Multi-Layer Perceptrons

Ioan Cristian Schuszter - **ioan.schuszter94@e-uvt.ro**

Data Mining Report
West University of Timișoara, Romania
June, 2017

Abstract

In the current context of the ever-growing popularity of the domain of Machine Learning, one of the most well-known and appreciated algorithms of the past years have been neural networks, drawing their popularity especially from deep networks, due to amazing feats achieved by them, such as the AlphaGo AI [1]. These algorithms are not new in any sense, with their origins being traced back to the 50's and 60's, but were refined along the years. Computing power has also increased tremendously, leading to practical applications of such algorithms everywhere.

This paper proposes studying the characteristics of multilayer perceptrons, a variant of neural networks which has been proven to solve really complex problems. Section 1 will explore the background of neural networks, whilst section 2 will present how the multilayer perceptron works and different algorithm design choices that can be made. Section 3 presents our own novel implementation of a multilayer perceptron, using Scala, and the results obtained by applying the algorithm to well-known problems. In the last section, we describe what future work could be done to improve our current implementation and some conclusions.

Contents

1 Introduction

Machine learning is a branch of computer science dealing with providing computers with a mechanism of learning without explicitly giving it a program specification. It is used in the field of data mining in order to devise complex models and algorithms hat lend themselves to the use of prediction; this is known as predictive analytics. A wide range of uses exist for machine learning, from email filtering, detection of network intrusion, to optical character recognition and (more widely) computer vision. It has strong ties with the field of mathematical optimization, as we'll see when we discusss the algorithms used within this paper. In this paper we explicitly sutdy just one such algorithm, called an artificial neural network.

These kinds of algorithms are called supervised learning ones.

1.1 Supervised learning

Since machine learning tries to infer a function from data, supervised learning refers to the fact that data is provided using labeled training data. The opposite, unsupervised learning, refers to data being unlabeled and the labels being infered by the algorithm used. A supervised learning algorithm analyzes the training data and produces a function, that can then be used to map new inputs to their respective classes. This requires the algorithm to generalize from training data to unseen situations in a "reasonable way".

The algorithm works in the following way: Given a set on N training examples in the form $\{(x_1, y_1), ..., (x_N, y_N)\}$ such that x_i is the feature vector of the ith example and y_i is its label, the learning algorithm tries to find a function $g : X \to Y$ where X is the input space and Y is the output space. A scoring function $f : X \times Y \to R$ is often used such that g is defined as returning the y value that gives the highest score:

$$g(x) = \arg\max_y f(x, y)$$

There are two basic approaches to choosing f or g: empirical risk minimization and structural risk minimization. Empirical risk minimization seeks the function that best fits the training data. Structural risk minimize includes a penalty function that controls the bias/variance tradeoff.

In both cases, it is assumed that the training set consists of a sample of independent and identically distributed pairs,(x_i, y_i).

When G contains many candidate functions or the training set is not sufficiently large, empirical risk minimization leads to high variance and poor generalization. The learning algorithm is able to memorize the training examples without generalizing well. This is called overfitting.

1.2 Artificial Neural Networks

Artificial neural networks (also known as connectionist systems) combine simple processing elements (called neurons) into an ensemble capable of delivering complex approximations of functions or classification, being inspired by the biological neural systems that constitute the human brain. Such systems learn (progressively improve their performance for a given task) to do tasks by looking at learning examples and being provided no explicit task programming needs.

For example, taking the task of image recognition, one of the simplest ones: handwritten digit recognition. If one were to explicitely program recognition in code, a lot of simple intuitions would work, such as "a 9 has a loop at the top and a vertical stroke in the bottom right". But handwriting is different for everyone, and soon enough the programmer would be coding around a myriad of caveats and special cases, rendering the problem from seemingly simple to almost impossible. [2].

