Handwritten Digits Recognition with Multilayer Backpropagation Neural Networks

Shilin Zhu
Ph.D. student, Computer Science
UCSD
La Jolla, CA
shz338@eng.ucsd.edu

Yunhui Guo
Ph.D. student, Computer Science
UCSD
La Jolla, CA
yug185@eng.ucsd.edu

1 Abstract

In this report, we improve the handwritten digits recognition with multi-layer back-propagation neural networks. We correctly derived and implemented back-propagation algorithm and many tricks of the trade to improve the performance of learning. Furthermore, we try out different network topologies to select the best model based on the accuracy on validation set. The results are quite satisfying: using all the tricks of the trade and model selection strategy, we successfully achieve 97.51% on the final test set.

2 Classification

2.1 Mini-batch gradient descent

In this section, we use mini-batch gradient descent to classify the MNIST dataset. We split the 60000 images in the training set into two parts: the first 50000 images are used to train the model, the last 10000 images are used as validation set to do early stopping. We stop the training procedure once the loss on the validation set goes up and we save the weights that achieves the minimum loss on the validation set. And there are 10000 images in the test set.

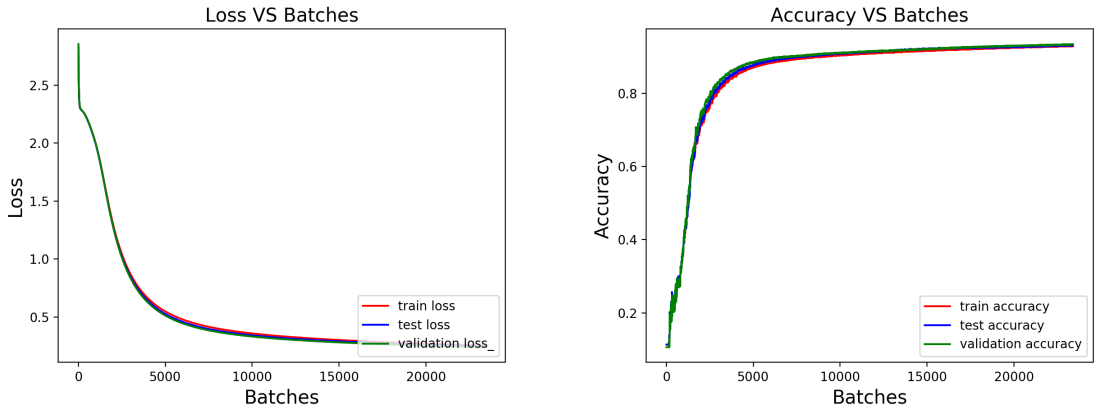
We use one hidden layer of 64 nodes, and the mini-batch size is 128. We use a learning rate of 0.01 and sigmoid activation function. We use standard normal distribution to initialize the weights and biases. For the weights, we multiply 0.01 to prevent large initialized values. We run the network for 60 epoches. We shuffle the dataset in each dataset.

We report the accuracy and loss on the training set, test se and validation set every batch. The following graphs show the accuracy and loss over each batch on different sets. Without any tricks, after 60 epoches, the accuracy on the test set is 0.9308%. This is no early stopping occurs, the possible reason is that the choice of learning is small

