Optimization for Big Data
Term Project Report

Recognizing Handwritten Digits *by using* Artificial Neural Network

An implementation on
MNIST Data Set

Submitted by

Soner Aydın
Süleyman Topdemir

Under the guidance of
Prof. Dr. İlker Birbil

Faculty of Engineering and Natural Sciences

Sabanci University
Istanbul

Spring 2017

Recognizing Handwritten Digits
by using
Artificial Neural Network

Submitted on June 1, 2017

1 Introduction

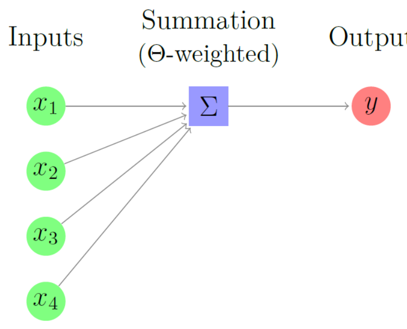
Multi-layer neural networks are powerful tools for a wide range of prediction tasks; image recognition, stock market forecasting, tumor recognition, fraud detection, just to name a few. Historically, neural networks (NN) were first developed in artificial intelligence community to imitate the working principle of actual brain neurons. Since 1980s, this method evolved into one of the most prestigious machine learning tools. With the advent of high performance computing facilities, research in NN gained a new impetus under a new form (deep learning) during 2000s. It had been shown that, multi-layer neural networks can fit any nonlinear smooth function, regardless of its orderKurkova (1998). In this project, we used a two-layer neural network to predict the labels of handwritten digits in MNIST dataset, by using different optimization algorithms, and compared their performances.

2 Structure of Artificial Neural Network

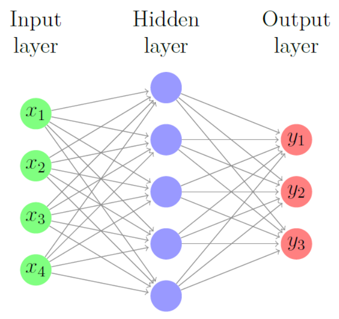
The structure and working principle of the neural networks can be understood more clearly considering a simple linear regression model. The major similarity between two concepts is that the ultimate aim is to minimize the total loss/error function value. In both cases we have a differentiable loss function to minimize.

One can construct just a linear model in linear regression, however, neural networks allow to establish nonlinear models, which can be considered more complex than linear models. A simple visualization of the linear regression as a graph is shown at Figure 1a.

The main motivation in neural network is to find an appropriate weighted model by using the training data. Let us denote the training data $T = \{(x_n, t_n) : 1 \leq n \leq N\}$. Considering such a model, $g(.)$ that satisfies $g(x_n) := t_n$; we are trying to find an approximation of $g(x)$ Stutz (2014).



(a) Linear Regression



(b) Simple Neural Network Model

Figure 1: Linear regression and neural network as graph

In the figure 1b one can see a simple feedforward neural network with one hidden layer. In this figure there are 4 input units in input layer and 3 output units in output layer. One can generalize these numbers such as C and D, respectively. Also can add $x_0 := 1$ and y_0^1 biases to input layer and hidden layer. For a feedforward neural network with one hidden layer, the output value of the i^{th} neuron of the l^{th} hidden layer, which can be denoted y_i^l, is

$$y_i^{(l)} = f\Big(\sum_{k=0}^{m^{(l-1)}} w_{ik}^{(l)} y_k^{(l-1)} \Big) \tag{1}$$

$w_{ik}^{(l)}$ denotes the weighted connection between the k^{th} unit of layer $(l-1)$ and the i^{th} unit of layer l. $m^{(l-1)}$ shows the number of $(l-1)^{th}$ hidden layer's neuron number. In a neural network with just one hidden layer there is only $m^{(1)}$ value and can be considered such $m^{(1)} := m$.