1.2.1 Perceptrons

The perceptron is in some sense the precursor of artificial neural networks. It's a supervised learning algorithm, using a linear function to perform binary classifications. The algorithm allows for online learning, processing one element at a time. It all began with the custom hardware created at the Cornell Aeronautical Laboratory by Frank rosenblatt. The perceptron was a machine, instead of a program, (called the Mark 1 perceptron) being one of the earliest model of neural networks created. It had around 400 photocells randomly connected to the neurons. The weights were encoded using potentiometers.

The perceptron was highly acclaimed by the press at that time, but it was quickly proven that perceptrons can't correctly classify many classes of patterns. This caused the field to stagnate for several years, until it was recognized that feedforward neural networks with two or more layers had far greater processing power. The perceptrons are only capable of learning linearly separable patterns. It 1969, the book "Perceptrons" by Minsky and Seymour Papert showed that this kind of neural network can't learn a XOR function.

1.2.2 Modern Approaches

The perceptron's failure created a long gap in the progress of artificial intelligence, but about ten years after Minsky's book, it was proven that more complex, nonlinear functions, can be learned using a feedforward neural network approach. Feed forward neural networks are the simplest form of modern-day neural networks, mapping a set of input data to a set of appropriate outputs. They are called feedforward because they contain no cycles, instead being a directed graph with each layer being fully connected to the next one. (see Fig. 1)

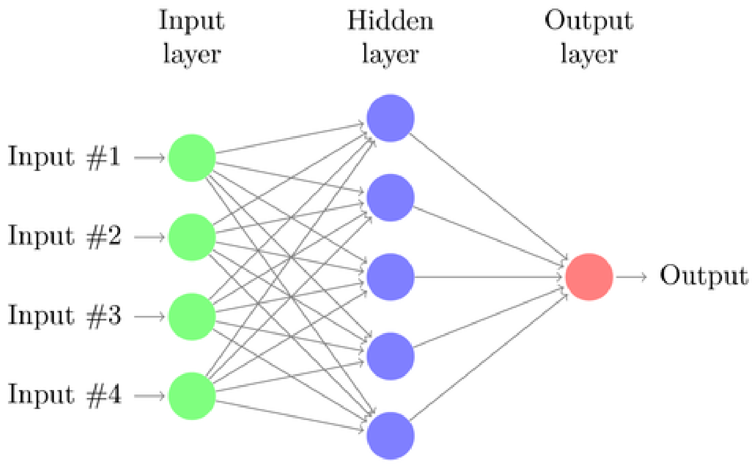

Figure 1: A simple one hidden-layer feedforward neural network (multilayer perceptron)

[3] states that, due to the universal approximation theorem for neural nets, every function can be approximated by a multi-layer perceptron with just one hidden layer. The result holds for a large number of activation functions. There are a variety of learning techniques for multi-layer perceptrons, the most popular being the backpropagation algorithm. In general, however, many tweaks need to be employed in order to get a network to train well on some input data, such that the model does not overfit and fail to capture the underlying true statistical process that generates the data.

Adjusting the weights is done by applying a general method for non-linear optimization named gradient descent, into which we'll go into more detail in the following chapter. The error is checked and the weights are adjusted in such a way that the error becomes smaller at the next iteration. As classification is a particular case of regression, when the response variable is categorical, MLPs are good classifier algorithms, as it will be shown in the next few sections.

1.2.3 State of the Art - Deep Learning, GPU processing and CNNs

More recently, artificial neural networks have seen a great increase in popularity, due to their successful applications in real-life problems. Deep learning is an offspring of this new wave of AI progress, using neural networks that contain more than one hidden layer. Deep learning branches into more subdomains, such as deep belief networks. There are new challenges to training these kinds of networks. DNNs are prone to overfitting because of the added layers of abstraction, which allow them to model rare dependencies in the training data.

CNNs (convolutional neural networks) use a variation of multilayer perceptrons designed to require minimal preprocessing. It is also known as shift invariant or space invariant artificial neural network (SIANN), based on its shared-weights architecture and translation invariance characteristics.

