MNIST

team4

January 31, 2017

Report **Natali Alfonso, Simon Kern, Vangelis Kostas**
NWI-NM048C-2016
Prof. Kappen Lab Date: 17/1/17

Introduction

Please run MLP.m with parameters (layers, neurons_h, eta , max_iter, momentum, randomize, w_init, batch_size,eta_decay, GPU)

A Multi-Layer-Perceptron (MLP) is a type of feed forward artificial neural network. It can be represented as an acyclic graph of layers of nodes where each layer is fully connected to two other layers. Each node of each layer implements a non-linear function f that maps weighted inputs to outputs. Functions that are often used are *tanh* or a *sigmoid* due to their property of mapping values to $-1 : 1$ or $0 : 1$ respectively, but any other non-linearity can be used[1]. The network is now trained to a mapping of Input I to Output O by adjusting the weights that influence the input to each node.
In the forward pass, a training pattern $P \in I$ is presented to the network and it's activation is passed through the network where the activation of each node n is defined as $a_n = f(\sum_j w_j a_j)$, where f is the activation function and a_j, w_j are the weights and the activation of the neurons j of the previous layer. At the final output node a loss function $E(P)$ is calculating the divergence between the output and the target given by the training example. This error is now passed back through the network.
This is called the backward pass. For each weight a gradient $\frac{\partial E}{\partial w}$ is calculated stating the magnitude of change for this particular weight with regard to the loss function. Backpropagation[2] is an efficient way of doing this. We calculate the final loss and then pass this error backwards through the network, where the update of each weight is given by the chain rule.

[1]If a linear activation function is used, the whole network could re-written as a linear regression.
[2]Which is just a version of naive gradient descent.

The weights are initialized with small random values, from which we descend the weight space. This process can easily get stuck in a local minima. To avoid this from happening, many methods have been proposed.

In this report we investigate different strategies to minimize the chance of getting stuck in a local minima. All experiments are run with the following settings, if not specified differently:

`Neurons: 100, Hidden Layers: 1, Iterations: 100, Learning rate: 0.001, Momentum: 0`

We chose numbers 4 and 9 because they have been reported to have the highest confusion [2]. We think that we can achieve more meaningful results than with an easy contrast. Our standard setup achieves an accuracy of 99.8% on the training set and 98.9% on the test set using these initial settings.

Problem Statement

We will address the following research questions:

- What effect does Momentum have on our convergence?
- How does the weight initialization change our settings?
- How does a Wide vs. a Deep network perform
- How does Batch vs Stochastic Gradient Descent perform?
- What is the optimal learning rate for our problem
- What happens if we overfit?
- Can we speed up the runtime using the GPU?

What is a good number of layers and neurons?

Choosing the optimal network architecture is an active topic of research and can take great effort and computational resources. In general it can be considered a trade-off between computation time, generalization and performance and is often solved by using a grid search over parameter space. As the MNIST problem is a rather simple one (a quickly set-up Random Forest was able to achieve 97.6% accuracy) we found many network architectures that achieved admissible results (See Subsection 'Wide vs. Deep'). In the end we decided for the above reported parameters with momentum=0.6 due to their simplicity. We experimented with stopping criteria (for instance: *if testerr(last-5) ¡ testerr(last-10)*) but found no criteria that stopped the simulation while not being able to achieve later results if left running. For that reason we used a fixed-early-stopping of 100 iterations.

What effect does Momentum have on our convergence?

Momentum is added by taking a smaller scaled part of the previous gradient step and adding it to the current gradient step. The update formula then becomes
$w(t+1) = w(t) - \Delta w(t) + \beta * \Delta w(t-1)$.
This can be compared to the physical phenomenon of momentum and has the following effects:

1. If we are moving into a direction, we will keep some of the velocity into that direction when doing the next gradient step.

2. If several steps go into the same direction, every step will be faster than the previous one
3. It is easier to overcome local minima as we can escape them if we have enough momentum from the previous step.

We tested different momentum. Figure 1 shows the result: We can see that higher momentum terms are generally preferable and converge faster. As expected, a too high momentum term inhibits learning, as the gradient steps add up to fast and the velocity of the traverse makes it impossible to settle into a minima.

