Recognition of Handwritten digits in MNIST dataset using Multilayer Perceptron

Kumar Rohit Malhotra

***Abstract*—Artificial Neural Networks are the state of the art machine learning models. Multilayer Perceptron is a feedforward neural network, with each layer completely connected to the next one, and the neurons in any individual layer not being connected to each other. Image recognition is one of the fields in which it has proven to be of great use in recent times. This paper talks about how to build a neural network for recognition of handwritten digits in MNIST database.**

***Keywords*—Artificial neural networks, MNIST database, Back-propagation, Batch gradient descent, Stochastic gradient descent, SGD-momentum, Downsampling, Early Stopping**

I. Introduction

Human brain is undoubtedly the smartest machine on the planet. Researchers have always been fascinated with the idea of replicating the human brain, so as to build the systems that can *think* the way humans brains do, and the idea behind building such systems is to replicate the neurons in a human brain. Artificial neural networks(ANN) is the very application of this idea. Just as in a neuron, or as in any machine learning algorithm, we have an input layer and an output layer in ANN. Additionally, we have hidden layers between the input and the output. These neural networks can be used to learn linear as well as non-linear functions.

Multilayer Perceptron(MLP) is a feedforward artificial neural network, in which each each layer is completely connected to the next one, as a directed graph, without any connections between the neurons of the same layer. A simple MLP with 3 input units, 2 output units, and 1 hidden layer with 4 units is shown in Fig. 1. Every neuron, which consists of all the units other than the input units, have an activation function. This activation function can be linear or non-linear. In addition to the neurons and the input units, we add a bias unit to every layer other than the output layer. Bias unit helps in shifting the intercept of our function when needed to avoid underfitting. The update of weights in the MLP are achieved by backpropagation. Backpropagation essentially means propagating the error back from the output to the previous layers.

In this paper, we explain how to build and train an MLP to recognise handwritten digits. The images for the handwritten digits are taken from the MNIST dataset. We have 70,000 images of handwritten digits with each image having a dimension 28 x 28, and labels for each image giving us the digit in that image. Out of these, 50,000 images are used for training the network, 10,000 images are used for validation so as to enable early stopping, and rest 10,000 images are used for testing. We can see some of these images in Fig. 1. Due to the constraints on computational power, we won't be using all the samples from the training dataset. The number of training samples used in each experiment have been stated ahead.

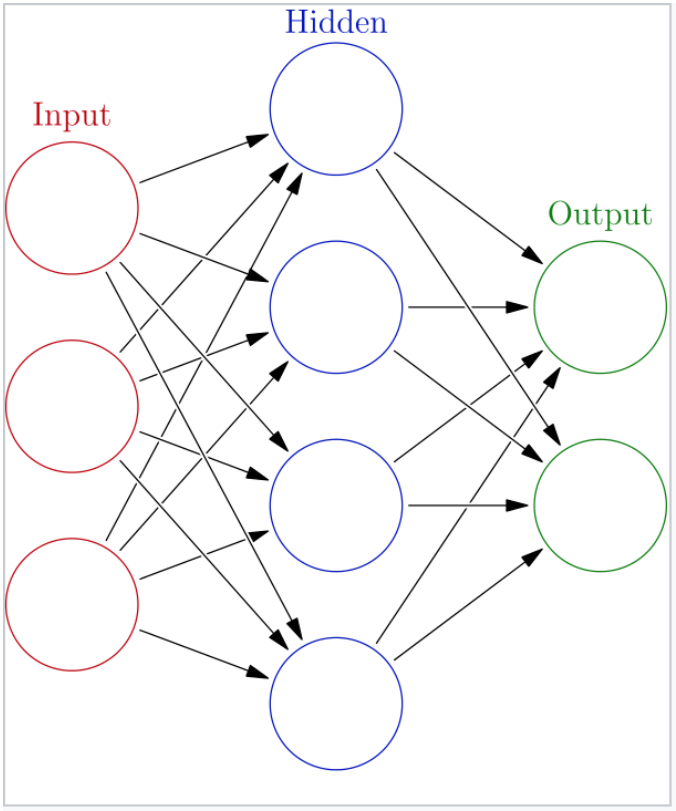

Fig. 1: An example of a 1 hidden layer MLP

We begin with training the MLP using batch gradient descent, wherein we train the MLP with the complete training set, and updating the weights at the end of that training. This is repeated over many iterations, known as epochs. After that, we train the MLP using Stochastic Gradient Descent(SGD) wherein we'll train the MLP with one sample at a time, updating the weights through backpropagation after forward pass of every sample. Algorithm 1 shows how backpropagation works in the case of SGD. Batch gradient descent algorithm is

This paper was submitted on Dec 12, 2016.

