Deep Understanding of Deep Learning

Bhavdeep Singh Sethi
Columbia University

May 22, 2015

Abstract

The purpose of this document is to gain a deeper insight into neural networks – what are the parameters, how sensitive is the learning to different parameters, different ways of optimizations, etc. We run multiple experiments and document the result.

1 Basic Structure and Terminology

A Multi-Layer Neural Network (or Multi-Layer Perceptron or Artificial Neural Network - ANN) is a network model consisting of a directed graph with multiple layers that map set of input data into relevant outputs. Figure 1 shows a graphical representation of a neural network with one hidden layer.[1] Such a network can be represented as:

$f : R^D \rightarrow R^O$, where D denotes size of the input vector x and O denotes size of the output vector $f(x)$.

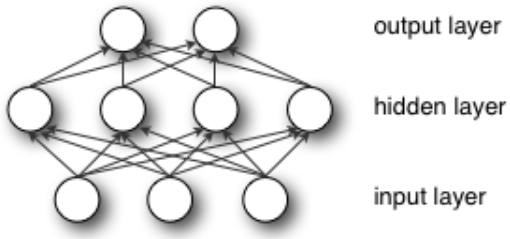

Figure 1: Neural Network with 1 hidden layer

Now,

$f(x) = A^{(0)}(W^{(0)} \cdot h(x) + b^{(0)})$

$h(x) = A^{(1)}(W^{(1)} \cdot x + b^{(1)})$

$b^{(1)}$: k-vector (called hidden units offsets or hidden unit biases)
$W^{(1)}$: $k \times d$ matrix (called input-to-hidden weights)
$b^{(0)}$: m-vector (called output units offset or output units biases)
$W^{(0)}$: $m \times h$ matrix (called hidden-to-output weights)

$A^{(0)}$, $A^{(1)}$ are activation functions of the output layer and hidden layer respectively. $h(x)$ represents output of the hidden layer. The elements of the hidden layer are called hidden units.

As we see, the output of the hidden layer is fed as input to the output layer. We can stack multiple such hidden layers to create a deep neural network. Each of these layers may have a different dimension k.

Logistic regression is a special case of the MLP with no hidden layer (the input is directly connected to the output). Neural networks with more than two hidden layers are generally called deep neural networks.

Using the above architecture, we'd like to find an algorithm which will let us find the weights and biases such that output from the network $f(x)$ approximates the actual output for the training data x. To quantify this, we define a cost (loss or objective) function C. One example of such a cost function is the mean squared error (MSE):

$C(W, b) = \frac{1}{2*n} \sum_x \|f(x) - y\|^2$

where, n is the number of training examples, y is the actual output from the training data.

We notice that $C(W, b)$ is a non-negative function. Furthermore, when $f(x) \approx y$ for all training data, then $C(W, b) \approx 0$. Inversely, if the predicted $f(x)$ is very far apart from actual output y, then $C(W, b)$ would be large. Thus, our task is to find W and b such that we minimize the cost function. We do this using algorithm called gradient descent (GD).

The main idea behind GD is to update a set of parameters in an iterative manner to minimize the error/cost function. In each update, the parameter is reduced by the gradient of the cost function, scaled by η, the learning rate. The update rule is

given by:

$$\theta_{t+1} = \theta_t - \eta \bigtriangledown L(\theta_t)$$

Gradient descent algorithm can be implemented in the batch mode, online mode or using mini-batches. In the batch mode, each update is made by running through all the data samples whereas in the online approach, an update is made for every sample that is misclassified by the current model. The batch mode has the disadvantage of ending up in the local minima. Also, when the number of training examples are high, the learning can be really slow. On the other hand, the online mode can get affected a lot because of noisy data. The standard online back-propagation performs better than many fast learning algorithms as soon as the learning task achieves a realistic level of complexity and when the size of the training set goes beyond a critical threshold. [2]

A compromise between the batch and online mode is the mini-batch gradient descent such that batches containing more than one sample, but not the entire sample set.

