weights will be the same [15]. Thus it is important to "break symmetry" by initializing the weights to small random values that will compute distinct updates.

At first, we sampled weights from the distribution $\mathcal{N}(0, \frac{1}{\sqrt{n}})$ where n is the number of inputs so that each neuron has variance 1.
We experimented with another sampling method called *normalized initialization* [16] defined as $\mathbf{W}_j \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \right]$ where n_j is the number of inputs for layer j. We remarked that this method led to a faster convergence for training thus we kept it for our analysis.

2.4.2 Number of layers

To determine what the optimal number of layers was we tried several layouts. There's no exact methodology to determine the optimal layout thus it must be derived heuristically by either pruning i.e. start with a large network, then reduce the layers and hidden units while keeping track of the validation set performance or growing i.e. start with a small network, then add units and layers. Here our problem is not linearly separable so obviously we must use at least one layer of hidden units. If we use more layers than required then we are more likely to overfit and it will also increase the running time because of the more complex structure. This can eventually be damped by using dropout.
To choose the optimal number of layers we tried several layer sizes and monitored the performance for various numbers of layers, for each layer size. Results are displayed on figure 2.3



(a) 50 neurons (b) 100 neurons

(c) 150 neurons (d) 200 neurons

Figure 2.3: Results for the number of layers

As we can observe results depend heavily on the number of neurons involved, for small number of neurons, 1 layer seems to be better while for more than 100 neurons using 2 layers is better. Those results should be taken with caution as we didn't try for a lot of different numbers of hidden units and different sizes for each layer although it is advocated in [17] to use the same size for all layers as its works generally better than using a decreasing or increasing size. For the final training 2 layers were preferred as it resulted in the best absolute error.

2.4.3 Number of hidden units

Choosing the best number of hidden units is a complex task as it can depend on many problem specific criteria such as the number of input and output units, the number of training cases, the complexity of the function to be learned or the type of the link function. A good but very crude rule of thumb is that the size of hidden layer should be between the number of inputs and the number of outputs [18]. If you don't have enough hidden units, you will get high training error and high generalization error due to underfitting and high statistical bias. With too many hidden units, you're prone to overfit and get low training error but high generalization error and variance as well as a longer running time.

Given 2 layers, we tried several numbers of hidden units. As we can see from figure 2.4, one should avoid using less than 100 hidden units per layer. For values between 100 and 250, the difference in accuracy is not huge. Variance is bigger for 250 units but this might be due to randomness. In terms of accuracy, 300 units seems to be on average the optimal value.

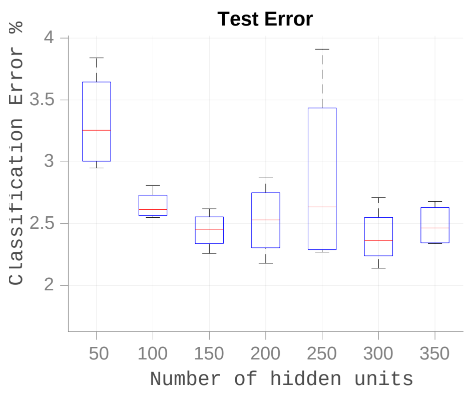
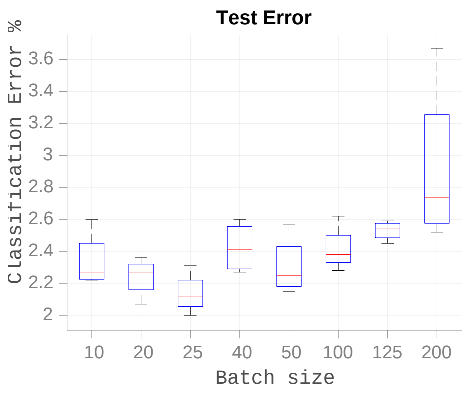


Figure 2.4: Results for the number of hidden units

2.4.4 Batch size

The batch size determines how many examples you look at before computing gradients and making a weight update. The choice of batch size pretty much boils down to the computational time. Here's a quick example for better understanding, assume we use the full training set of 60,000 observations. If we set the batch size to 200, we will train the neural network iteratively on 200 samples at a time, while making the matrix multiplications less computationally intensive, we will have to repeat the process $\frac{60000}{200} = 300$ times for just one sweep through the data, also called an epoch (see 2.4.7). Moreover, we will use several epochs to reach convergence. As `Matlab` is more efficient at computing bigger matrix multiplications but less of them, for our implementation the running time decreases as the batch size increases.