Kumar Rohit Malhotra is with the Department of Computer and Information Science and Engineering at the University of Florida, Gainesville, FL 32611, USA (e-mail: krohitm@ufl.edu)

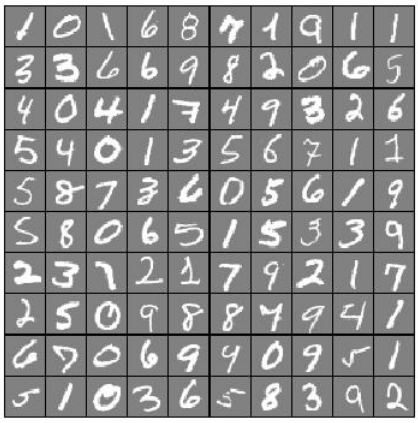

Fig. 2: Displaying 100 random images from the training dataset

slightly different from this algorithm in the case that it makes a forward pass over the whole training dataset, and then performs backpropagation to update the weights.

We usually face the problem of being stuck in a local

Algorithm 1 Backpropagation learning algorithm

for each sample i in training batch **do**
  FORWARD PASS
    Starting from the input layer, do a forward pass through the network, computing the activities of the neurons at each layer.
  BACKWARD PASS
    Compute the derivatives of the error function with respect to the output layer activities
    for layer in layers **do**
      Compute the derivatives of the error function with respect to the inputs of the upper layer neurons
      Compute the derivatives of the error function with respect to the weights between the outer layer and the layer below
      Compute the derivatives of the error function with respect to the activities of the layer below
    end for
    Update the weights.
end for

minima in case of neural networks, which can be overcome by regularization such that the model goes beyond the local minima to reach the global minima. For this, we'll be using the SGD-momentum algorithm. In all the experiments, we are using logistic sigmod function as the activation function, given as:

$$a_i^l = \frac{1}{1 + e^{-z_i^l}} \tag{1}$$

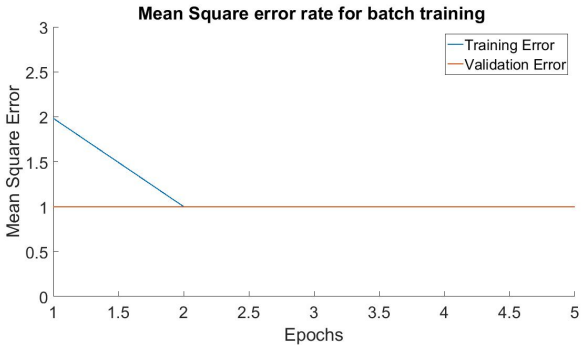

Fig. 3: Learning Curve for Batch Gradient Descent using a learning rate of 0.01

where a_i^l is the activation of i^{th} unit in the l^{th} layer, and z_i^l is given as :

$$z_i^l = \sum_{j=1}^{J} \omega_j x_j + b \tag{2}$$

where J is the total number of units in the previous layer, excluding bias, ω_j is the weight incident on i^{th} unit in the l^{th} layer from the j^{th} unit in the $(l-1)$ layer, x_j is the output of the j^{th} unit in the $(l-1)$ layer, and b is the bias unit. The activation function being used is a non-linear function giving output between 0 and 1. In order to utilize this function for our outputs, we change the one unit outputs to ten unit outputs, such that for any sample *i*, the value in corresponding ten unit output is 1 at the index equal to the value of digit in sample *i*, and rest all nine units are zero. For digit zero in the labels, we set the tenth unit of corresponding new ten-unit output as 1. Thus, we'll be training ten different classifiers within a single MLP. This is also known as one-vs-rest classification. Mean Square Error(MSE) has been used as the cost function in all the experiments.