The function $f(\cdot)$ is called *activation function* and can be in different forms. One of the most common forms of activation functions is the sigmoid function.

$$\phi(z) = \frac{1}{1+e^{-z}} \tag{2}$$

Another alternative is hyperbolic tangent function which is in form:

$$\phi(z) = \frac{1-\exp(-z)}{1+\exp(-z)} \tag{3}$$

In LeCun (1998) and alternative activation function is introduced:

$$\phi(z) = 1.7159 \tanh(\frac{2}{3}z) \tag{4}$$

Error Function The sum-of-squared error function takes the form Stutz (2014)

$$E(w) = \sum_{n=1}^{N} E_n(w) = 1/2 \sum_{n=1}^{N} \sum_{k=1}^{C} (y_k(x_n, w) - t_{nk})^2 \tag{5}$$

The derivatives of error function $E(\cdot)$ with respect to W_1 and W_2, which are the weight matrices for hidden layer and output layer respectively using chain rule

$$\frac{\partial E}{\partial W_1} = -X^T[-(Y - g(g(XW_1)W_2)) \odot g(g(XW_1)W_2) \odot (1 - g(g(XW_1))W_2)W_2^T] \odot g(XW_1) \odot (1 - g(XW_1)) \tag{6}$$

$$\frac{\partial E}{\partial W_2} = -g(XW_1)^T[-(Y - g(g(XW_1)W_2)) \odot g(g(XW_1)W_2) \odot (1 - g(g(XW_1)W_2))] \tag{7}$$

Those derivations can be employed instead of back-propagation method which is basically for updating the weights backward.

3 Network Training

In the network training step, the problem we are interested in solving is:

$$\begin{aligned} \min_{w} \quad & E(w) \\ \text{s.t.} \quad & w \in \mathbf{R}^n \end{aligned} \tag{8}$$

3.1 Parameter Optimization

Algorithm 1 Steepest Descent Algorithm

```
1: Input: $w_0$
2: Output: $w^*$
3: $k \leftarrow 0$
4: repeat
5:     $d_k \leftarrow -\frac{\partial E(w)}{\partial w_k}$
6:     Choose step size $\gamma_k$
7:     $w_{k+1} \leftarrow w_k + \gamma_k d_k$
8:     $k \leftarrow k + 1$
9: until the stopping criterion is satisfied
```

The step size may be adjusted at each iteration using the *exact line search* procedure:

$$\begin{aligned} \min_{\gamma} \quad & E(w_k + \gamma d_k) \\ \text{s.t.} \quad & \gamma \geq 0 \end{aligned} \tag{9}$$

Here d_k is called search direction at w_k which is calculated such in line 5 in steepest descent algorithm. Even though the exact line search method can provide appropriate step size at each iteration, the computational cost of obtaining an optimal solution for the minimization problem which is shown at 2 is expensive, which means better results in terms of length of solution time and objective function value can be acquired via some basic or sophisticated ways. In the machine learning field the learning rate is taken as a constant because of the computational cost of finding learning rate at each iteration.

Stochastic Gradient Descent (SGD) simply does away with the expectation in the update and computes the gradient of the parameters using only one or a few training examples. Formally, $E(w)$ can be decomposed into;

$$\sum_{n=1}^{N} E_n(w) \tag{10}$$

where N is the total number of data points we have in the training process. Calculating the gradient $\nabla E(w)$ can be problematic in terms of the time spent at each iteration considering the data size and number of parameters. We can approximate on the assumption that most of the data covers similar features. Instead of using all data points, we can take a data point or a sample from the all set.

$$\nabla E_n(w_k) \approx g_k := \nabla E_i(w_k) \text{ where } i \xleftarrow{\text{random}} \{1, ..., N\} \tag{11}$$

The algorithm which contains the calculation of the gradients using just one training data point at each iteration is called stochastic gradient descent. Instead of taking one data point, a sample that can be considered a mini-batch can be taken randomly from the data set to calculate the gradients. The algorithm which is working according to this principle is called mini-batch stochastic gradient descent. These stochastic gradient descent approaches use the steepest descent algorithm structure almost completely, the only difference is the number of data points that error function covers.