The training set is used for minibatch stochastic gradient descent on the differential of the objective function. As we perform this gradient descent, we periodically calculate the validation set to see how our model is doing on the real objective function. When we see a good model on the validation set, we save it and check it on the test set.

2 Learning Basic Functions

We start the experiments by running the code [1] on syntenthic data set which uniformly varies from -6 to 6. The data set consists of 1000 training, 100 validation and 100 test examples.

2.1 Learning $y = x$

For $y = x$ dataset, we run the experiment using the following configuration:

- 1 Hidden Layer with 1 hidden unit and no activation function
- Logistic Regression as the Output layer

[1] Source Code: https://github.com/BhavdeepSethi/DeepLearning.git

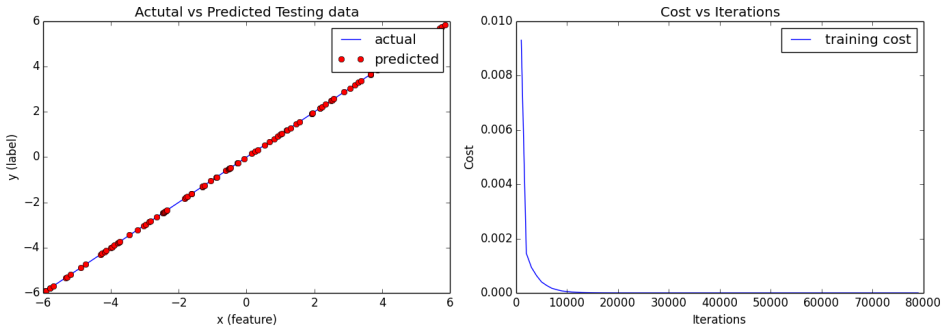

Figure 2: Graphs for $y = x$ data

- Cost $C(W, b)$: MSE
- W and b initiallized randomly
- η = 0.01
- $batch\_size$ = 10
- L1 and L2 regularizers as 0.0

Here are the final parameter values that we observe:

$b^{(1)}$: [0.25452751]
$W^{(1)}$: [-1.07836336]
$b^{(0)}$: [0.23603131]
$W^{(0)}$: [-0.92733121]

Ideally, the weight of both the hidden layer and output layer should be 1.0 with biases as 0.0, but due to precision errors, we get the above as the best parameters.

As seen in Figure 2, $f(x) \approx y$ for all test data and thus, $C(W, b)$ is minimized to 0.0.

2.2 Learning $y = x^2$

We first run the experiment using the same configuration as in $y = x$ except two changes:

- η = 0.0001

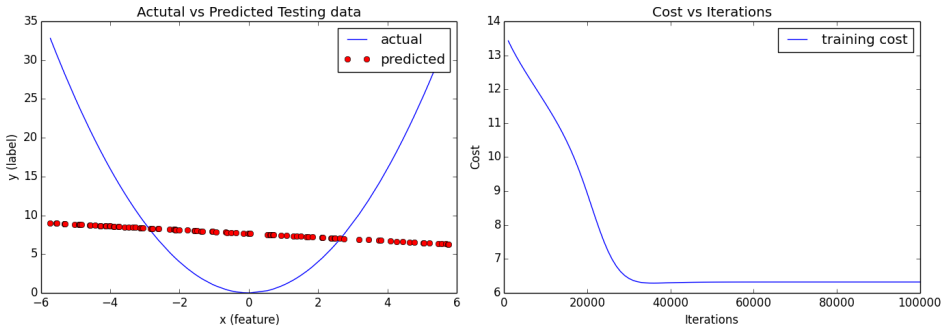

Figure 3: Graphs for $y = x^2$ data (without activation)

- $batch\_size$ = 100

As expected, the above configuration is not able to learn the function $y = x^2$. This is verfied from Figure 3. We need to introduce non-linearity in the activation functions.