II. Batch Gradient Descent

We begin our experiment with batch gradient descent algorithm. We run the feedforward algorithm for the complete training set, and find the net training error using MSE as the cost function. We update the weights of the network using the backpropagation algorithm. Then we get the validation error by running the feedforward algorithm on the validation dataset and finding the MSE. We run the experiment for various learning rates. The network attains a local minima within the first epoch, as can be seen in Fig. 3. The maximum test accuracy obtained in batch gradient descent is 11.35%. On testing this trained model on our testing data, we get the results denoted by the confusion matrix given in Table I. The row indices denote the true labels and the column headers indicate the predicted labels. As can be seen from the table, we are not getting any good results from this method. All the digits are being classified as a single digit, which is 1 in this case.

TABLE I: Confusion Matrix for Batch Gradient Descent

	0	1	2	3	4	5	6	7	8	9
0	0	980	0	0	0	0	0	0	0	0
1	0	1135	0	0	0	0	0	0	0	0
2	0	1032	0	0	0	0	0	0	0	0
3	0	1010	0	0	0	0	0	0	0	0
4	0	982	0	0	0	0	0	0	0	0
5	0	892	0	0	0	0	0	0	0	0
6	0	958	0	0	0	0	0	0	0	0
7	0	1028	0	0	0	0	0	0	0	0
8	0	974	0	0	0	0	0	0	0	0
9	0	1009	0	0	0	0	0	0	0	0

III. Stochastic Gradient Descent

Since batch gradient descent doesn't give us a good accuracy, we try to train the model using stochastic gradient descent algorithm. The weights update function used in SGD is given by:

$$\omega(n+1) = \omega(n) + \lambda\varepsilon(n)x(n) \tag{3}$$

where $\omega(n+1)$ denotes the weights for the $(n+1)^{th}$ sample, $\omega(n)$ denotes the weights used in forward pass for the n^{th} sample, λ is the learning rate, $\varepsilon(n)$ is the error observed for the n^{th} sample and $x(n)$ is the n^{th} sample. Algorithm 1 gives the method followed in SGD. We run this algorithm with a slight change. We run the backpropagation algorithm for a random sample at a time, making the forward pass to get the activations and the backward pass to update the weights. This way we run the backpropagation for 100 samples, and find the training error, validation error, and test accuracy after that. We do this for 200 epochs. This way, we train our model using 20,000 samples from the training set. This has been done just for the ease of representing the graphs for the learning curve and accuracy. The experiments were done for a single hidden layer with varying number of hidden units from 400 to 500. Every epoch marks 100 samples, thereby leading to an experiment for 20,000 samples in training set. The maximum accuracy obtained over test set in this case was 93.18%, which was for λ=0.7, a hidden layer with 500 units, and 200 epochs. The learning curve and accuracy for the best result is shown in Fig. 4 and Fig. 5 respectively.

The confusion matrix for this case can be seen in table II. The row indices denote the true labels and the column headers denote the predicted labels. A huge number of digits can be seen to be predicted correctly in the confusion matrix. The digit 4 is being predicted as 9 for 67 cases, which is more than for any other digits being predicted as the incorrect digit. This may be because of the shape of 4 being most similar to 9 in many cases. Similarly, digit 2 is being incorrectly predicted as 8 for 31 samples, digit 7 is being predicted as 9 for 26 samples, digit 5 being predicted as 3 for 23 samples, and digit 9 is being incorrectly as 7 for 14 samples, which are the maximum number of incorrectly predicted digits for each of them. Cases like 1 being predicted as 8 for 19 samples, and