Conjugate Gradient Algorithm

Algorithm 2 Nonlinear Conjugate Gradient / The Fletcher-Reeves Method

```
Input: $w_0$
Output: $w^*$
Evaluate $E_0 \leftarrow E(w_0), \nabla E_0 \leftarrow \nabla E(w_0)$
$d_0 \leftarrow -\nabla E_0$
$k \leftarrow 0$
while $\nabla E_k \neq 0$ do
    compute $\gamma_k$
    $w_{k+1} \leftarrow w_k + \gamma_k d_k$
    evaluate $\nabla E_{k+1}$
    $\beta^{fr}_{k+1} \leftarrow \frac{\nabla E^T_{k+1} \nabla E_{k+1}}{\nabla E^T_k \nabla E_k}$
    $d_{k+1} \leftarrow -\nabla E_{k+1} + \beta^{fr}_{k+1} d_k$
    $k \leftarrow k + 1$
end while
```

To find a β value at each iteration, there are several approaches. One of them is β^{fr} which is can be calculated in line 10 in algorithm 2. The β is calculated in the Polak-Ribiere Method as follows:

$$\beta^{pr}_{k+1} = \frac{\nabla E^T_{k+1}(\nabla E_{k+1} - \nabla E_k)}{\|\nabla E_k\|^2} \tag{12}$$

An Algorithm that takes into account second order information

In order to fit the neural network faster in terms of number of iterations (epochs), we thought that using the second order information would be effective. However, using the exact Hessian matrix would be inefficient in both computation time and memory usage. When the gradient is in order of $O(n)$, the Hessian is in order of $O(n^2)$. In our case, it would be impractical to use a method that requires the exact Hessian (such as Newton's method). To overcome this shortcoming, we came up with a naive way to approximate the Hessian. We used the secant rule (division of the difference between two consecutive gradients by the difference between two consecutive X values) in elementwise fashion and obtained a diagonal approximation of the Hessian. When we first implemented it in this way, it caused "division by zero" problem. We handled this problem by adding an ϵ term to the denominator in secant rule. Later on, we had a divergence problem when the function value approached a local minimum. We handled this problem by using a scalar term (norm of the gradient at the current iteration) to adjust the scale of the Hessian approximation at the current iteration.

Algorithm 3 An Algorithm that takes into account second order information

```
1: Input: $w_0, H_0, g_o$
2: Output: $w^*$
3: $k \leftarrow 0$
4: repeat
5:     $d_k \leftarrow H_k^{-1} g_k$
6:     $w_{k+1} \leftarrow w_k + \gamma d_k$
7:     $y_{k+1} \leftarrow g_{k+1} - g_k$
8:     $s_{k+1} \leftarrow w_{k+1} - w_k$
9:     $H_{k+1}^{-1} = diag(s_{k+1} \oslash y_{k+1})$
10:    $k \leftarrow k + 1$
11: until the stopping criterion is satisfied
```

3.2 Momentum

The momentum parameter is used to prevent the system from converging to a local minimum or saddle point. Momentum term accumulates the effect of previous gradients in a way like "exponential smoothing". This adds acceleration to the current gradient. 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 also can slow the training of the system. We are calculating the new weight values without momentum structure at each iteration as follows:

$$w_{k+1} = w_k + \triangle w_k \tag{13}$$

$\triangle w_k$ covers the momentum term, in such a way that:

$$\triangle w_k = -\gamma_k \nabla E_n(w_t) + \eta \triangle w_{k-1} \tag{14}$$

where if step size γ_k is taken as a constant at iteration, it can be changed to γ which is known as the learning rate. n which is in $E_n(w_t)$ can be a number or a set depending on the algorithm chosen. $\eta \in (0, 1]$ is the momentum parameter and determines the amount of influence from the previous iteration on the present one.

3.3 Adagrad

Adagrad is an adaptive learning rate method Duchi (2011). It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. Let $E(w)$ which may not necessarily cover all data points denote the error function we choose and g_{ik} is the gradient of this function with respect to the w^i at time step k. Here w^i is the i^{th} component of the weight vector. Then the updating equation can be written as follows:

$$w^i_{k+1} = w^i_k - \gamma g_{ik} \tag{15}$$

Adagrad modifies the general learning rate γ at each time step k for every parameter w^i based on the past gradients that have been computed for w^i. It can be shown as:

$$w^i_{k+1} = w^i_k - \frac{\gamma}{\sqrt{G_{iik} + \epsilon}} g_{ik} \tag{16}$$

where G is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients at step k. ϵ is the a small term in order to avoid division by zero. The updating rule can be written as after these definitions.