2.2 Gradient checking

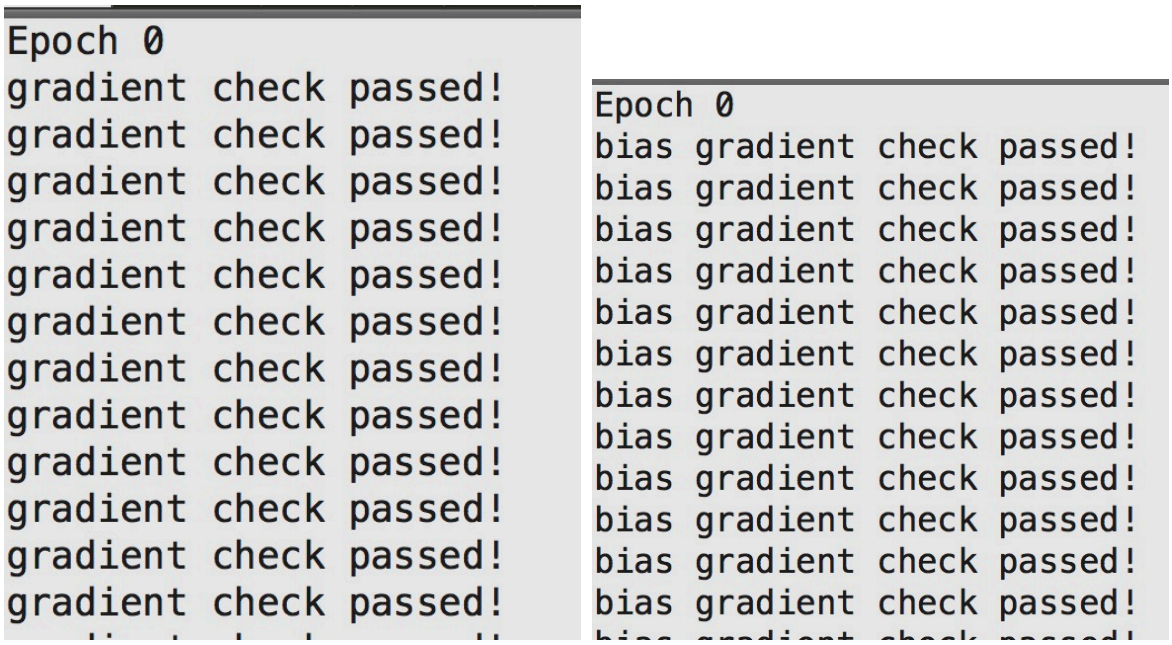
To verify the correctness of implementation of back-propagation, we compute the slope with respect to one weight using the numerical approximation: $\frac{\partial E^n}{\partial w_{ij}} \approx \frac{E^n(w_{ij}+\epsilon)-E^n(w_{ij}-\epsilon)}{2\epsilon}$ where we compute the numerical gradient for every weight and bias and for every example. Here we choose $\epsilon = 10^{-2}$ and according to the numerical theory, the difference between the gradients should be within $O(\epsilon^2)$ so that we expect the gradients to agree within 10^{-4}. The gradient checker on weights and biases has verified our backpropagation implementation as shown in Fig. 2.

After successfully verifying that our back-propagation implementation is correct, we turn off the numerical gradient checking when learning since it is way slower than back-propagation.



(a) (b)

Figure 1: The loss and accuacy of different sets over the batches without any tricks.

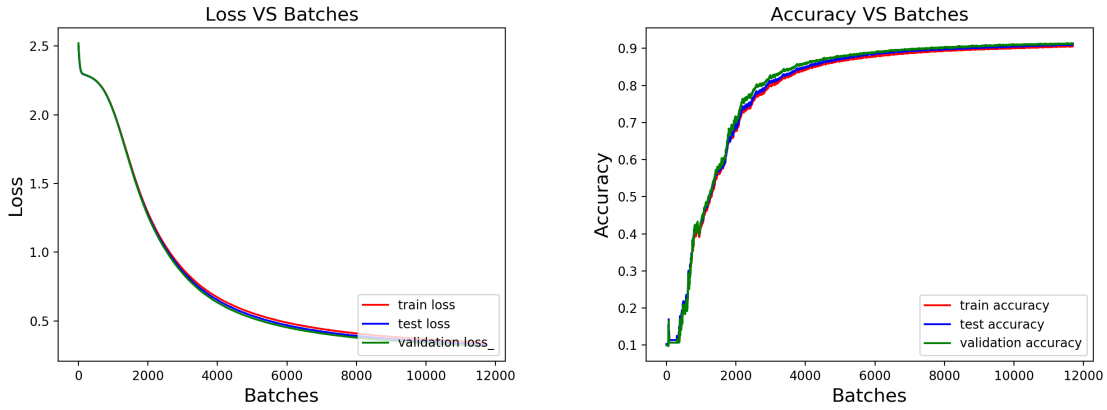


(a) (b)

Figure 2: Gradient checking results. (left) verification on weights. (right) verification on biases.

3 Adding the Tricks of the Trade

In this section, we follow Yann LeCun's paper [1] to understand and implement several tricks of the trade. In the next section we will add more tricks at the same time when we try out different network topologies. Here for simple comparison, we re-run the entire network for 30 epochs (where in the next section we will use 60 epochs) after we add each of these tricks to show the performance improvement after implementing each trick.