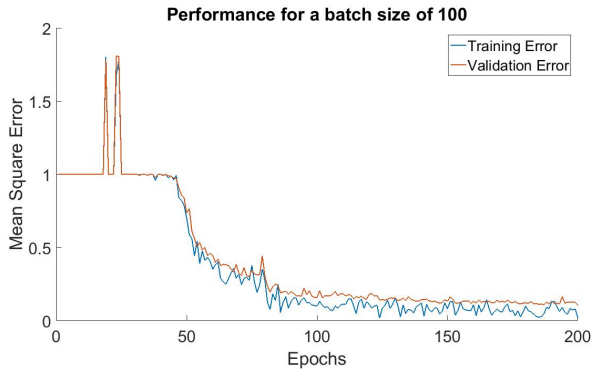

Fig. 4: Learning Curve for SGD with λ=0.7 and a hidden layer with 500 units

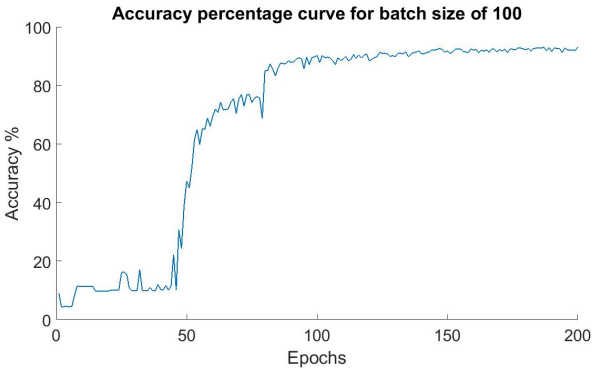

Fig. 5: Accuracy for SGD with λ=0.7 and a hidden layer with 500 units

TABLE II: Confusion Matrix for SGD with λ=0.7 and a hidden layer with 500 units

	0	1	2	3	4	5	6	7	8	9
0	963	0	0	2	0	4	5	1	5	0
1	0	1101	4	4	0	1	4	1	19	1
2	8	0	928	11	9	3	13	20	31	9
3	5	0	9	937	2	10	3	18	12	14
4	1	3	3	0	872	1	20	1	14	67
5	13	1	2	23	3	797	15	7	18	13
6	13	3	0	1	7	14	916	2	2	0
7	2	12	6	4	10	0	0	957	1	26
8	5	2	2	12	6	3	13	12	904	15
9	6	5	0	11	8	9	3	14	10	943

TABLE III: Performance for experiments using SGD

Hidden Units	Epochs	Learning Rate	Accuracy
400	150	0.5	91.96
400	200	0.5	92.82
500	150	0.7	92.90
500	200	0.7	93.18
600	200	0.7	91.78

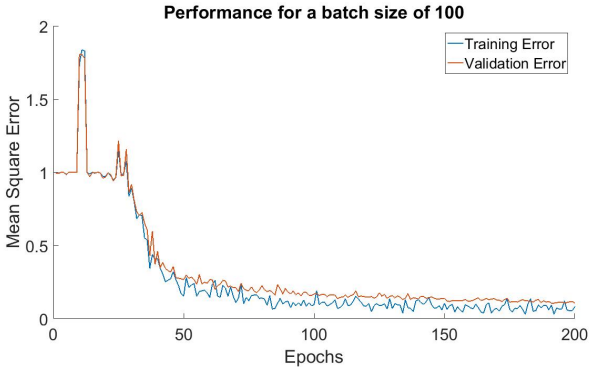

Fig. 6: Learning Curve for SGD-momentum with λ=0.1, α=0.8 and a hidden layer with 500 units

3 being predicted as 7 for 18 samples may not be explained with the same basis though. The performance for some other experiments using SGD are given in table III, with each epoch having a batch size of 100.

IV. SGD-Momentum

While working with non-linear functions, such situations may arise where our model gets stuck in a local minima while training, instead of getting to a global minima. In order to deal with such cases, we use a variant of SGD called SGD-momentum. The weight update in SGD momentum is done as follows:

$$\omega(n+1) = \omega(n) + \lambda\varepsilon(n)x(n) + \alpha(\omega(n) - \omega(n-1)) \quad (4)$$

where α is the regularization parameter for the momentum. The idea behind this is to give enough *momentum* to the weight update so as to push it beyond the local minima sink into potential global minima. The experiments for SGD-momentum were done for a single hidden layer with hidden units varying from 400 to 600. The learning curve and accuracy for the best result is shown in Fig. 6 and Fig. 7 respectively. Every epoch marks 100 samples, thereby leading to an experiment for 20,000 samples. The maximum accuracy obtained over test set in this case was 93.61%, which was not more than 0.5% improvement from the best result in SGD, for λ=0.1, α=0.8, and a hidden layer with 500 units. The confusion matrix for this case can be seen in table IV. The row indices denote the true labels and the column headers denote the predicted