GPU programming can and was applied successfully in order to train these networks much faster. TensorFlow is an open source software library for machine learning across a range of tasks, and developed by Google to meet their needs for systems capable of building and training neural networks to detect and decipher patterns and correlations. It has recently been upgraded to allow training of neural networks on the GPU. These kinds of GPU procesing operations help speed-up training quite a bit, because training a neural network is not much more than huge matrix multiplication.

2 Problem description and Algorithm Details

As previously noted, the purpose of supervised learning and neural networks is to map an input to outputs correctly for instances that it has never seen before. This section details the construction and training process of the multi-layer perceptron. Figure 1 has shown the architecture of such a multi-layer perceptron.

2.1 Architecture and notation

Let there exist an input matrix X which contains the features that represent one instance of the training data to be fed into the network. Let also $w_{i,j}$, a weight matrix for each of the synapses that lie between the layers. In the case of the simple 3-layer architecture we use the following notation in the subsequent cases:

- 1 input layer, with a neuron for each feature in the input set
- 1 hidden layer, with a variable amount of neurons that can be set when training the network
- 1 output layer, the number of neurons in this layer is equivalent to the number of classes that we wish to classify. In the simplest case, there's only one neuron in the output layer, turning the problem of classification into a regression problem.

The follwing notation will be used throughout the explanations of the algorithm:

- X - input data, each row is an element, dimensions $numExamples \times inputLayerSize$
- y - labels for the input data, dimensions $numExamples \times outputLayerSize$
- $W^{(1)}$ - layer 1 weights (between the input layer and the hidden layer), dimensions $inputLayerSize \times hiddenLayerSize$

- $W^{(2)}$ - layer 2 weights (between the hidden layer and the output layer), dimensions $hiddenLayerSize \times outputLayerSize$
- $z^{(2)}$ - activation of layer 2
- $a^{(2)}$ - activity of layer 2 (after applying the activation function on $z^{(2)}$)
- $z^{(3)}$ - activation of layer 3
- f - activation function

Activation, as previously mentioned, refers to applying a function f to each element of the matrix on some layer, yielding the activity matrix. The most common activation function, the one that we use in our implementation in the next section, as well, is:

$$f(x) = \frac{1}{1 + e^{-x}}$$

The above function is also known as the sigmoid logistic function. An advantage to this function used as an activation function is the fact that the derivative is easily computed, as needed in the future subsections of this section:

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}$$

2.2 Forward propagation

The network operates in two modes, in training and prediction mode. [4] For the training of the network, we use two datasets, a trainning one and a test set (that we want to predict). The forward propagation is used in both modes. In the training mode it is used in odrder to verify the current error of prediction for the test set, whilst in the prediction mode (post-training), the network uses the forward propagation of new inputs to get predictions for their classes (or values, in the case of regression).

The forward propagation takes place at each layer of the network and yields the activity of the last layer (the output layer). In our 3-layer example, the following operations are done:

1. The layer 2 activations are computed, by multiplying the input matrix with the weights matrix:

$$z^{(2)} = XW^{(1)}$$

2. The layer is activated using the activation function:

$$a^{(2)} = f(z^{(2)})$$

3. The activity is propagated to the final layer and the activity of the final layer is returned as the output of the forward propagation y':

$$\begin{aligned} z^{(2)} &= a^{(2)}W^{(2)} \\ y' &= f(z^{(3)}) \end{aligned} \tag{1}$$

If more than three layers are to be involved, the part detailed at step **3** would be repeated for each layer in the hidden units.

2.3 The Backpropagation Algorithm

Now that forward propagation is defined, one could theoretically try all the combinations of weights in the weight matrices (the main changing parts in the network) in order to get a good result. However, since there are more possible combinations than stars in the universe (for a small number of weights, like 9), a more clever approach needs to be needed.