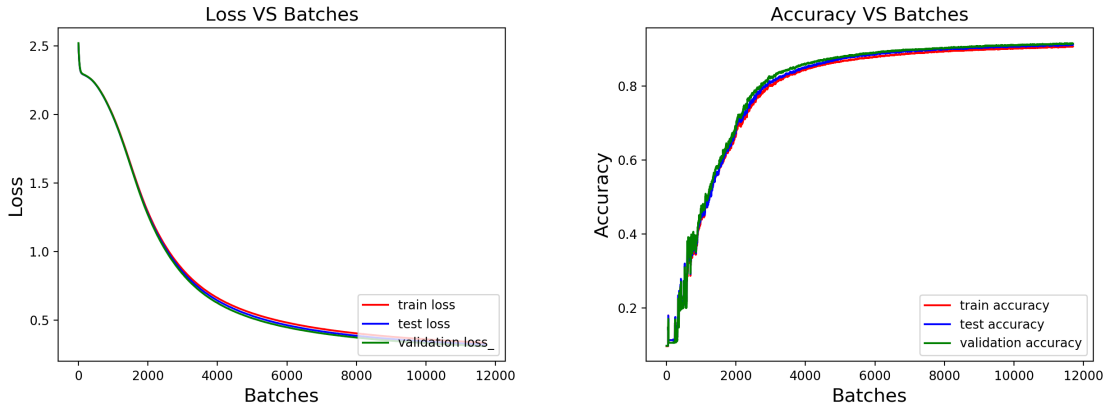
(a) (b)

Figure 3: The loss and accuacy of different sets over the batches without any tricks.

Fig. 3 shows the result on original vanilla version of the model without adding any tricks. The final test accuracy is 91.12% after 30 epochs.

3.1 Training data shuffling

According to [1], we should shuffle the training set so that successive training example rarely belong to the same class and present input examples that produce a large error more frequently than examples that produce a small error. Here we use mini-batches and we shuffle the examples after each epoch.



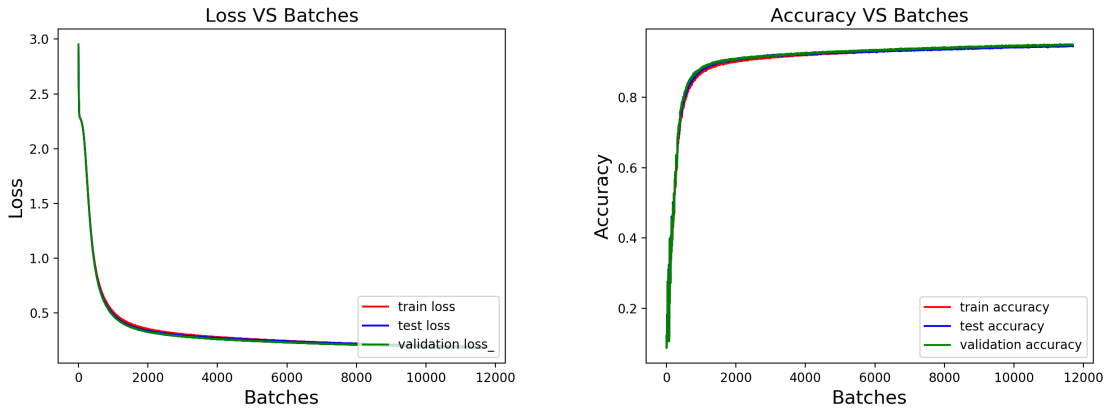
(a) (b)

Figure 4: The loss and accuacy of different sets over the batches with shuffling example trick.

Fig. 4 shows the result on the network after adding the shuffling example trick and re-run the learning process. The final test set accuracy is 91.26% after 30 epochs, showing some degree of improvement.

3.2 Activation function

The activation function highly affects the learning process including speed and ability of representation such as non-linearity so it is critical to set it correct. According to [1], we should use symmetric sigmoids such as hyperbolic tangent since it often converge faster than the standard logistic function. Here we use the recommended sigmoid $f(x) = 1.7159 * \tanh(2x/3)$ since it has several good properties: (a) $f(1) = 1, f(-1) = -1$, (b) the second derivative is maximum at $x = 1$ which can make good use of non-linearity, (c) the effective gain is close to 1.



(a) (b)

Figure 5: The loss and accuacy of different sets over the batches with shuffling example and special designed sigmoid tricks.

Fig. 4 shows the result on the network after adding the shuffling example and special designed sigmoid tricks and re-run the learning process. The final test set accuracy is 94.56% after 30 epochs, showing a large improvement of performance. This indicates that the activation function highly affects the learning process.