We run the experiment on the same dataset using the new configuration:

- 1 Hidden Layer with 100 hidden units and following activation function: $h(x) = \sum_{i=1}^{4} (W^{(i)} \cdot x + b^{(i)})^i$
- Logistic Regression as the Output layer
- Cost $C(W, b)$: MSE
- W and b initiallized randomly
- η = 0.0000001
- $batch\_size$ = 30
- L1 and L2 regularizers as 0.0

As we see in Figure 4, adding the activation function helps us to learn the non-linearity in the data. We notice that the current network configuration is probably an overkill for learning $y = x^2$ and we may end up overfitting. Figure 5 shows the improvement (in fit) we achieve when we run the experiment again using L1 value of 0.1.

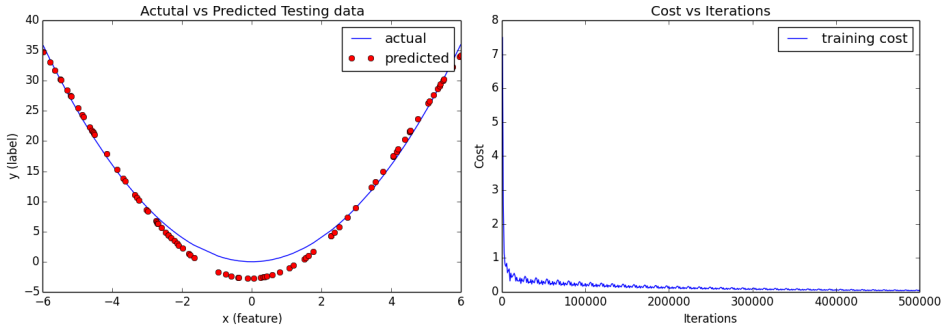

Figure 4: Graphs for $y = x^2$ data (with activation)

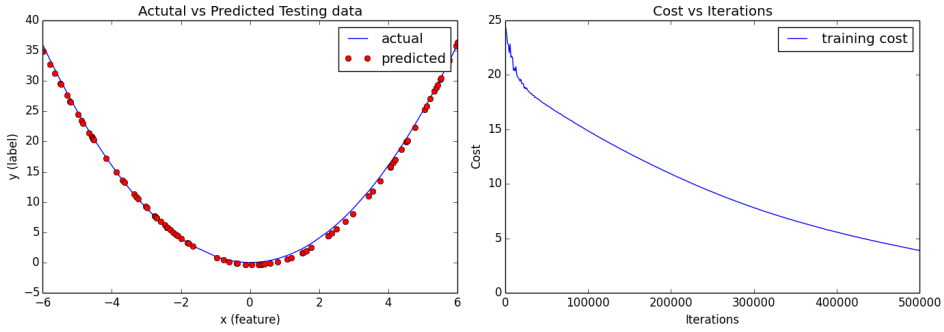

Figure 5: Graphs for $y = x^2$ data (with activation) and 0.1 L1

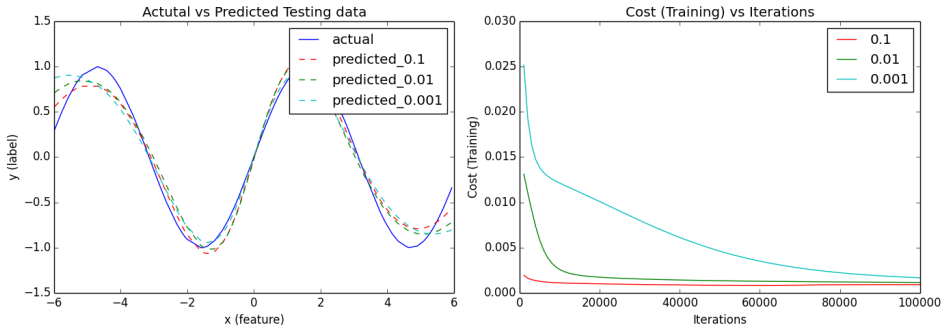

Figure 6: ANN for learning sinx using different learning rates

3 Learning $\sin x$ with Varying Parameters

Using data sampled from $\sin x$ function, we perform experiments to see the impact of different hyper-parameters on the learning task.[3] The default configuration of our network is as follows:

- 1 Hidden Layer with 100 hidden units and $tanh$ activation function.
- Logistic Regression as the Output layer
- Learning method: Standard Gradient Descent
- Cost $C(W, b)$: MSE
- W and b initiallized randomly
- η = 0.1
- $batch\_size$ = 100
- L1 and L2 regularizers as 0.0

3.1 Role of Learning Rate η

There is a lot of research on suitable value of learning rate. The learning rate η indicates the step size in the search process. If η is too high, the gradient starts diverging; if it is too low, it takes a long time to learn. A good rule of thumb is, starting at 0.1 and then decrementing in factors of approximately 3. For eg., your

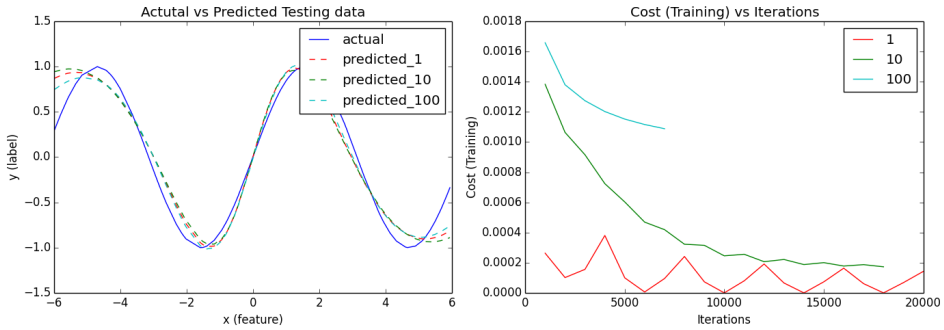

Figure 7: ANN for learning sinx using different batch size

initial experiments will be with η as 0.1, 0.03, 0.01, 0.003, 0.001 and so on. We fix η to the number which gives us the least validation error.

We compare results of η values 0.1, 0.01 and 0.001. As we see in Figure 6, η value 0.1 is able to learn the function much faster (less number of iterations) as compared to η value of 0.01 and 0.001.

3.2 Role of $batch\_size$

Batch size and learning rate generally complement each other. $batch\_size$ of 1 is called as online learning. Basically, the neural network learns from each training example. This is useful to ensure the descent doesn't get stuck in local minima but at the same time it is quite sensitive to noisy data. $batch\_size$ equal to training data is the standard gradient descent. Using a $batch\_size$ between the two extremes is the preferred to perform gradient descent. Consider you have 20000 training examples. Using a $batch\_size$ of 100 as opposed to the entire dataset speeds up estimating the gradient by a factor of 200!

We compare results of $batch\_size$ of 1, 10 and 100. As we see in Figure 7, $batch\_size$ of 1 (on-line learning) is able to learn the function much faster, but the cost function has lots of ups and down as every training example updates the gradient. This can have a lot of impact in real world applications where data contains lots of noise as well.

Batch size of 100 on the other hand, considers lots of training examples for every update making it much slower but stable.

An ideal value of batch size should be such that the cost function remains smooth and at the same time learns the data at a good rate.