We define a cost function J in our algorithm, which for this paper's purpose will be the sum of squared errors between the predictions and the actual labels of the training data:

$$J = \sum \frac{1}{2}(y - y')^2$$

We wish for our network to minimize this cost (basically, the errors of approximation), introducing a minimization problem. The way that backpropagation works, is by computing the directions in which the objective function decreases, for each of the weight matrices. These weight matrices then get adjusted after each step of the training of the algorithm, changing so as to increase the accuracy of the model by being driven in the direction of the minimum. A learning parameter can be used to decide how fast to move in the direction of the minimum, by multiplying it with the values of the gradients.

The basic idea is to compute the partial derivatives for each of the weights in the network:

$$\frac{\delta J}{\delta W^{(i)}}, \forall i \in 1..n$$

These derivatives are computed by "backpropagating" the error of the predictions, hence the name of the algorithm. The first derivative (the one for the output unit) is simply obtained as:

$$\frac{\delta J}{\delta y'} = \sum \frac{1}{2}\frac{\delta(y - y')^2}{\delta y} = -(y - y')$$

If the layer isn't the output one, however, the chain rule must be applied in order to obtain the gradients for each weight matrix.

$$\frac{\delta J}{\delta W^{(2)}} = -(y - y')\frac{\delta y'}{\delta W^{(2)}} \tag{2}$$

We then need to apply the chain rule in order to get the last derivative, so we get:

$$\frac{\delta J}{\delta W^{(2)}} = -(y - y')\frac{\delta y'}{\delta z^{(3)}}\frac{\delta z^{(3)}}{\delta W^{(2)}} \tag{3}$$

Since we have the derivative of the activation function. We can replace $\frac{\delta y'}{\delta z^{(3)}}$ by $f'(z^{(3)})$:

$$\frac{\delta J}{\delta W^{(2)}} = -(y - y')f'(z^{(3)})\frac{\delta z^{(3)}}{\delta W^{(2)}} \tag{4}$$

For each of the synapses, $z^{(3)}$ is just the activation of a on that synapse, meaning that we can finally obtain the last derivative. The activities in the previous layer ($a^{(2)}$) in the case of our second layer, need to be multiplied by the value obtained above, which will be called delta, as it specifies how much the weights need to be adjusted. The next system fills in the gaps for obtaining the gradient for the last weights matrix:

$$\begin{aligned} \frac{\delta J}{\delta W^{(2)}} &= (a^{(2)})^T \delta^{(3)} \\ \delta^{(3)} &= -(y - y')f'(z^{(3)}) \end{aligned} \tag{5}$$

The next steps apply to all of the hidden layers, even if they are 1 or many. The previous delta that was obtained needs to be multiplied with the weights of the next layer transposed and then with the derivative of the the current layer's activation. In the case where this is the first layer, the input data transposed is multiplied with the neurons.

$$\frac{\delta J}{\delta W^{(1)}} = X^T \delta^{(3)} (W^{(2)})^T f'(z^{(2)}) \tag{6}$$

or:

$$\begin{aligned} \frac{\delta J}{\delta W^{(1)}} &= X^T \delta^{(2)} \\ \delta^{(2)} &= \delta^{(3)} (W^{(2)})^T f'(z^{(2)}) \end{aligned} \tag{7}$$

2.4 Training and generalization

There are multiple ways of training the network. The simplest method is batch training, in which the entire training dataset is passed through the algorithm and only then the weights are adjusted. The training process goes through the following steps:

- forward propagate the training data
- compute error and backpropagate
- adjust the weights
- restart the training process if the cost function hasn't decreased below a certain threshold or a given number of steps has been reached

After the backpropagation step, all of the weights are modified simultaneously, scaled by the learning rate of the algorithm:

$$\begin{aligned} W^{(1)} &= W^{(1)} - \alpha * \frac{\delta J}{\delta W^{(1)}} \\ &. \\ &. \\ &. \\ W^{(n)} &= W^{(n)} - \alpha * \frac{\delta J}{\delta W^{(n)}} \end{aligned} \tag{8}$$

Overfitting could be a problem. The test dataset should be tested with the cost function. If the cost becomes way bigger than before, it may be a sign that the network is experiencing overfitting, meaning that the generalization capacity of the network for real-life data may be gone. The network might memorize the inputs and not perform well at all for other examples. (new ones).