$$w_{k+1} = w_k - \frac{\gamma}{\sqrt{G_k + \epsilon}} \odot g_k \tag{17}$$

3.4 Adadelta

One can consider that the Adadelta method is the extension of the adagrad method. Instead of accumulating all past squared gradients which is valid for Adagrad method, Adadelta restricts the window of accumulated past gradients to some fixed size m Zeiler (2012). In other words, to find a new direction, the past gradients are employed such that the effect of the past ones are low. Formally, the running average $E[g^2_k]$ at time step k then depends only on the previous average and the current gradient:

$$E[g^2]_k = \xi E[g^2]_{k-1} + (1 - \xi) g^2_k \tag{18}$$

One can change the ξ value to tune the effect of current gradient. The update vector $\triangle w$ can be written as in Adadelta:

$$\triangle w = -\frac{\gamma}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{19}$$

The updating rule can be written as after these definitions.

$$w_{k+1} = w_k - \frac{\gamma}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{20}$$

4 Results

The results can be found below. Note that, Alg. means which algorithm employed, S $\rightarrow$ mini-batch Stochastic Gradient, NA $\rightarrow$ New Approach.

Alg.	Mom	AdaG.	AdaD.	γ	η	Batch	Hidden Num	epoch	time	percen.
S	-	-	-	0.1	-	100	50	500	175	93.22
S	-	-	-	0.1	-	100	100	500	175	93.22
S	-	-	-	0.1	-	100	200	500	175	92.54
S	-	-	-	0.1	-	100	300	500	175	92.41
S	-	-	-	0.1	-	100	400	500	175	92.19
S	-	-	-	0.1	-	100	500	500	175	91.97
S	-	-	-	0.1	-	100	600	500	175	92.00
S	-	-	-	0.1	-	100	700	500	175	92.12
S	-	-	-	0.1	-	100	800	500	175	91.88
S	-	-	-	0.1	-	150	50	5000	187	96.98
S	-	-	-	0.1	-	200	100	4000	344	97.54
S	-	-	-	0.1	-	200	100	4000	326	97.50

Alg.	Mom	AdaG.	AdaD.	γ	η	Batch	Hidden Num	epoch	time	percen.
S	-	-	-	0.5	-	100	50	500	167	94.39
S	-	-	-	0.5	-	100	100	500	177	95.71
S	-	-	-	0.5	-	100	200	500	203	95.38
S	-	-	-	0.5	-	100	300	500	231	95.02
S	-	-	-	0.5	-	100	400	500	260	95.42
S	-	-	-	0.5	-	100	500	500	293	95.24
S	-	-	-	0.9	-	100	100	500	176	94.85
S	-	-	-	0.9	-	100	200	500	219	95.12
S	-	-	-	0.9	-	100	300	500	235	94.94
S	-	-	-	0.9	-	100	400	500	260	94.03

Alg.	Mom	AdaG.	AdaD.	γ	η	Batch	Hidden Num	epoch	time	percen.
S	-	-	-	0.5	-	20	100	500	167	94.11
S	-	-	-	0.5	-	50	100	500	163	89.77
S	-	-	-	0.5	-	200	100	500	195	95.79
S	-	-	-	0.5	-	300	100	500	214	96.77
S	-	-	-	0.5	-	400	100	500	232	96.96
S	-	-	-	0.5	-	1000	100	500	338	97.23
S	-	-	-	0.9	-	1500	100	500	430	97.56
S	-	-	-	0.9	-	2000	100	500	1244	97.63
S	-	-	-	0.9	-	5000	150	500	11230	8.25