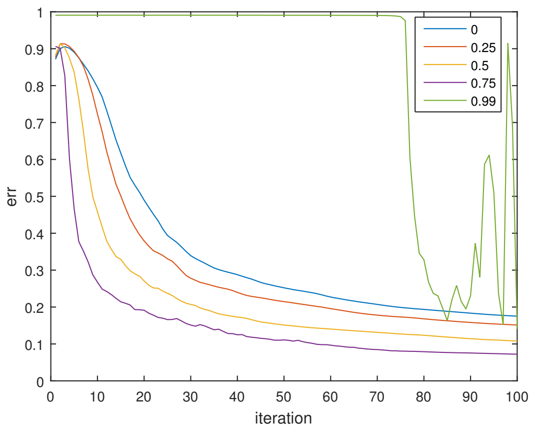

Figure 1: Convergence of different momentum values

Weight initialization

At the beginning the weights of the neurons are initialized to random values drawn from simple distribution such as a $Uniform$. We tested if it mattered for convergence from which range the values were drawn. Figure 2 shows different ranges of the Uniform distribution. We can see that smaller initial weights seem to ensure a quicker convergence. If weights are too high initially, the process can get stuck in a local minima early on already. The best weight initialization was achieved with ± 0.1 which is the value that is also suggested in the literature [1] as the rule-of-thumb $\frac{1}{sqrtneurons\_per\_layer}$

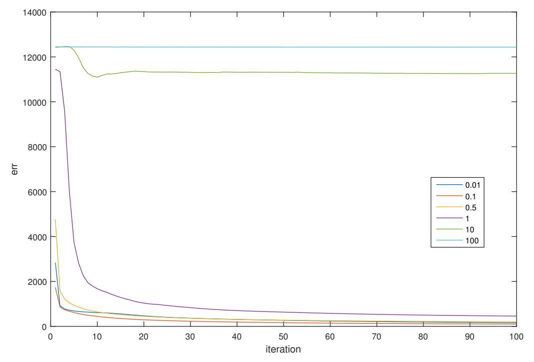

Figure 2: Convergence of weight initializations from a Uniform distribution $\pm x$

Learning rate

The learning rate is a crucial parameter defining the step size taken in the gradient direction. If the learning rate is too high, the procedure might jump over valleys of a minima while if the learning rate is too low it might get stuck in even tiny minimas. Our results can be seen in Figure 3. Although 0.01 had a slightly lower error at iteration 100 we evaluate an eta of 0.001 as optimal because the curve has less variance (jitter), making it more likely to find and stay in a minima. In generally it can be seen, that all learning rates between 0.0001 and 0.1 seem to converge to a similar error level. High learning rates of 0.5 and 1 do not perform well as the probably overshoot minima whereas the smalles learning rate of 0.000001 immediately get's stuck.

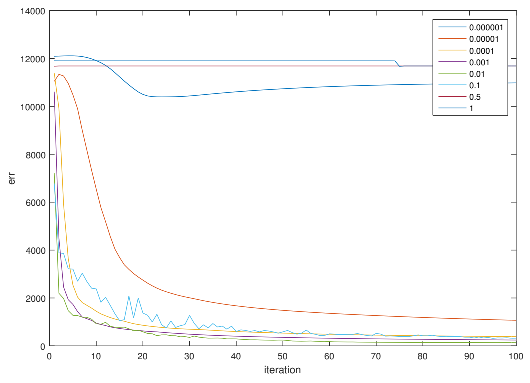

Figure 3: Convergence of different learning rates

Learning rate decay

One trick to improve convergence to a lower error level is to decrease the learning rate with time. This can be done with the following formula:
$eta_0(1 + \frac{i}{T})$
where i is the iteration number and T is a value to define the starting point of the first halving of the learning rate. Figure 4 shows our results. To our surprise we did not find improvements using the learning rate decay. The earlier we started the decrease of the decay, the slower the convergence of the network, leaving the version with almost no decay (T=1000000) as the best. This could be due to the fact, that after 100 iterations no minima is present yet that is small enough not to be found by a smaller learning rate (the valleys are not narrow enough).