No good method [4] exists for the selection of an architecture, this being an experimental process and subject to research in order to find a good rule of thumb.

2.5 Improvements

A few improvements could be explored in order to make the algorithm work better than the "vanilla one".

The first such improvement that we take into consideration is the addition of a regularization parameter λ to our algorithm implementation. Regularization helps prevent overfitting in statistical models. By adding the norms of the weight matrices to the cost function this can be achieved, the function becoming:

$$J = \sum \frac{1}{2}(y - y')^2 + \lambda \frac{1}{2} \sum (W^{(i)})^2$$

The lambda param should be taken as a really small number, the regularization acting as a kind of perturbation to the data. The lambda param should also be added to the derivative of the cost function, essentially affecting the gradient computations. This is done by adding the weight matrix scaled by lambda to each of the gradients:

$$\frac{\delta J}{\delta W^{(i)}} = \frac{\delta J}{\delta W^{(i)}} + \lambda * W^{(i)}, \forall i$$

3 Implementation and Results

Our implementation is done in the Scala programming language. We use an external library for ease of use in mathematical operations. This library is called **breeze** and is a mature linear algebra toolkit that also provides graphing capabilities with which we can see the performance of the network.

The main features that are implemented in this project are:

- a class called **SequentialNeuralNet** that is highly configurable from the algorithm's point of view
- plotting utility classes, used to display the trained network and different measures read by the network
- IDX format reading-writing capabilities, for the MNIST dataset files.

The discussion features the testing of the implemented neural network on two different data sets, a simple one and a much more advanced one, namely the XOR dataset and the MNIST handwritten digit dataset (the full one).

3.1 Implementation

The main configuration parameters for the multilayer perceptron implementation are specified in the constructor of the **SequentialNeuralNet** class, as well as in the parameters passed to the training method itself. The implementation doesn't have a limit of how many layers the neural network might have. One specifies a list of numbers, detailing the amount of layers.

For example: $hiddenLayers = List(3, 10)$ means that the first hidden layer has 3 neurons, whilst the second hidden layer has 10 neurons.

An example of the constructor call:

```
SequentialNeuralNet(inputLayerSize: Int,
                    outputLayerSize: Int = 1,
                    hiddenLayers: List[Int] = List(3),
                    alpha: Double = 1,
                    lambda: Double = 0.0001)
```

The class implements a few extra things apart from standard multi-layer perceptron:

- regularization, by means of the lambda parameter, which adds or removes
- configurable alpha parameter, which allows for different step sizes to be tested on a dataset

The training method's signature can be seen below:

```
def train(trainX: DenseMatrix[Double],
          trainY: DenseMatrix[Double],
          testX: DenseMatrix[Double],
          testY: DenseMatrix[Double],
          its: Int = 10000,
          miniBatchSize: Int = 0,
          debug: Boolean = false): (Array[Double], Array[Double])
```

The parameters which are mandatory are the training dataset (with X for data and y for labels), whilst the rest of them are for tweaking and testing. The parameters are explained below:

- **its** - the number of max. iterations to perform on the algorithm. The algorithm does not implement adaptive stopping, as would be the case, so it always goes to "its"
- **miniBatchSize** - part of the minibatch implementation of the algorithm, to be discussed below, 0 means disabled
- **debug** - whether or not to print out debug details in the algorithm

The miniBatchSize parameter is in fact very important, especially when training on data that is highly dimensional. The parameter specifies how many rows to be tested at once and fed into the backpropagation algorithm. If the number of entries is large and the batch size is small, many small adjustments will be done to the weights at each epoch, the contrary being to do huge matrix multiplications for each epoch. The findings in [5] denote that minibatch SGD is an excellent method for providing great results in the cases where we have large datasets.

The minibatch approach to sampling the dataset creates a very powerful algorithm on the MNIST dataset with less than 30 iterations, while the non-minibatch approach would take many more epochs. A more powerful, distributed version, is also described in [5].