3.3 Weight initialization

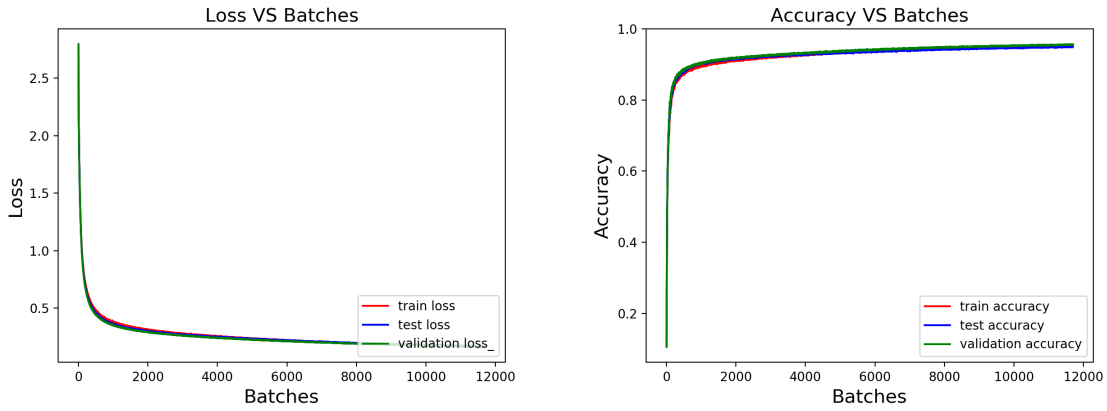
In order to improve and speed up the learning process, it is better to make the outputs of each node have mean zero and a standard deviation of approximately one. Assuming that the training set has been normalized and we use the previous modified sigmoid function, then we can derive that weights should be randomly drawn from a distribution with zero mean and standard deviation as $\sigma = m^{-1/2}$ where m is the number of connections feeding into the node.

Fig. 4 shows the result on the network after adding the shuffling example, special designed sigmoid, and fan-in weight initialization tricks and re-run the learning process. The final test set accuracy is 95.00% after 30 epochs, showing a small improvement of performance.

3.4 Momentum

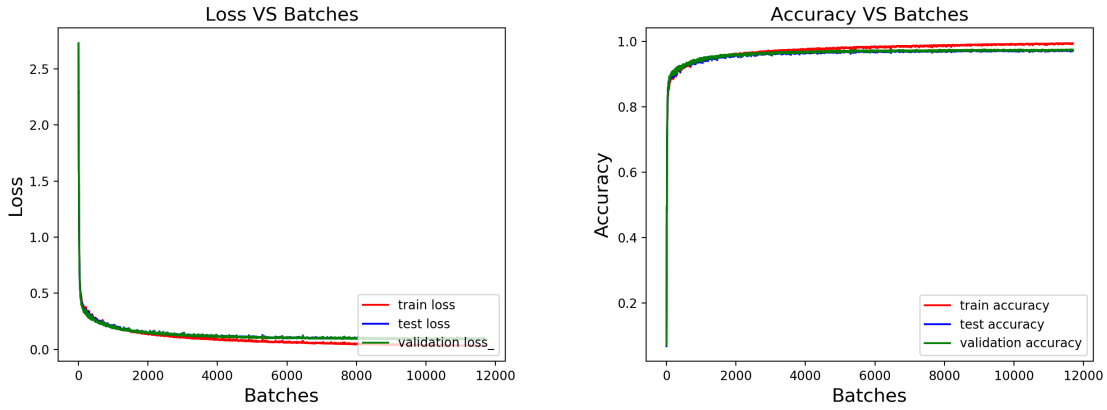
Momentum ($\Delta w(t+1) = \eta \frac{\partial E_{t+1}}{\partial w} + \mu \Delta w(t)$) can increase speed when the cost surface is highly non-spherical since it damps the size of steps along directions of high curvature thus yielding a larger effective learning rate along the directions of low curvature.

Fig. 4 shows the result on the network after adding the shuffling example, special designed sigmoid, fan-in weight initialization, and momentum tricks and re-run the learning process. The final test set



(a) (b)

Figure 6: The loss and accuacy of different sets over the batches with shuffling example, special designed sigmoid, and fan-in weight initialization tricks.



(a) (b)

Figure 7: The loss and accuacy of different sets over the batches with shuffling example, special designed sigmoid, fan-in weight initialization, and momentum tricks.

accuracy is 97.19% after 30 epochs, showing a large improvement of performance. This indicates that adding momentum can largely increase the speed of learning.