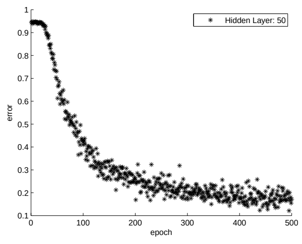
Alg.	Mom	AdaG.	AdaD.	γ	η	Batch	Hidden Num	epoch	time	percen.
S	+	-	-	0.1	0.7	100	100	3000	194	97.07
S	+	-	-	0.1	0.7	100	150	3000	286	97.55
S	+	-	-	0.1	0.7	50	100	3000	981	96.38
S	+	-	-	0.1	0.7	50	100	6000	199	97.14
S	+	-	-	0.15	0.7	100	100	2000	127	96.88
S	+	-	-	0.1	0.7	100	200	2000	283	97.27
S	+	-	-	0.1	0.9	100	200	3000	178	96.20

Alg.	Mom	AdaG.	AdaD.	γ	η	Batch	Hidden Num	epoch	time	percen.
S	-	+	-	0.5	-	300	100	500	224	96.66
S	-	+	-	0.5	-	300	100	1000	444	97.28
S	-	+	-	0.5	-	400	100	2000	1255	97.64
S	-	+	-	0.5	-	500	100	4000	3262	97.78

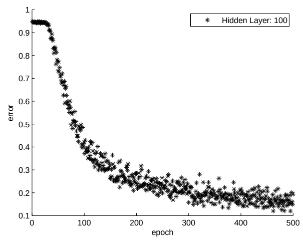
Alg.	Mom	AdaG.	AdaD.	γ	η	Batch	Hidden Num	epoch	time	percen.
S	-	-	+	0.5	-	200	100	1000	1200	93.63
S	-	-	+	0.5	-	300	100	1000	1502	93.23
S	-	-	+	0.5	-	500	100	1000	1965	93.96

Alg.	Mom	AdaG.	AdaD.	γ	η	Batch	Hidden Num	epoch	time	percen.
NA	-	-	-	0.1	-	100	100	500	321	92.63
NA	-	-	-	0.1	-	100	100	1000	661	81.23
NA	-	-	-	0.1	-	150	100	250	237	92.96

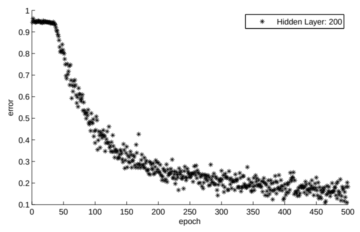
4.1 Changing the Hidden Layer Number



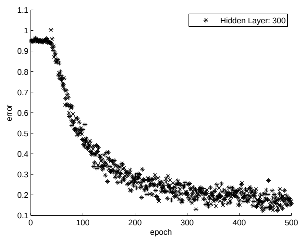
(a) Hidden 50



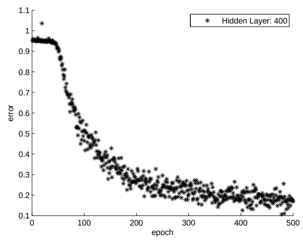
(b) Hidden 100



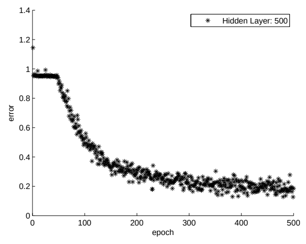
(c) Hidden 200



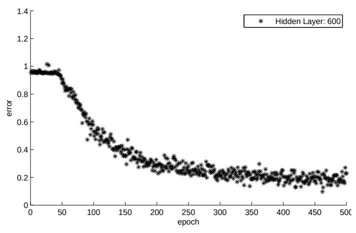
(d) Hidden 300



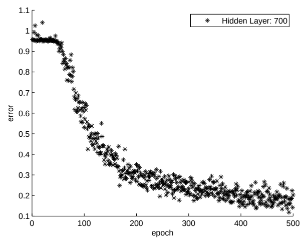
(e) Hidden 400



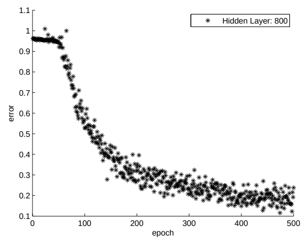
(f) Hidden 500



(g) Hidden 600



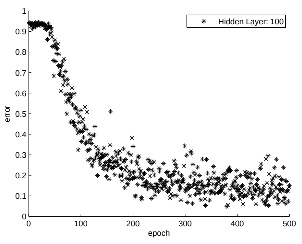
(h) Hidden 700



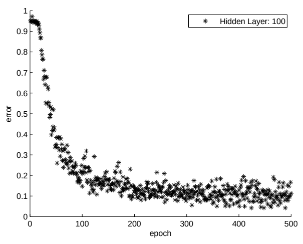
(i) Hidden 800

Figure 2: Results for different hidden layer numbers with learning rate 0.1 and batch size 100