3.2 Results

3.2.1 XOR data

The most classic example of neural networks learning to predict non-linear functions is the XOR method. We created a network with the 2 configurations:

1. A hidden layer with 3 neurons, trained for 1000 epoch, $\alpha = 15$ and $\lambda = 0.0001$. No miniBatch processing was added to the mix. See figure 2 for performance:

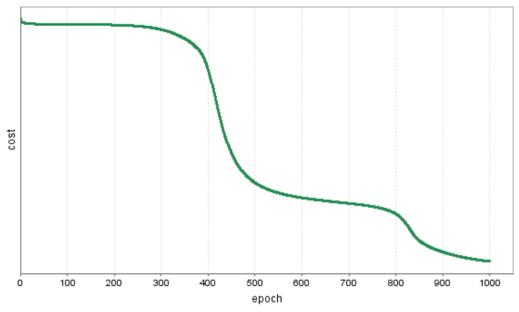

Figure 2: One hidden layer XOR network

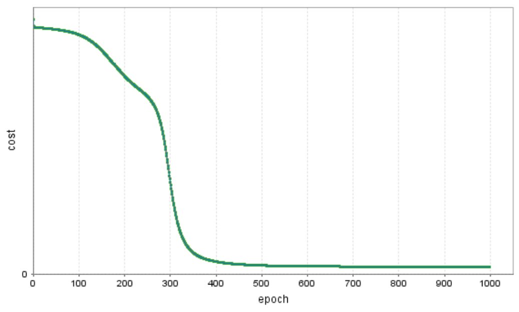

Figure 3: Two hidden layer XOR network

2. Two hidden layers, with 3 neurons each, $\alpha = 10$. The rest of the parameters were like above. See figure 3 for performance:

As it can be seen, the second layer converges a few hundred epochs faster than the first one. This is most likely due to the complexity of the model built with two layers. However, the overfitting is good in this case as we need to approximate the function. The two-layer approach approximates it better.

The printout for the first layer's run is:

```
predicted:
0.15310856969963493
0.8054789358331848
0.8438715686957257
0.1169492733962203
expected:
0.0
1.0
1.0
0.0
```

While the printout of the second network's run is:

```
predicted:
0.05312360105118474
0.9333297096356272
0.9162709625020431
0.08069707649259043
expected:
0.0
1.0
1.0
0.0
```

3.2.2 MNIST dataset

The MNIST dataset contains 60k training images and 10k test images. Some special code had to be written in order to parse the IDX3 format required by the MNIST dataset. The images in the set are 28x28 character handwritten digits. Some of them are "uglier", in the sense that the labeling of those digits could be problematic even for a human being. A few runs on subsets of the data have been made, with the following parameters and results.

A parameter of $\alpha = 3.0$ has been used for all of the runs. 30 iterations have been taken each time, you can see the following figures explaining what the network's training yielded. A miniBatchSize of 10 was chosen for the classification.

The network architecture is the following:

- input layer = 28x28 = 764 features, one for each grayscaled pixel of the image
- one hidden layer = 15 neurons
- one output layer = 10 neurons

The first run was done on 30k images and 2k test files. The end of the run shows some overfitting starting to take place:

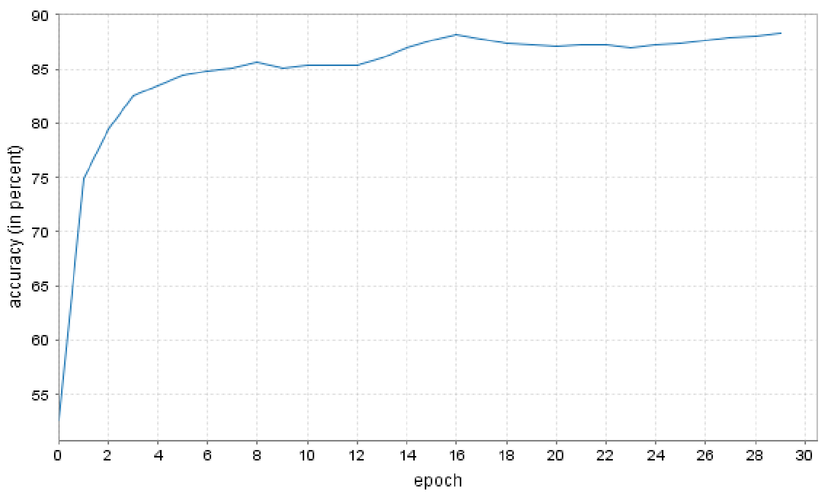

Figure 4: Accuracy / epoch on MNIST 30k

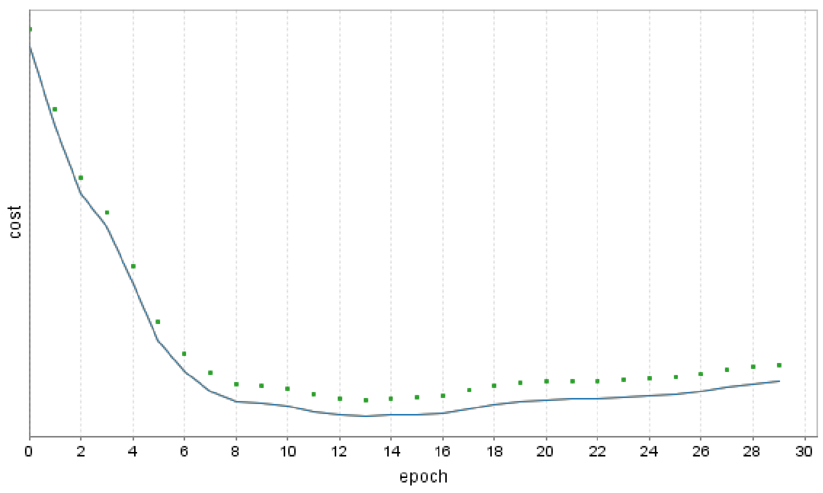

Figure 5: Cost function / epoch on MNIST 30k

The second run was done on 60k images and 5k test files. Unsurprisingly, the accuracy grew a bit more, and a little bit of parameter tuning might bring the network to the levels of the most advanced networks today. Below, the figures present the data.

The last figure show what the network thought some characters were and what the characters actually were, taken from a subset of 10 images.