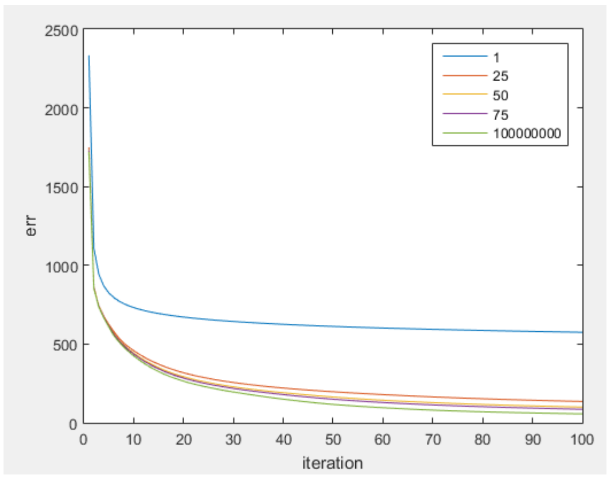

Figure 4: Convergence of different values T for the learning rate decay

Wide vs. Deep

Next we looked into how a network with many neurons per layer compares to a network with many layers of less neurons. We chose 100 neurons, either in one layer or spread out in 5 layers 20 neurons. Both networks converge relatively similar. The Deep network takes more time to go into a steep descent of the error, this could be explained with a vanishing gradient in the first layers. We can also observe that the Deep network slightly overfits by looking at the test set where the error starts to decrease less while still decreasing on the train set at the same time.

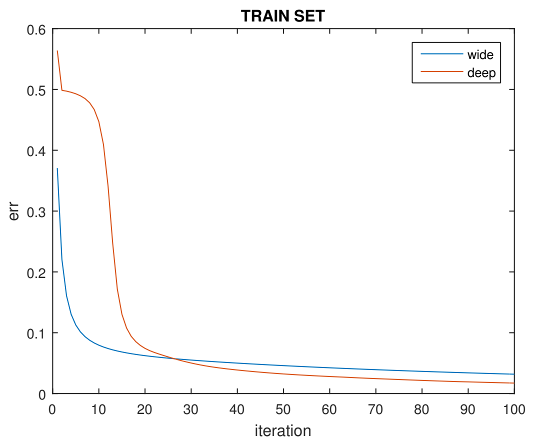

Figure 5: Train error of Wide vs. Deep

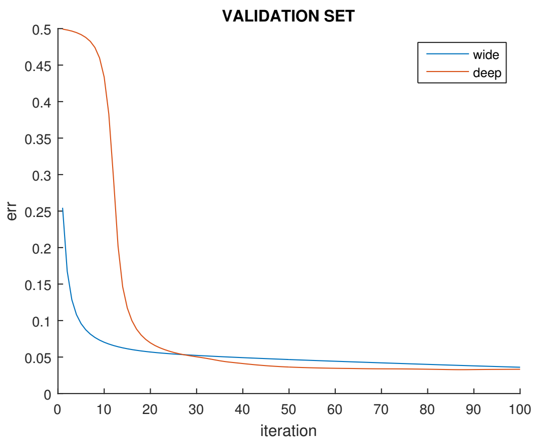

Figure 6: Test error of Wide vs. Deep

Training Variants

Different variants of using data while backpropagating exist. These can roughly be put into the following categories:

- Stochastic Gradient Descent: We take each training data point and calculate it's gradients. We update our weights after each data point.
- Mini-Batch Gradient Descent: We take n data points and add up their gradient steps. We update the weights after n data points.
- Batch Gradient Descent: We take the whole training set, accumulate all gradients of all data points and update the weights only after a full iteration of the training set.

While a paper in 2003 [3] showed nicely that Stochastic Gradient Descent is superior in almost all cases in terms of convergence, recently Mini-Batch has had a rise in popularity due to the speed up when being able to parallelize on several thousand GPU cores. Our results confirm the literature in the fact that Stochastic Gradient Descent has the fastest convergence. As a shortcoming of this experiment it is to mention that we did not adapt the learning rate to each batch size which might increase convergence speed. This effect also explains why using a full Batch Mode the error increases and get's stuck on a high level. Although adding up gradients before applying them should decrease the variance of the weight updates, we did not see any positive effect of that on the convergence.