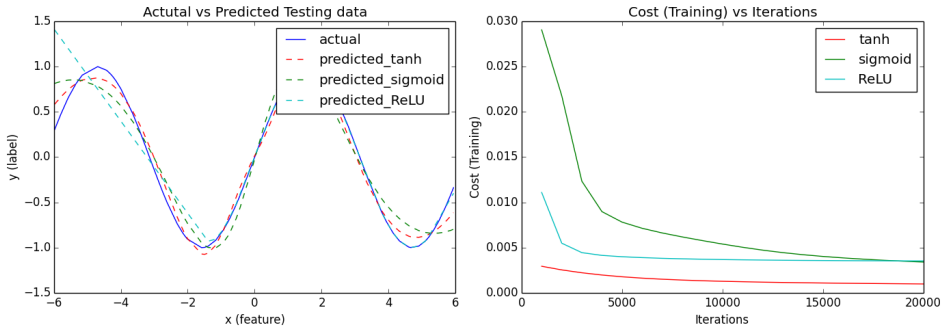

Figure 8: ANN for learning sinx using different activation functions

3.3 Role of Activation Functions

We introduce activation functions to handle non-linearity. Three of the most common ones used are $tanh$, $sigmoid$ and $RectifiedLinearUnit(ReLUs)$.

$tanh(a) = \frac{(e^a - e^{-a})}{(e^a + e^{-a})}$

$sigmoid(a) = \frac{1}{(1+e^{-a})}$

$(ReLUs) : f(a) = max(0, a)$

As explained in "Efficient BackProp"[4] activation functions that are symmetric around the origin are preferred because they tend to produce zero-mean inputs to the next layer. In practice, $tanh$ perform better than $sigmoid$. $ReLUs$ are generally preferred over other nonlinearities since it has been found to speedup the convergence of stochastic gradient descent compared to the $sigmoid$ or $tanh$ functions. $ReLUs$ can be implemented by simply thresholding a matrix of activations at zero. [5]

Figure 8 captures the result of various activation functions. $tanh$, $sigmoid$ and $ReLUs$ are able to learn the $\sin x$ data easily. A higher learning rate negatively impacts $ReLUs$ so we need to be careful when choosing our activation function.

3.4 Role of Hidden Units

This is dependent on the dataset. The more complicated input distribution will require more hidden units (network capacity) to model it. Important thing to notice is that as we increase the number of hidden units, the need to use L1/L2 regularizer also increases. The additional capacity may overfit the training data giving poor test

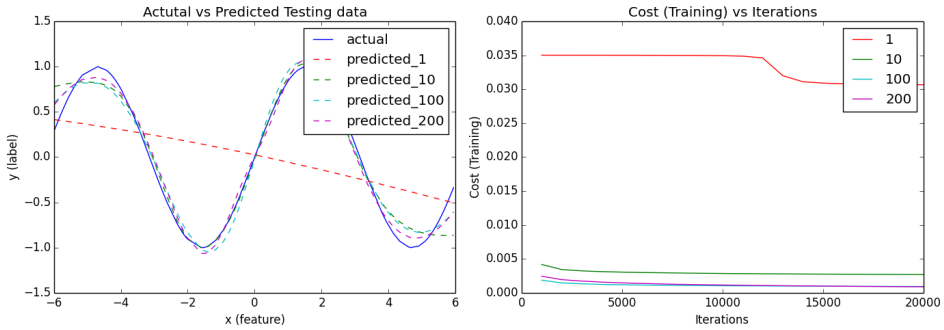

Figure 9: ANN for learning sinx trying different number of hiddent units

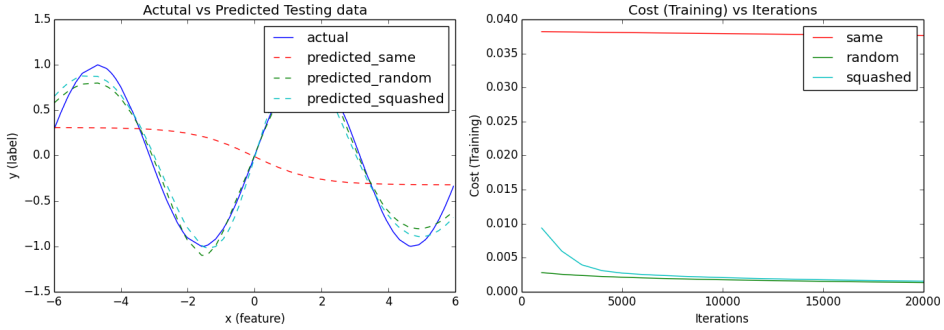

Figure 10: ANN for learning sinx using different parameter initializations

results. Also, adding more capacity naturally makes the process computationally slower.