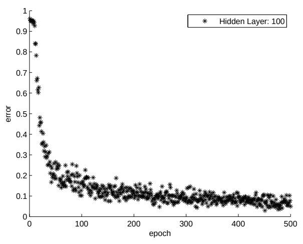
4.2 Changing the Batch Size



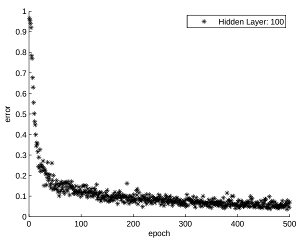
(a) Batch Size 20



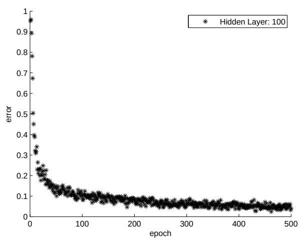
(b) Batch Size 50



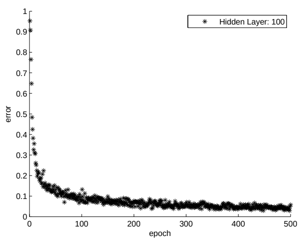
(c) Batch Size 100



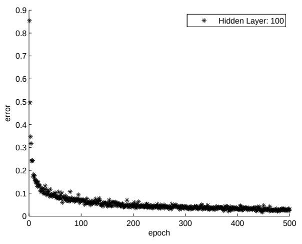
(d) Batch Size 200



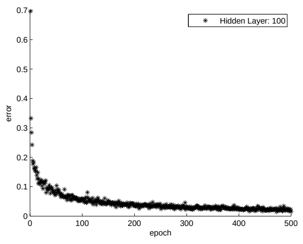
(e) Batch Size 300



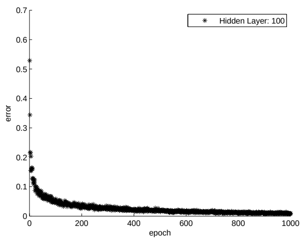
(f) Batch Size 400



(g) Batch Size 1000



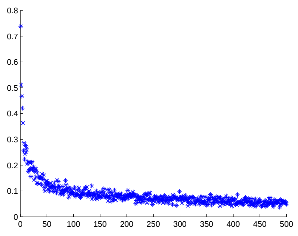
(h) Batch Size 1500



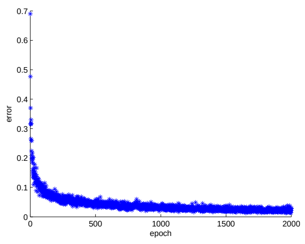
(i) Batch Size 2000

Figure 3: Results for different batch sizes with learning rate 0.5 and hidden number 100

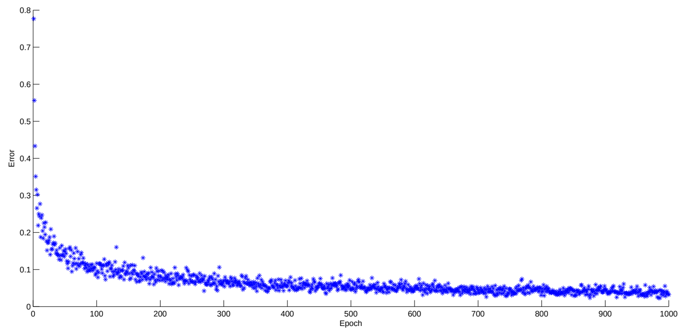
4.3 AdaGrad



(a) Batch Size 300



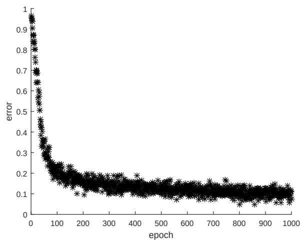
(b) Batch Size 400



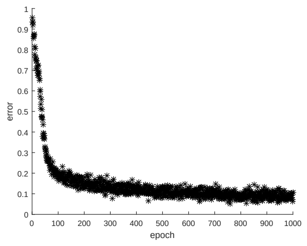
(c) Batch Size 300

Figure 4: Results for AdaGrad implementation with learning rate 0.5 and hidden number 100