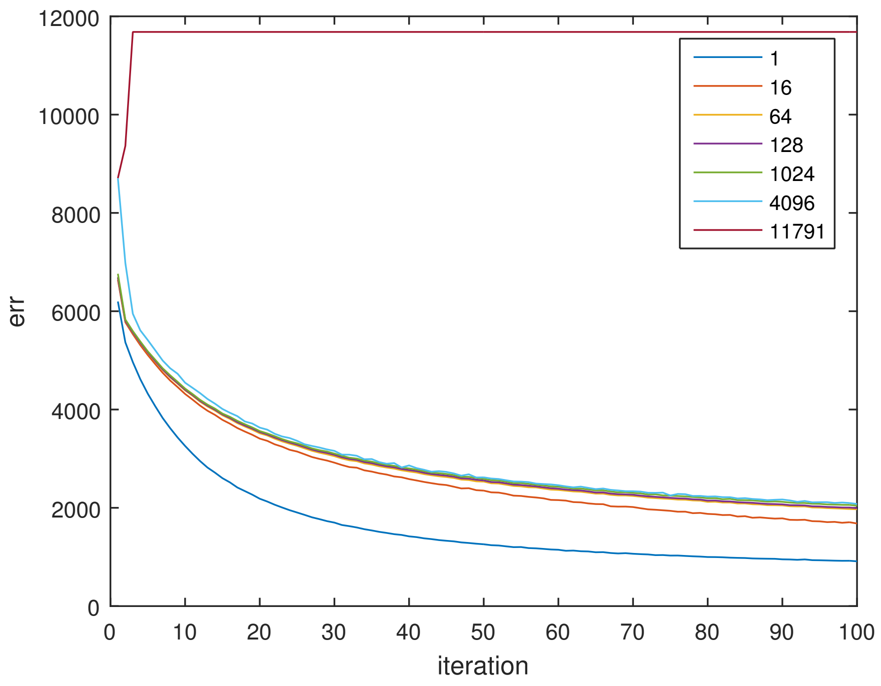

Figure 7: Convergence of different batch sizes.

Overfit

One pitfall of neural networks is the phenomenon of overfitting. This happens when a large enough network is starting to get too specific to the train set and is not able to generalize well

anymore to the test/validation set.
In this study we wanted to find out whether we can artificially create overfit. We ran a network with 11791 neurons, one for each training image. Our intuition was that a behavior like one neuron firing per image could emerge. We ran the simulation for 5000 iterations (8h of computation using the GPU). Figure 8 shows the result on the train and test set. Although a slight increase of the error can be seen while looking at the raw values (0.1% error increase per epoch) the accuracy of the network on the test set was almost not inhibited. Our guess is that the test and train set are too similar as the MNIST problem is a very simple one.