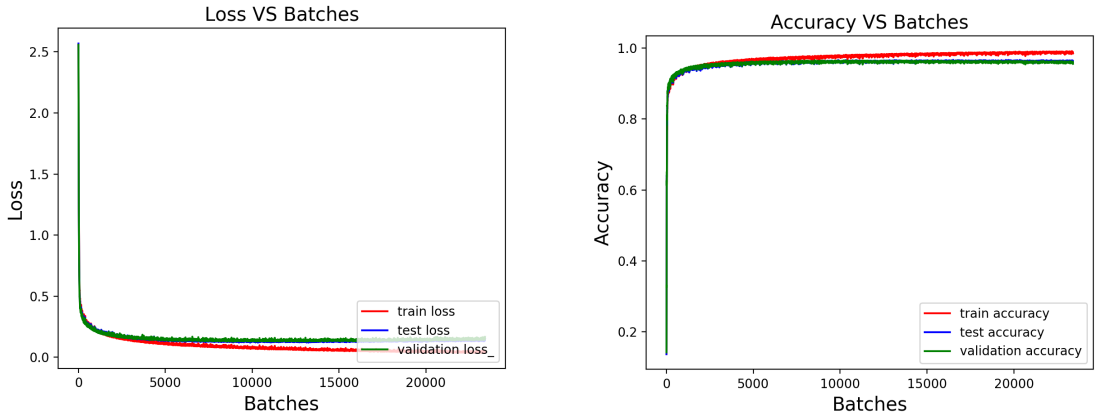
To sum up, these tricks can all help increase the learning speed so that it is essential to carefully consider the model of the network, initialization, activation function and optimization method.

4 Experiment with Network Topology

4.1 Experiments with differnet hidden units

We use a momentum of 0.9 and use the sigmoid in Section 4.4 of "lecun98efficient.pdf". The initialization method of weights are as described in 4 (c) in Programming assignment 2. Learning rate is 0.01.

First, we half the hidden units. Now we have a network of 3 layers with a hidden layer with 32 nodes. We run the network for 60 epoches except early stopping occurs. After 60 epoches, the accuracy on the test set is 0.9635%. This is no early stopping occurs, the possible reason is that the choice of learning is small enough.



(a) (b)

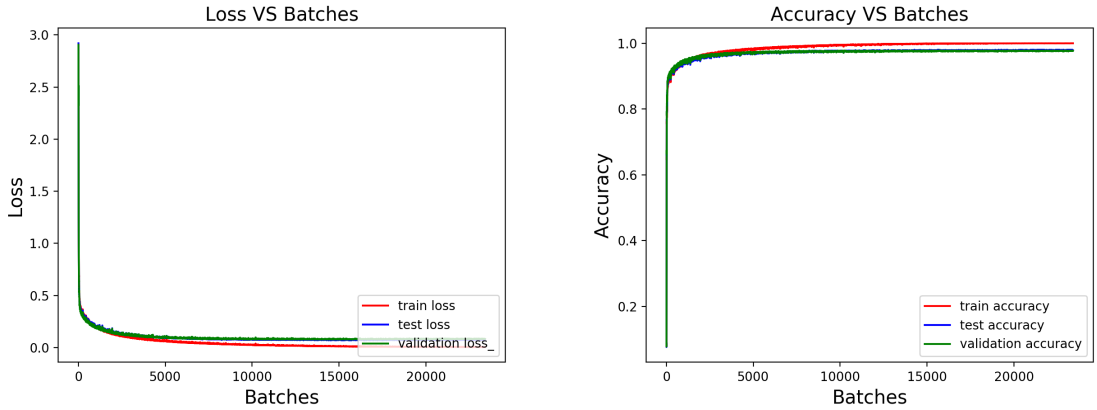
Figure 8: The loss and accuacy of different sets over the batches with a hidden layer of 32 hidden nodes.

Then, we double the hidden units. Now we have a network of 3 layers with a hidden layer with 128 nodes. We run the network for 60 epoches except early stopping occurs. After 60 epoches, the accuracy on the test set is 0.9787%. This is no early stopping occurs, the possible reason is that the choice of learning is small enough.

To further examine the influence of the numebr of hidden units, we decrease the hidden units to two nodes. We find that using two hidden units cannot capture the relation between different pixels. This suggests that if the number of hidden units is too small, we cannot get good results. If the number is too large, we can capture the information between pixels, but it can lead to overfitting and have much more parameters to tune.