Figure 9 highlights the impact of increasing the hidden units on learning. 1 hidden unit is not sufficient to handle the non-linearity of data sampled from $\sin x$. Beyond a particular point, increasing the number of hidden units will not have any positive impact on the learning method. Even though we're running the experiment on same number of iterations, the time required per iteration increases as we increase the number of hidden units.

3.5 Role of Parameter Initialization

This is a very important and well documented problem. When we create our neural network, we have to initialize the weights and baises. It is possible to make the learning hard because of our choice of initial weights. The simplest apporach for

this is to avoid this is to initialize the weight parameters randomly using a Gaussian distributions with mean 0 and standard deviation 1.

Since we want the activation function to operate in it's linear regime (gradients are largest), we want the weights to be small enough around the origin.

As mentioned in [Xavier10], a good practice is to initialize based on the activation function used. For eg.

$tanh(a) = uniform[-\frac{\sqrt{6}}{\sqrt{n_in+n_out}}, \frac{\sqrt{6}}{\sqrt{n_in+n_out}}]$

$sigmoid(a) = uniform[-4 * \frac{\sqrt{6}}{\sqrt{n_in+n_out}}, 4 * \frac{\sqrt{6}}{\sqrt{n_in+n_out}}]$

In practice, it doesn't much matter how you initialize the biases. We can continue to initialize the baises randomly and let gradient descent learn the appropriate biases.

One important thing to note is that no matter how you initialize the parameters, you may still end up with the same accuracy. What will differ is the number of epochs required to reach there.

We run experiments using three weight initializing techniques:

- Same: All weight params are initialized to the same randomly selected value.
- Random: Weight params are randomly sampled from a Gaussian distributions with mean 0 and standard deviation 1.
- Squashed: Randomly distributed using the formula given above for $tanh(a)$

As seen in Figure 10, both the Random and Squashed technique perform pretty well. If we initialize all weights to the same value, then all the neurons will follow the same gradient, they will behave the same way as one another and will always end up with the same weights in the end. Also, initialing with the same weight baises your solution to particul set of weights i.e. you end up getting stuck in local minima.

3.6 Role of Learning Methods

As seen before, The learning rate η is typically set experimentally, using a tuning procedure in which the highest possible learning rate such that no divergence is seen, and the convergence is also reasonably fast, is hand picked. Even after many experiments, the learning rate chosen may be sub-optimal. Additonally, a fixed learning rate cannot handle the shape of a typical multi-dimensional error function which has lots of minima.