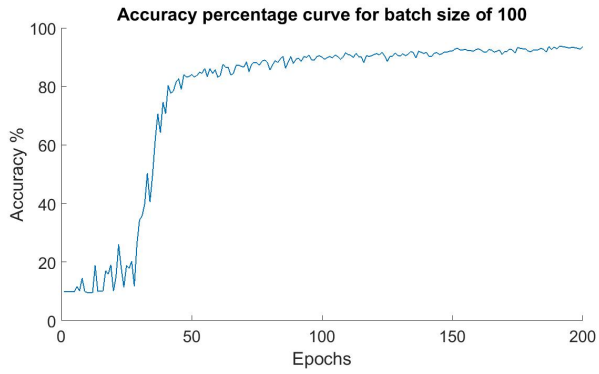

Fig. 7: Accuracy for SGD-momentum with λ=0.1, α=0.8 and a hidden layer with 500 units

TABLE IV: Confusion Matrix for SGD-momentum with λ=0.1, α=0.8 and a hidden layer with 500 units

	0	1	2	3	4	5	6	7	8	9
0	963	0	0	2	0	4	5	1	5	0
1	0	1100	4	4	0	1	4	1	20	1
2	8	0	928	11	9	3	13	20	31	9
3	5	0	9	937	2	10	3	18	12	14
4	1	3	3	0	872	1	20	1	14	67
5	13	1	2	23	3	797	15	7	18	13
6	13	3	0	1	7	14	916	2	2	0
7	2	12	6	4	10	0	0	957	1	26
8	5	2	2	12	6	3	13	12	904	15
9	6	5	0	11	8	9	3	14	10	943

labels. The results are obtained in this case are almost similar to the ones obtained in case of SGD, as there is less than 0.5% improvement.

The performance for some other experiments using SGD-momentum are given in table V. In table V, we can see that the performance of some of the cases in SGD-momentum are equivalent to those seen in Batch Gradient Descent. This may

TABLE V: Performance for experiments using SGD-momentum

Hidden Units	Epochs	Learning Rate	Regularization Parameter	Accuracy
400	150	0.5	0.1	92.78
400	150	0.1	0.8	92.6
400	200	0.3	0.8	11.4
500	200	0.1	0.8	93.61
500	200	0.1	0.9	9.82
500	200	0.3	0.8	9.74
600	200	0.1	0.8	91.25

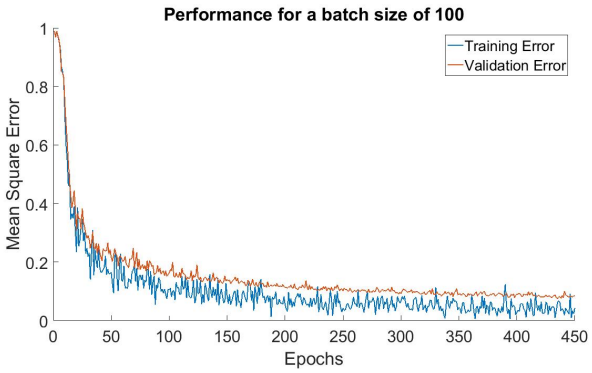

Fig. 8: Learning Curve for SGD-momentum with λ=0.1, α=0.8 downsampling by factor 2, and a hidden layer with 250 units

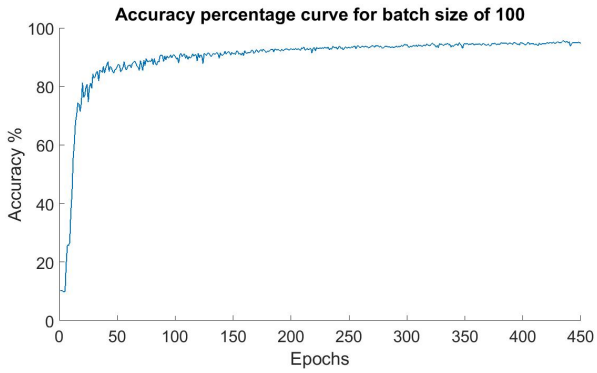

Fig. 9: Learning Curve for SGD-momentum with λ=0.1, α=0.8 downsampling by factor 2, and a hidden layer with 250 units

be because the learning rate and regularization parameter used in these cases are togther taking the network beyond a global minima or a good local minima to a poor local minima, and is then not able to go beyond that.