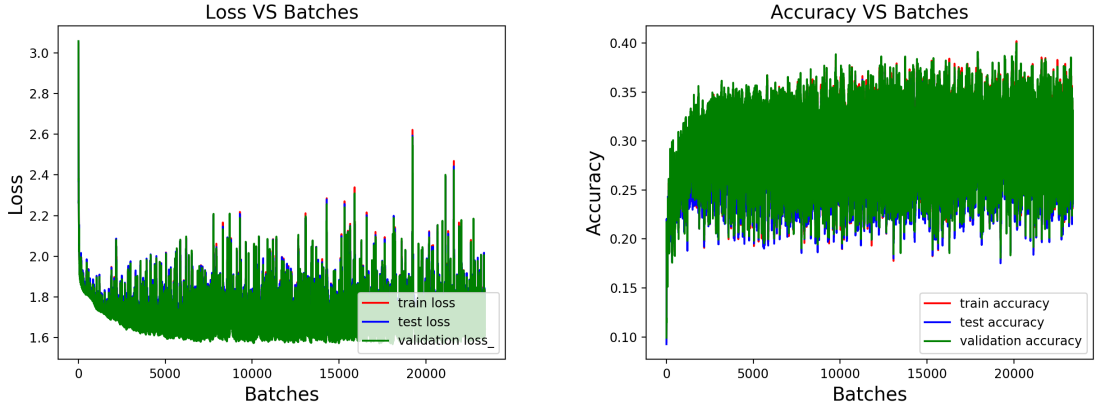
4.2 Doubling the hidden layers

For a network with one hidden of 64 nodes, there are approximately about 50890 parameters. If we increase the hidden layers while keep the same number of parameters, there will be 58 hidden nodes in each hidden layer. We use a momentum of 0.9 and use the sigmoid in Section 4.4 of "lecun98efficient.pdf". The initialization method of weights are as described in 4 (c) in Programming assignment 2. Learning rate is 0.01. After 60 epoches, the accuracy on the test set is 0.9766%. This is no early stopping occurs, the possible reason is that the choice of learning is small enough.



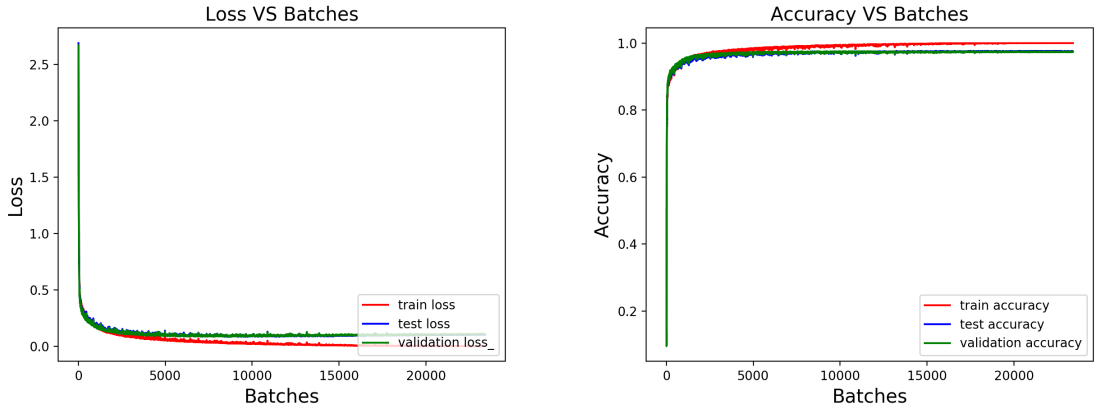
(a) (b)

Figure 9: The loss and accuacy of different sets over the batches with a hidden layer of 128 hidden nodes.



(a) (b)

Figure 10: The loss and accuacy of different sets over the batches with a hidden layer of 2 hidden nodes.



(a) (b)

Figure 11: The loss and accuacy of different sets over the batches with two hidden layers.

4.3 More tricks

In this section, in order to improve the preformance of the network, we consider the following tricks. In our experiments, we found that the network can achieve fast convergence and higher test accuracy with the tricks.