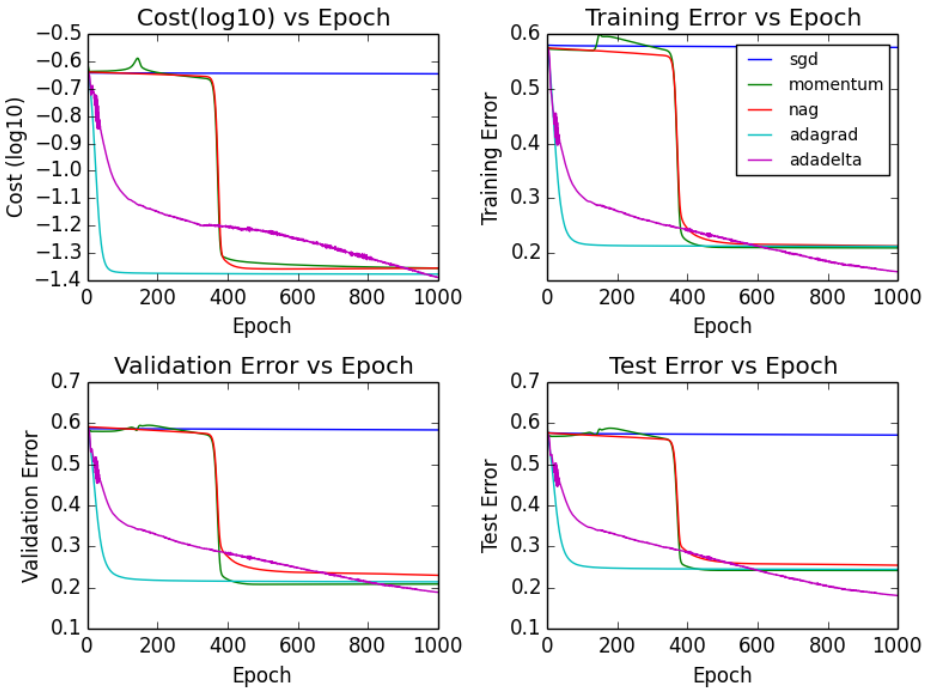

Figure 11: ANN for learning sinx using different learning methods

In the scenario where we have a fixed learning rate, the optimum value that learns all parameters reasonably well would be the inverse Hessian. However, computing the inverse of the Hessian matrix is computationally intensive. Some adaptive learning techniques provide a good approximation of the Hessian, thereby allowing Stochastic Gradient Descent to exhibit fast convergence as well as a high success rate.

Some of the adaptive learning methods are Momentun[6], Nestorov's Accelerated Gradient (NAG)[7], AdaGrad[8], AdaDelta[9], AdaSecant[10], RMSProp[11], etc.

We try to learn $\sin x$ using different adaptive methods for few number of epochs. As seen in Figure 11, the adaptive learning methods perform better than the vanilla Gradient Descent. While AdaGrad has the lowest test error earlier on, it saturates (due to annealing effect) and stops learning. On the other hand, AdaDelta ends up with the least test error after 1000 epochs learning considerably faster than the rest of the methods.

4 Experimental Setup

4.1 DataSets

We run our experiments on the 20newsgroup data which is a standard dataset for document classification. It is the biggest dataset that we're running our experiments on. The training data consists of 11,314 documents with total of 20 distinct classes. We use TF-IDF representation of the text as features and limit the number to 20,000 features for our experiments. The test data consists of 7,532 documents. We have also stripped the headers, footers and quotes of the data since they were overfitting the model for the 20newsgroup dataset. By stripping these data specific items, we ensure that our model can work on any dataset.

4.2 Architecture & Framework

The code is written in Theano[12]. Theano is a python library that lets us define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. Some of the features of using Theano are tight intengration with numpy, transparent use of GPU, efficient differentiation, dynamic C code generation. [13]

We're running the experiment using Deep Neural Nets with one hidden layer. The hidden layer consists of 200 neurons and uses Rectified Linear Unit as the ac-