V. Improvements

We can further improve our model by decreasing the training time. This can be done by reducing the number of features being used for the model, which can be done by downsampling or PCA, and by stopping the training of the model once there is no further improvement in the model, known as early stopping.

A. Downsampling

Since each image is a vector of dimensions 784×1, it takes long to train our model. This is why so far, we have done experiments with a maximum of 20,000 training samples. In order to speed up the training process, we can decrease the number of features taken by downsampling our images. By downsampling an image vector by a factor *k*, we take the dimensions from the image vector that are multiples of *k*. Thus, for a downsampling by a factor of 2, we'll have dimensions which are multiples of 2 and hence each image vector would be reduced to a vector of dimensions 392×1. For our experiments, we have tried downsampling the images by scales of 2 and 4. After downsampling, the model was trained for the architecture that gives us highest accuracy in case of SGD-momentum, i.e. using λ=0.1 and α=0.8. As the performance of the network became faster, we have done experiments with epochs varying from 200 to 450. We have taken one hidden layer with hidden units varying from 125 to 300. Every epoch marks 100 samples, thereby leading to experiments for a maximum of 45,000 samples in training set. The maximum accuracy obtained over test set in this case was 95.45% for λ=0.1, α=0.8, a hidden layer with 250 units and 450 epochs. The learning curve for cost function for training and validation, and the accuracy on test set can be seen in Fig. 8 and Fig. 9 respectively. The performance of some other experiments after downsampling can be seen in table VI. We

TABLE VI: Performance for experiments using Downsampling with SGD-momentum

Factor	λ	α	Hidden Units	Epochs	Accuracy
2	0.1	0.8	125	450	93.61
2	0.1	0.8	250	200	92.88
2	0.1	0.8	250	250	93.27
2	0.1	0.8	250	300	94.1
2	0.1	0.8	250	350	95.01
2	0.1	0.8	250	450	95.45
2	0.1	0.8	300	350	94.35
2	0.1	0.8	300	400	94.99
4	0.1	0.8	125	200	88.09
4	0.1	0.8	125	400	89.46
4	0.1	0.8	250	400	89.23

can see from the table that by using downsampling, we can get equivalent or even better performance with lesser number of units in the hidden layer. As in third experiment in the table, we are getting an accuracy of 93.27%, with just 250 hidden units and 250 epochs, which is quite near the best performance obtained by using SGD-momentum with 500 hidden units and 200 epochs. Also, increasing the downsampling to a factor of 4 deteriorates the performance of the network in our case.

B. Validation-based Early Stopping

As we can see in our previous results, after a certain number of epochs, our training model becomes stable with no more major improvement in accuracy or decrease in cost. This shows that we don't need to train the model for so long and can stop early, once we observe that the model won't improve any further. This also helps us in generalization of the model and avoid overfitting. The method we followed for early stopping in our experiments is given in algorithm 2. In this algorithm, *patience* denotes the epochs that we skip before starting our checks for early stopping, $Cost^*_{val}$ represents the optimum