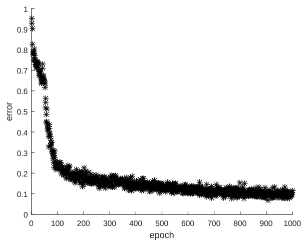
4.4 AdaDelta



(a) Batch Size 200



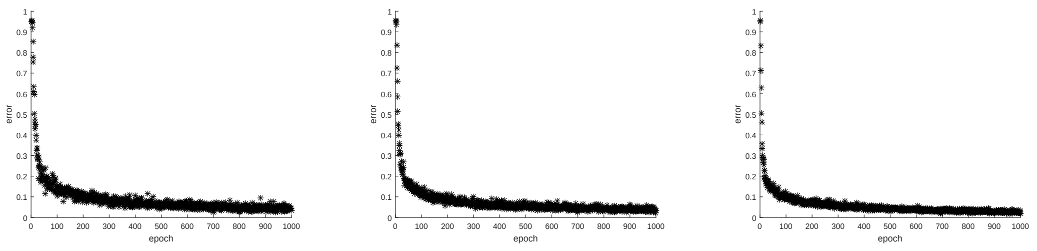
(b) Batch Size 300



(c) Batch Size 500

Figure 5: Results for AdaDelta implementation with learning rate 0.5 and hidden number 100

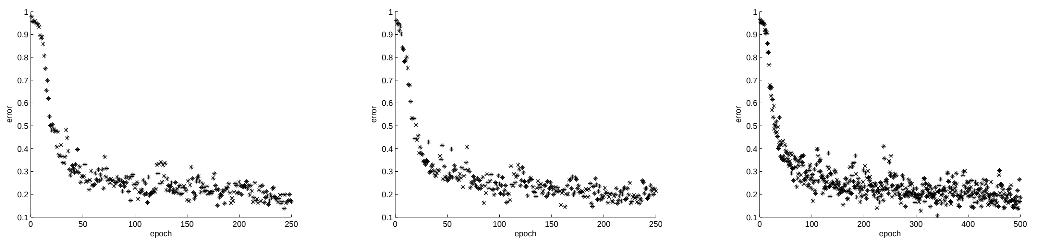
4.5 Momentum



(a) Batch Size 200 (b) Batch Size 300 (c) Batch Size 500

Figure 6: Results for Momentum implementation with learning rate 0.1 and hidden number 100

4.6 New Approach



(a) Batch Size 100 (b) Batch Size 100 (c) Batch Size 150

Figure 7: Results for New Approach implementation with learning rate 0.1 and hidden number 100

5 Conclusion

During this project, we had a chance to experiment various first order methods. In some settings, these methods were able to reach almost 98 % classification accuracy. We also handcrafted our own method to use second order information. Its accuracy is not as good as the other ones that we tried (92 % classification accuracy). Finally, we attacked the exact loss function in order to use it in minFunc. We derived its gradient and supplied it to minFunc, but it stopped after the first iteration. We could not find the cause of this problem. Given the less memory requirement and more simplistic operations, first order methods are often more preferable than second order methods in neural network training. In our case, our results with these methods were shockingly sufficient.

References

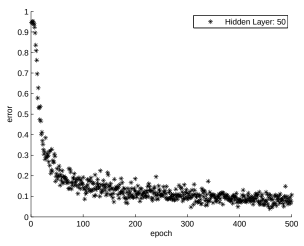
Kurkova, Vera*Kolmogorov's theorem and multilayer neural networks.*1992: Neural networks 5.3:501-506.

D. Stutz *Introduction to Neural Networks* 2014:Selected Topics in Human Language Technology and Pattern Recognition WS 13/14.