4.3.1 ReLU

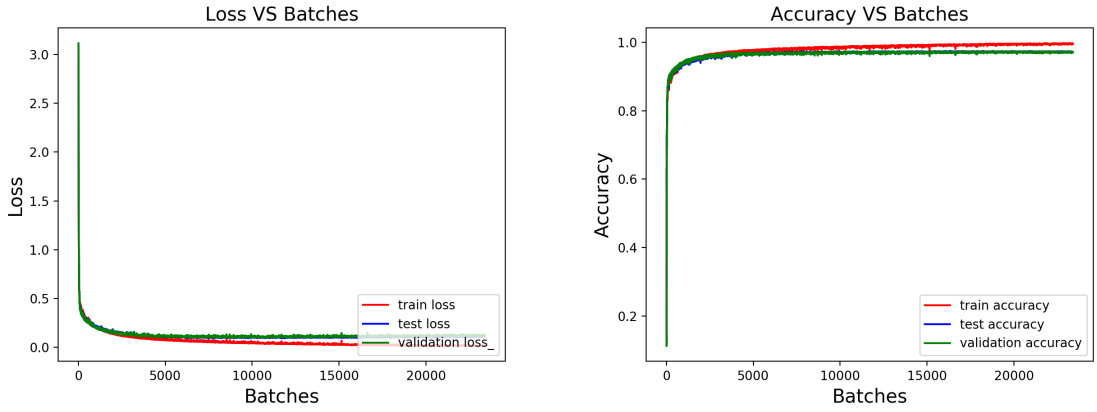
We consider using ReLU as the activation function. The ReLU function can be

$$\text{ReLU}(x) = \begin{cases} x & \text{if x} > 0, \\ 0 & \text{otherwise,} \end{cases}$$

The gradient of ReLU can be caculated as,

$$\text{dReLU}(x) = \begin{cases} 1 & \text{if x} > 0, \\ 0 & \text{otherwise,} \end{cases}$$

We use a three layer network with a hidden layer with 64 nodes. We use a momentum of 0.9 and use ReLU as activation function. The initialization method of weights are as described in 4 (c) in Programming assignment 2. Learning rate is 0.01.



(a) (b)

Figure 12: The loss and accuacy of different sets over the batches with ReLU.

4.3.2 Leaky ReLU

We consider using leaky ReLU as the activation function. The leaky ReLU function can be

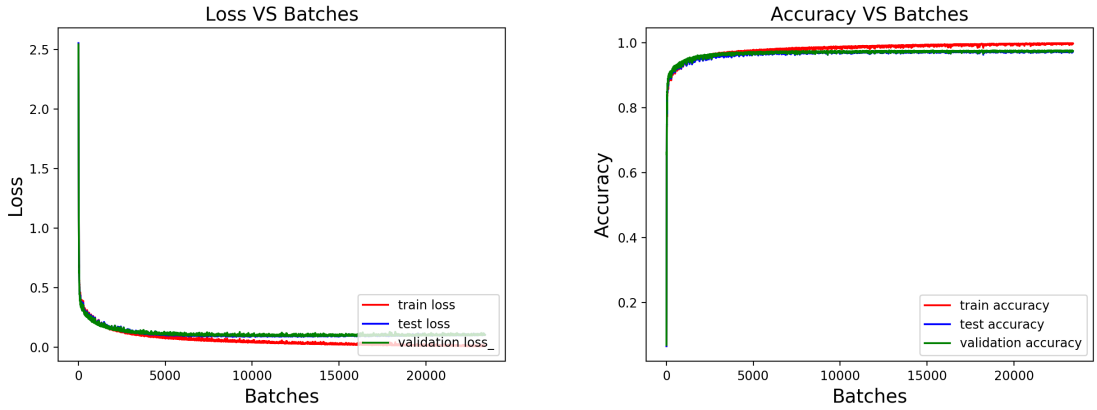
$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if x} > 0, \\ 0.01x & \text{otherwise,} \end{cases}$$

The gradient of leaky ReLU can be caculated as,

$$\text{dLeakyReLU}(x) = \begin{cases} 1 & \text{if x} > 0, \\ 0.01 & \text{otherwise,} \end{cases}$$

We use a three layer network with a hidden layer with 64 nodes. We use a momentum of 0.9 and use leaky ReLU as activation function. The initialization method of weights are as described in 4 (c) in Programming assignment 2. Learning rate is 0.01.

After 60 epoches, the accuracy on the test set is 0.9723%.