Algorithm 2 Validation-based Early Stopping

for each epoch j **do**
  Apply algorithm 1 for the jth training batch
  Skip EARLY STOPPING if j $\leq$ *patience*
  EARLY STOPPING
    $Cost^*_{val} \leftarrow$ Minimum validation cost after *patience*
    $\omega^* \leftarrow \omega_{Cost^*_{val}}$
    if j $\geq$ ($Cost^*_{val}$ *epoch* + *window size*) **then**
      Stop Training the model
      Return ω^*
    end if
end for

validation cost till epoch j after the patience level, $Cost^*_{val}$ *epoch* denotes the epoch giving $Cost^*_{val}$, *window size* denotes the number of epochs that we check for any improvement after finding the minimum validation error $Cost^*_{val}$, and ω^* denotes the optimum weights. In our experiments, we have kept a *patience* of 100 epochs, as we saw in our previous results that the model is taking more than 100 epochs to be stable, after which the model will start checking whether it should stop training on the basis of no further improvement in the cost function. We have a kept a *window size* of 20 epochs. Once the model's training goes beyond 100 epochs, it takes the minimum cost that it finds from 100 epochs to the current epoch, and then checks if there is any decrease in cost within next 20 epochs. If there is any decrease, the model stores that as the new minimum cost and checks for the next 20 epochs by repeating the previous step. If we don't find any improvement in validation cost within the window size, we stop the training and return the weights obtained from epoch corresponding to optimum cost as the optimum weights for the network.

We applied validation-based early stopping on the network architecture giving the best result in the previous experiments, that is for a downsampling of 2 factor, with λ=0.1, α=0.8, a hidden layer with 250 units, and 450 epochs. We get an accuracy of 95.30% using this method, with our network stopping the training at 360 epochs. Thus, it found the minimum cost at 360 epochs, checked for a lower cost till 380 epochs and not having found that, it stopped before reaching 450 epochs and thereby avoiding training on 7,000 extra samples. Thus, as we saw in table VI, the accuracy changes from experiment 5 to 6. However, with early stopping, our experiment doesn't run for 450 epochs but stops at 360 only, giving a similar accuracy. The learning curve and accuracy for this result can be seen in Fig. 10 and 11 respectively.

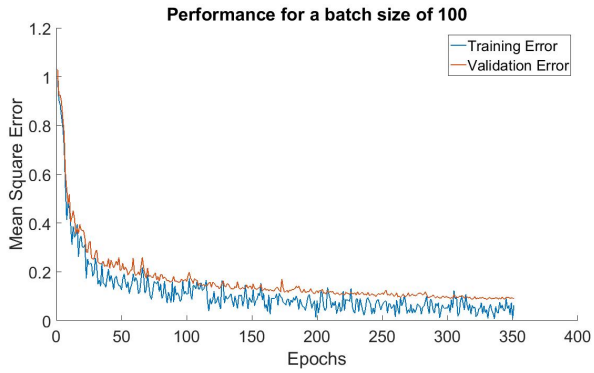

Fig. 10: Learning Curve for SGD-momentum with early stopping, λ=0.1, α=0.8 downsampling by factor 2, and a hidden layer with 250 units

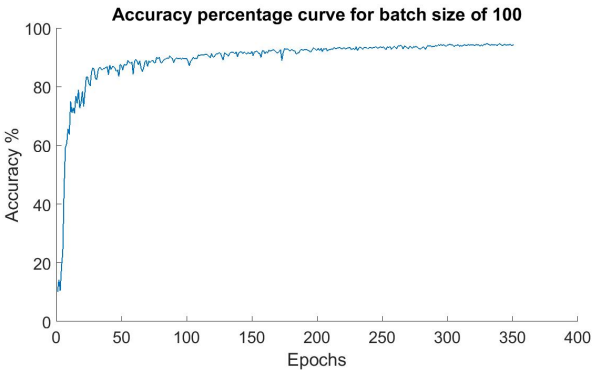

Fig. 11: Accuracy for SGD-momentum with early stopping, λ=0.1, α=0.8 downsampling by factor 2, and a hidden layer with 250 units

VI. CONCLUSIONS

We can draw a few conclusions from our experiments. We see that batch gradient descent may not be a useful way of training a neural network. We can use stochastic gradient to train our model by updating the weights after forward pass for each sample. We can further improve our accuracy by using optimum hyper parameters, to push our network from a local minima to the global minima. Also, we may improve the performance of our network in terms of both speed of training and accuracy by downsampling the inputs. Additionally, by using validation-based early stopping, we can train our model in less time by avoiding unnecessary when no improvement is seen in the network. Using all these method collectively, we can improve our performance in terms of speed of training the network as well as the accuracy.