Batch size	Running time (s)
10	662
20	391
25	345
40	237
50	209
100	141
125	135
200	129

Figure 2.5 & Table 2.1: Batch size results

Figure 2.5 shows the classification errors we obtained for different values of the batch size. One observation worth mentioning is that for batches of more than 200 samples, the performance of the model drastically declined. In terms of classification error, 25 seems to be the optimal number with 2.18%(±0.12).
Table 2.1 shows the running time of training with 15 epochs, 2 layers and 300 hidden units each for one split in seconds. As we can observe the running time is really important for sizes below 100. Given the fact that the relative difference in classification error between 25 and 100 is $\approx 0.2\%$ while the gain in running time is decreased by a factor of 4.7, we estimated that a batch size of 100 was the optimal tradeoff between accuracy and running time.

2.4.5 Learning rate

This is maybe the most important of all the parameters to tune as it is at the core part of the computations of weight updates. If we choose a learning rate too large, the loss will increase and we will never reach convergence, if the learning rate is too small, the model doesn't learn fast enough and convergence will be slow. Usually the optimal learning rate is the largest one that does not cause divergence of the training criterion [19] (the loss function here). 10 log-spaced values between 10^{-3} and 10^{-1} were tried, results are displayed on figure 2.6.

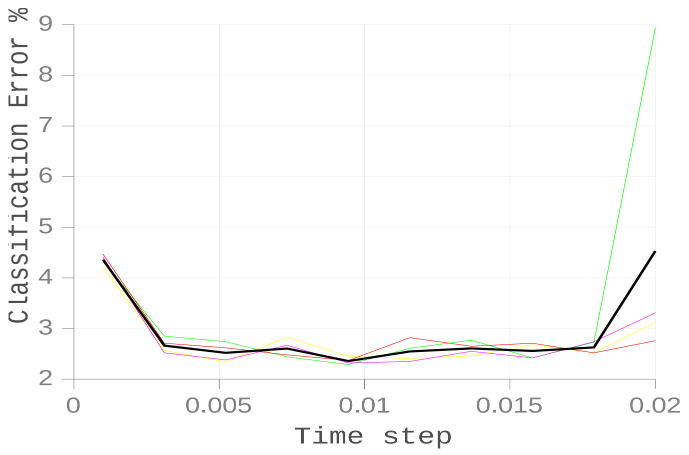


Figure 2.6: Results for the time step

Each color curve represents the evolution of the classification error for a split as a function of the time step. The thick black line represents the mean evolution of all splits. As we can observe there is not much difference for values below $\alpha = 0.018$, on the other hand we diverge for values above 0.018. Applying the rule stated above we chose $\alpha = 0.015$ for our learning rate.

2.4.6 Momentum

As we said in section 1.5.2, adding a momentum term helps the performance by preventing the system from converging to a local minimum or a saddle point. A high momentum parameter can also help to increase the speed of convergence of the system. However, setting the momentum parameter too high can create a risk of overshooting the minimum, which can cause the system to become unstable. A momentum coefficient that is too low cannot reliably avoid local minima, and can also slow down the training of the system [20].

Results we obtained are displayed on figure 2.7, we tried 9 different values with $\mu \in [0, 0.8]$ with a step of 0.1. Each colored curve represents the evolution of the classification error as a function of the momentum for a split of the data while the thick black line represents the mean over all splits. Note that values above 0.8 were not considered as it was the critical point that made the system unstable. We can observe that $\mu = 0.4, 0.5$ gave the best results overall, mainly because the blue and red splits obtained a very low error for these two values. We ran the whole process a second time (not shown here) and obtained the same conclusion. A momentum term of $\mu = 0.4$ was finally chosen.