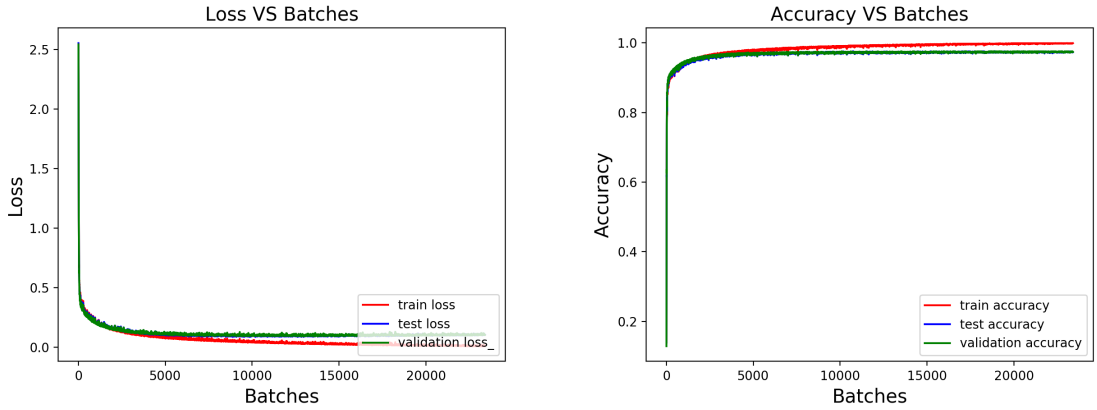


(a) (b)

Figure 13: The loss and accuacy of different sets over the batches with leaky ReLU.

4.3.3 Nesterov momentum

We consider using Nesterov momentum. We use a three layer network with a hidden layer with 64 nodes. We use a momentum of 0.9 and use leaky ReLU as activation function. The initialization method of weights are as described in 4 (c) in Programming assignment 2. Learning rate is 0.01. After 60 epoches, the accuracy on the test set is 97.34%.



(a) (b)

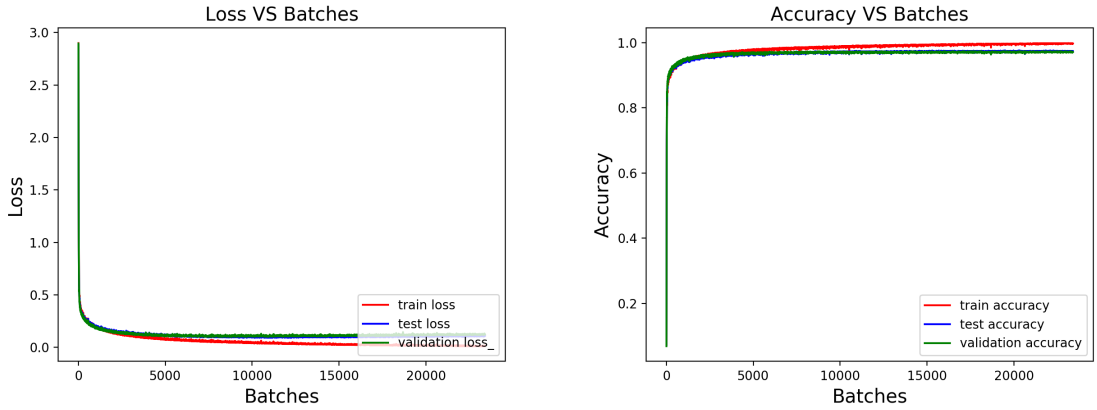
Figure 14: The loss and accuacy of different sets over the batches with Nesterov momentum.

4.3.4 Xavier initializtion

We consider using Xavier initializtion. We use a three layer network with a hidden layer with 64 nodes. We use a nesterov momentum of 0.9 and use leaky ReLU as activation function. The initialization method of weights are as described in 4 (c) in Programming assignment 2. Learning rate is 0.01. After 60 epoches, the accuracy on the test set is 97.51%.

5 Summary

In this work, we successfully derived and implemented multilayer backpropagation neural networks based handwritten digits recognition. We achieve a test accuracy around 97.5%. We experimented



(a) (b)

Figure 15: The loss and accuacy of different sets over the batches with Xavier initializtion.

with different tricks and different network topology. Many of the tricks improve the performance and the convergence speed. This makes us better understand the training process of the backpropagation. We found that use ReLU and add momentum can greatly improve of performance of the network. Overall, we learnt a lot from this assignment and understand backpropagation better.

6 Contributions

Shilin Zhu did the implementation of gradient checking in Section 3.d, added the tricks of the trade, did corresponding experiments, and write corresponding parts report in Section 4.

Yunhui Guo did the implementation of mini-batch gradient descent in Section 3(a,b,c,e), experimented with different network topologies, and write corresponding parts report in Section 5.

Three discussions and pair programming were made before submitting this report.

Acknowledgments

We would like to thank Prof. Gary Cottrell and all TAs' efforts in preparing and grading this assignment.

References

[1] LeCun, Y., Bottou, L., Orr, G. B., and Mller, K. R. (1998). Efficient backprop. In Neural networks: Tricks of the trade (pp. 9-50). Springer, Berlin, Heidelberg.