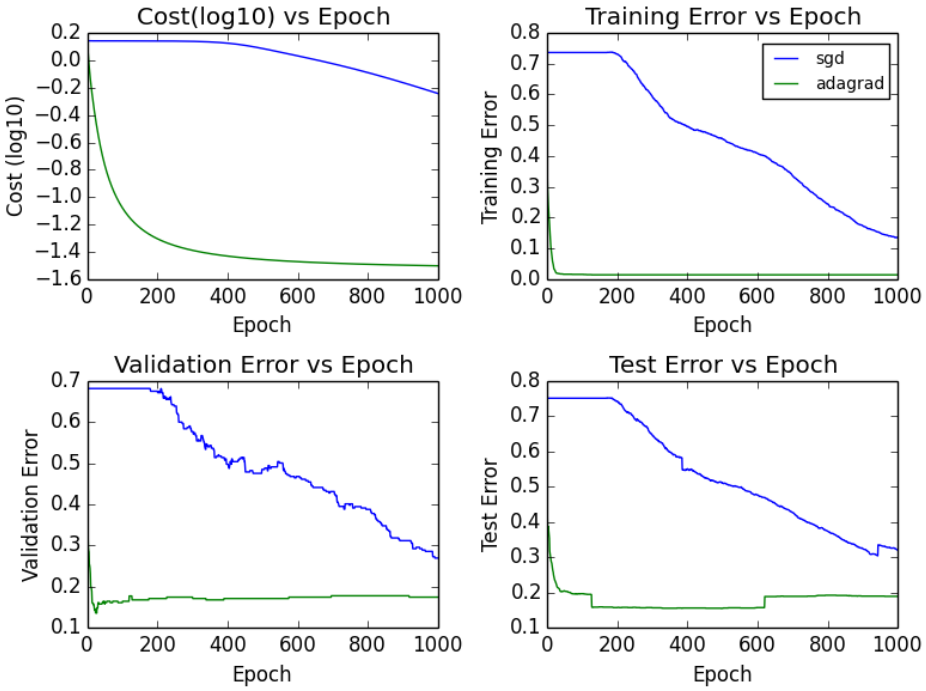

Figure 12: ANN using 20newsgroup

tivation function. The output layer uses Logistic Regression with Negative Log Likelihood as the loss function. The prediction is made using arg max over the Softmax function. L1 regularizer co-efficient used is 0.0, while the L2 regularizer co-efficient is inverse of the size of the training data.

4.3 Machine

We're running the experiments on Amazon EC2 instances. The configuration of the machine is: g2.2xlarge (26 ECUs, 8 vCPUs, 2.6 GHz, Intel Xeon E5-2670, 15 GiB memory, 1 x 60 GiB Storage Capacity)

5 Results

5.1 Baselines

We start our experiments by getting baseline results using SVM and NaiveBayes. We achieved 31.85% test error using NaiveBayes and 32.06% using SVM.

5.2 ANN

We run the experiment with ANN using standard gradient descent with a fixed learning rate and gradient descent with adaptive learning technique (AdaGrad).

As shown in Figure 12, Adagrad learns much faster than standard Gradient Descent. By the end of 1000 epochs, we achieved 32.06% test error using standard Gradient Descent and 15.5% test error using AdaGrad. Running them for much longer, we were able to achieve 15.5% test error for both the methods, a significant improvement over the baselines. As observed, AdaGrad was able to achieve the lowest test error in just 330 epochs. Comparing this to SGD, which had a test error of 62.25% in the same number of iterations.

6 Summary

From our experiments, we observe that deep neural nets give a significant performance improvement over standard SVM and NaiveBayes. We understand the impact of various hyper-parameters on the performance of our neural network and document the results of tuning them individually.

References

[1] Deep learning tutorials, 2015. The tutorials presented will introduce you to some of the most important deep learning algorithms and will also show you how to run them using Theano.

[2] W. Schiffmann, M. Joost, and R. Werner. Comparison of optimized backpropagation algorithms. In *Proc. of ESANN'93, Brussels*, pages 97–104, 1993.

[3] Michael Nielsen. Neural networks and deep learning.

[4] Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. In G. Orr and Muller K., editors, *Neural Networks: Tricks of the trade*. Springer, 1998.

[5] Convolutional neural networks for visual recognition, 2015. Class notes for CS231n: Convolutional Neural Networks for Visual Recognition, Stanford University.

[6] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1988.

[7] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147, 2013.

[8] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[9] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[10] Çaglar Gülçehre and Yoshua Bengio. ADASECANT: robust adaptive secant method for stochastic gradient. *CoRR*, abs/1412.7419, 2014.

[11] Yann N. Dauphin, Harm de Vries, Junyoung Chung, and Yoshua Bengio. Rmsprop and equilibrated adaptive learning rates for non-convex optimization. *CoRR*, abs/1502.04390, 2015.

[12] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[13] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.