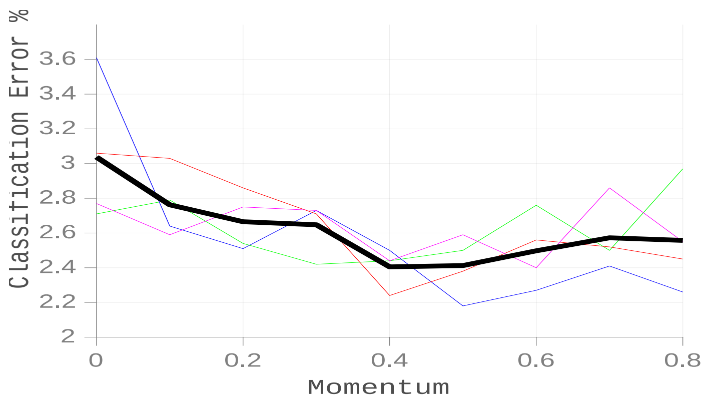


Figure 2.7: Results for the momentum

2.4.7 Number of epochs

One epoch consists of one full training cycle on the training set. The criterion which will be used to assess here the optimal number of epochs is the evolution of the accuracy on the validation set. If we don't do enough sweeps through the data, the model will underfit and we will end up with high bias and high generalization error. Conversely, if we use too many epochs, at one point the learning process will saturate and we will overfit, yielding high generalization error and high variance. Figure 2.8 shows the evolution of both training and validation accuracy (percentage of correctly classified digits on each respective set). We can see that both accuracies increase until epoch 15 where the validation accuracy reaches its limit. The gap between training and validation accuracy is very good (less than 2%), meaning that we don't seem to overfit. Based on this, we can conclude that the optimal number of epochs is 15.

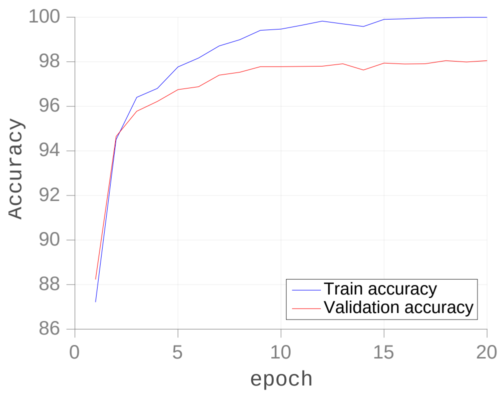


Figure 2.8: Results for the number of epochs

2.4.8 Regularization

There exists a handful of techniques to reduce the amount of overfitting in the context of neural networks. We present some of them that were considered during the analysis.

Early stopping

Perhaps one of the most efficient and easy to set up technique is *early stopping*. This works as follows: after each epoch, you monitor the validation accuracy, when it stops improving, terminate training. This simplifies a bit the tedious process of optimizing hyperparameters as you basically optimize the number of epochs "for free". As stated in [19], early stopping is an inexpensive way to avoid overfitting i.e. even if the other hyperparameters would yield to overfitting, early stopping will considerably reduce the overfitting damage that would otherwise ensue. It also means that it hides the overfitting effect of other hyperparameters, possibly obscuring the analysis that one may want to do when trying to figure out the effect of individual hyperparameters. For this reason, early stopping should be turned off when analyzing the effect of other parameters.

The tricky step in early stopping is how to define the stopping criterion. To this end, you need to accurately define what it means to "stop improving". In practice, the accuracy can jump around quite a bit, even when the overall trend is to improve. If we stop the first time the accuracy decreases then we'll almost certainly stop when there are more improvements to be had. A better rule is to terminate if the best classification accuracy doesn't improve for quite some time [21]. In our implementation, we proceeded in the following way: we stopped if the validation error increased for 3 consecutive epochs, or if the absolute difference between two consecutive iterations was less than 3×10^{-3}, passed 10 epochs. With this criterion, training our model on MNIST stopped at epoch 15 in 90% of cases (± 1 epoch), thus it seems well suited for our problem according to section 2.4.6.

Dropout

Overfitting can also be reduced by using *dropout* to prevent complex co-adaptations on the training data. On the presentation of each training case, each hidden unit is randomly omitted from the network with a probability, called the dropout fraction. Using dropout makes the model more robust as it acts as averaging the effects of a very large number of different networks [22].

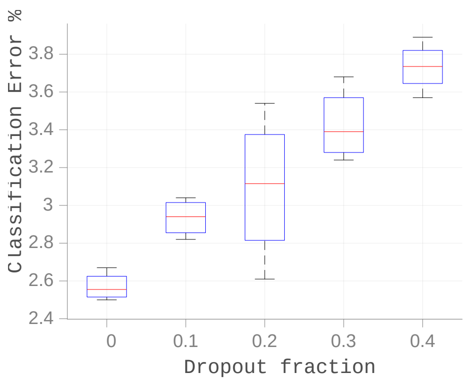


Figure 2.9: Results for the dropout

As we can see from figure 2.9, adding dropout negatively affects the accuracy of the model. Previous results also suggested that we don't overfit thus we concluded to not add dropout for the final results.