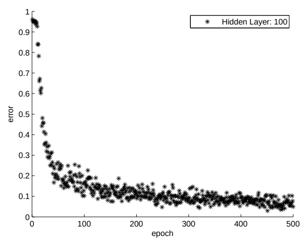
LeCun, Yann A., et al*Efficient BackProp* 2012:Neural networks: Tricks of the trade. Springer Berlin Heidelberg 9-48.

J. Duchi, E. Hazan, Y. Singer *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization 2121-2159.* 2011:Journal of Machine Learning Research.

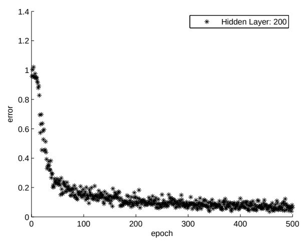
Matthew D. Zeiler *ADADELTA:An Adaptive Learning Rate Method* 2012: arXiv preprint arXiv:1212.5701



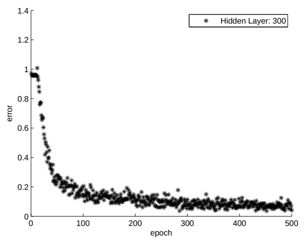
(a) Hidden 50



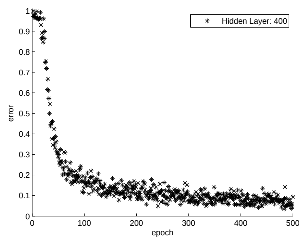
(b) Hidden 100



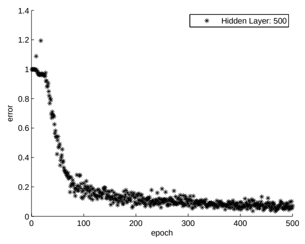
(c) Hidden 200



(d) Hidden 300

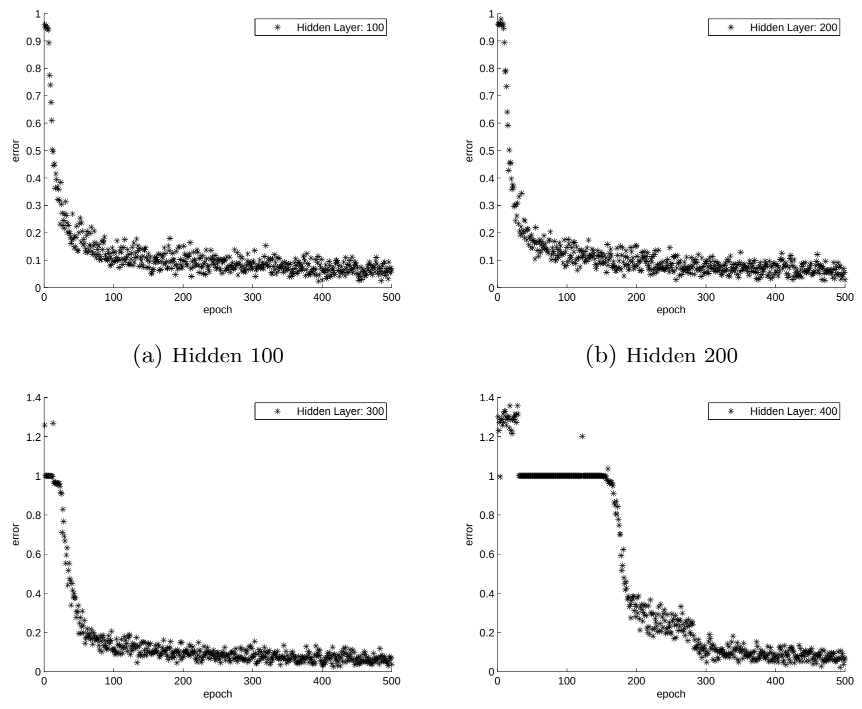


(e) Hidden 400



(f) Hidden 500

Figure 8: Results for different hidden layer numbers with learning rate 0.5 and batch size 100



(a) Hidden 100 (b) Hidden 200

(c) Hidden 300 (d) Hidden 400

Figure 9: Results for different hidden layer numbers with learning rate 0.9 and batch size 100