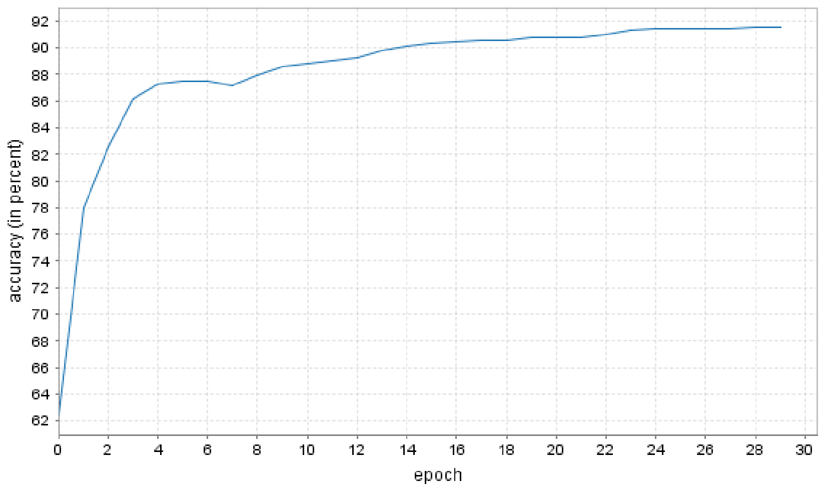

Figure 6: Accuracy / epoch on MNIST 60k

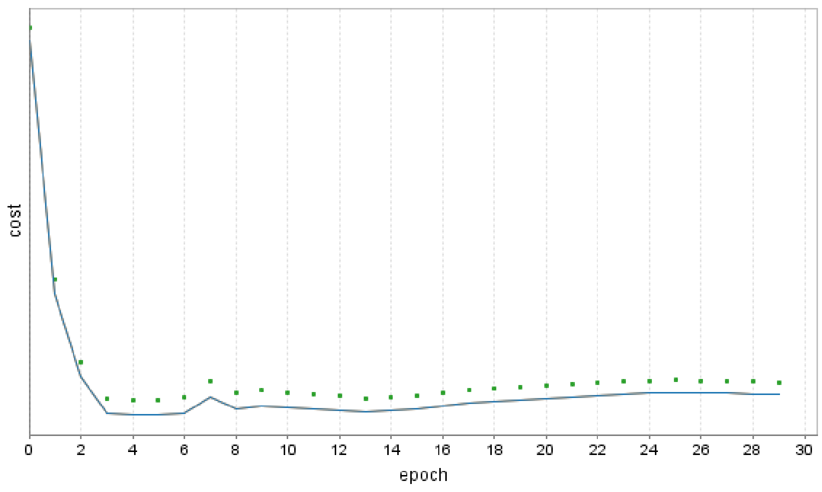

Figure 7: Cost function / epoch on MNIST 60k

4 Conclusions and Future Work

In conclusion, many variants of neural network implementations and optimizations exist. While seductive in their power, a grain of salt must be taken in order to fully be able to use these kinds of machine learning algorithms. The experiments that were undertaken in this project show the fact that there's no one set of parameters to rule all and that extensive work must be done to ensure both that overfitting doesn't take place and the fact that there's no recipe for the number of layers or the number iterations needed in order to reach convergence.

Regarding future work, a prototypal version of a distributed neural network implementation was born, bearing similarities to the one provided by Apache Spark in their implementation. Batch processing of the neural network would've been impossible on my i5 machine, with training taking up a huge amount of time in order to be effective on the 60k MNIST dataset. The dataset and others would benefit from a distributed implementation, allowing matrix multiplications to be done on many machines at once.

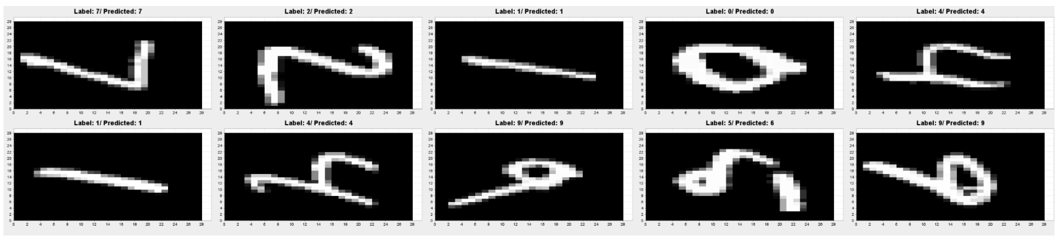

Figure 8: Labels / predictions on a subset of digits taken from the MNIST 60k test

References

[1] David Silver, Aja Huang et al., "Mastering the game of Go with deep neural networks and tree search", Nature 529, 2016.

[2] Michael Nielsen, "Neural Networks and Deep Learning", Chapter 1, 2017.

[3] Hornik, K., Stinchcombe, M. & White, H. (1989) “Multilayer feed forward networks are universal approximators,” Neural Networks, Vol. 2, no. 5, pp. 359-366, Jul. 1989.

[4] Daniel Svozil, Vladimir Kvasnicka, Jiri Pospichal, "Introduction to multi-layer feed-forward neural networks", Chemometrics and Intelligent Laboratory Systems 39, pp. 43 - 62, 1997.

[5] Mu Li, Tong Zhang, Yuqiang Chen and Alexander J. Smola, "Efficient Mini-batch Training for Stochastic Optimization", Carnegie Mellon University, Baidu. Inc. and Google