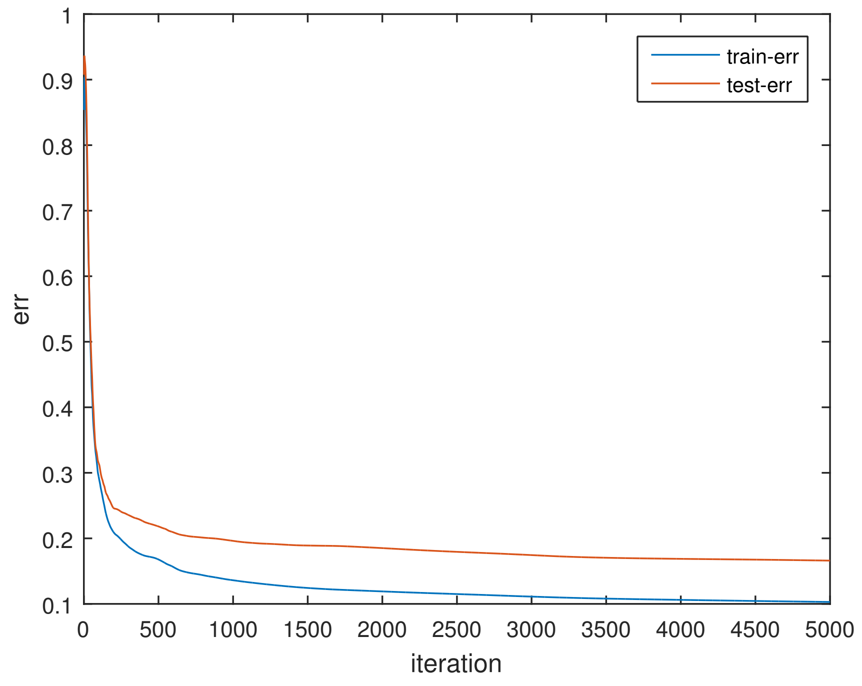

Figure 8: Trying to create overfit

GPU acceleration

Last but not least we wanted to see if we can enhance our network by using the GPU instead of the CPU. Luckily this has become relatively easy with Matlab. Each array must be initialized as a gpuArray and many of the built-in functions are overloaded to work on the GPU.

```
weights = rand(100,100)
weights = gpuArray(weights)
```

Our results show a tremendous speed-up while using the GPU (GeForce 1060) vs CPU (4x3.2Ghz). There is a slight overhead created when using smaller matrices, making the GPU version lower bound by around 20 seconds. When simulating more than 400 neurons the speed-up becomes visible. The larger the individual weight matrices are, the bigger the speedup. If neurons are spread out in different layers (= the individual weight matrices are smaller), the GPU overhead induces a slight decrease of performance for the GPU, but still far faster than the CPU (this is not shown in any graph).

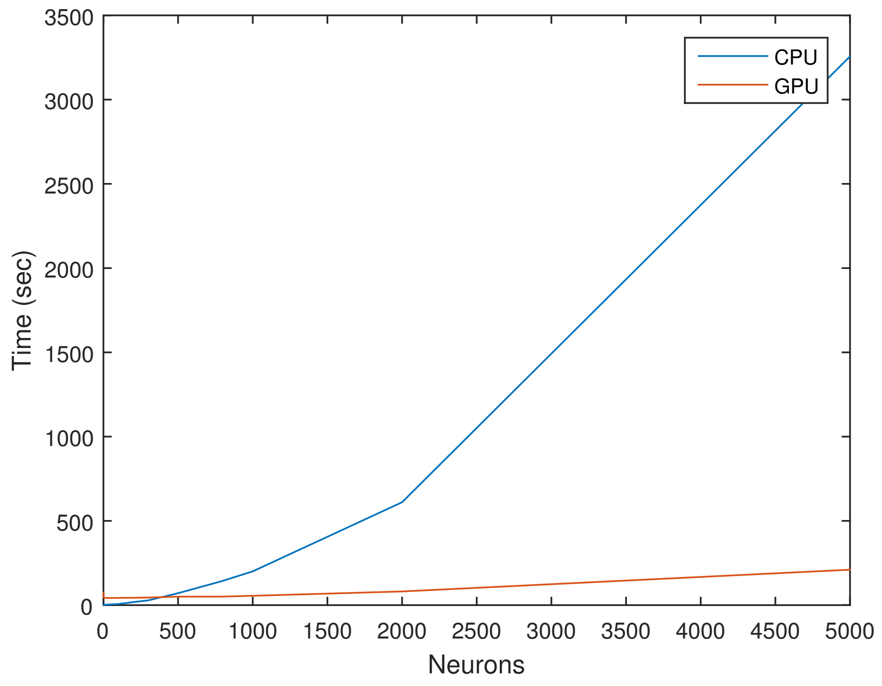

Figure 9: Using the GPU vs CPU using N neurons on a 1-layer network

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[2] Thorsten Torbenwouldkovsky. MNIST Confusion Matrix. `https://ml4a.github.io/demos/confusion_mnist/`, 2016. [Online; accessed 17-Jan-2017].

[3] D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003.