L2 regularization

The idea of *L2 regularization* or *weight decay* is to add an extra term to the cost function, called the *regularization* or *penalization* term. The resulting new cost function is of the form :

$$E_{L2}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2} \sum_{w} w^2$$

Where $E(\mathbf{w})$ is the original cost. The penalty term causes the weights to converge to smaller absolute values than they otherwise would. Large weights can hurt generalization in two different ways. Excessively large weights leading to hidden units can cause the output function to be too rough, possibly with near discontinuities. Excessively large weights leading to output units can cause wild outputs far beyond the range of the data if the output activation function is not bounded to the same range as the data [20]. Thus L2 regularization has the effect to "shrink" the weights to smaller values. Note that some additional considerations have to be taken while using this type of regularization, it's important to standardize the inputs, adjust the gradient accordingly since you change the cost function and you also have to omit the bias from the penalty term.
We tried several log-spaced values of λ from 10^{-3} to 10^{-1}. As expected, adding the penalty term resulted in a decrease of accuracy thus we omitted it.

2.5 Final results

Now that we have optimized each of our hyperparameters, we are ready to train the model on the whole set and see how well it performs. For each trial, we randomly shuffled the observations of the data and trained on the whole training set, the error rate was evaluated on the whole testing set. Results are summarized in table 2.2.

Trial	Test error rate (%)
1	1.82
2	1.86
3	1.65
4	1.94
5	1.84
average	1.82 ($\pm$0.11)

Table 2.2: Final results

Chapter 3

Conclusions

In this project, a neural network was implemented and tested on the MNIST dataset. We first reviewed the theory related to neural networks, why using nonlinear classification and what are the benefits from using it. We saw the structure of neural networks and the main advantage of using them: the backpropagation algorithm. We also reviewed one of the most famous optimization technique, the gradient descent algorithm.
This project was not entirely theoretically focused but also practically oriented as we optimized our model on a hands-on dataset: the MNIST database of handwritten digits. The data was first preprocessed and we conducted a first analysis by testing the data on a baseline model which was the softmax regression. We then tuned the hyperparameters of our neural network to obtain an optimal error rate of 1.82%.
The main goal of the project was to first implement a learning framework and the neural network, then to conduct a statistically robust and significant analysis.

Bibliography

[1] "Statistical classification." `https://en.wikipedia.org/wiki/Statistical_classification`.

[2] C. M. Bishop, *Pattern Recognition and Machine Learning.* Springer, 2006.

[3] Y. W. Jianqing Fan, Yingying Fan, "High-dimensional classification," *Ann. Statis.*, vol. 36, no. 6, 2008.

[4] J. Weston, "Suport vector machine: Tutorial,"

[5] "Artificial neural network." `https://en.wikipedia.org/wiki/Artificial_neural_network`.

[6] M. Seeger, "Pattern classification and machine learning," 2013.

[7] "Genevieve orr. momentum and learning rate adaptation.." `https://www.willamette.edu/~gorr/classes/cs449/momrate.html`.

[8] J. T. X. Fang, H. Luo, "Structural damage detection using neural network with learning rate improvement," tech. rep., Department of Mechanical Engineering, The University of Connecticut, 2005.

[9] "Mnist databse website." `http://yann.lecun.com/exdb/mnist/`.

[10] C. M. Bishop, *Neural Networks for Pattern Recognition.* Oxfor University Press, 1996.

[11] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, pp. 141–142, 2012.

[12] P. Domingos, "A few useful things to know about machine learning," tech. rep., Department of Computer Science and Engineering, University of Washington, 2012.

[13] A. F. G. M. Bashiri, "Tuning the parameters of an artificial neural network using central composite design and genetic algorithm," *Scientia Iranica*, vol. 18, pp. 1600–1608, December 2011.

[14] G. B. O. K.-R. M. Yann LeCun, Leon Bottou, "Efficient backprop," tech. rep., 1998.

[15] "Stanford cs class: Convolutional neural networks for visual recognition." `http://cs231n.github.io`.

[16] Y. B. Xavier Glorot, "Understanding the difficulty of training deep feedforward neural networks," tech. rep., Université de Montréal, 2010.

[17] H. Larochelle, *Étude de techniques d'apprentissage non-supervisé pour l'amélioration de l'entraînement supervisé de modèles connexionnistes.* PhD thesis, Université de Montréal, 2012.

[18] A. Blum, *Neural networks in C++: an object-oriented framework for building connectionist systems.* Association for Computing Machinery, 1992.

[19] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," *CoRR*, vol. abs/1206.5533, 2012.

[20] W. S. Sarle, "Neural network faq." `ftp://ftp.sas.com/pub/neural/FAQ.html`, 2002.

[21] M. Nielsen, "Improving the way neural networks learn," in *Neural Networks and Deep Learning*, ch. 3, Determination Press, 2015.

[22] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.