Image Classification Using Multilayer Perceptron

Yongchuang Huang
PID: A53322973
yoh001@eng.ucsd.edu

Jingpei Lu
PID: A91033661
jil360@eng.ucsd.edu

Yiran Xu
PID: A53270264
y5xu@eng.ucsd.edu

Abstract

In this report, we use Multilayer Perceptron via Stochastic Gradient Descent with momentum to optimize our model on Fashion MINST dataset. Also, to achieve a better performance, L_2-Regularization and different activations are tried. Finally, we achieved 88.08% accuracy on the test set with one 50-unit hidden layer.

1 Introduction

Multilayer Perceptron (MLP) is a deep neural network that can be used for image classification. Different from single layer perceptron, MLP have hidden layers between input layer and output layer, which involves more parameters for feature representation. On the other hand, however, similar to single layer perceptron, Stochastic Gradient Descent (SGD) is usually applied to optimize the objective function and update parameters. It is also common to use momentum to accelerate the convergence of SGD. Moreover, to better generalize the model, L_2-regularization term is considered as a penalty term to reduce the the strength of parameters and to avoid overfitting.

2 Background

Classic image classification methods include Support Vector Machine (SVM) [3], Decision Tree [4], MLP-based networks and CNNs [5]. Since large ImageNet [6] challenge came out, deep learning algorithms have been placed in a dominant position.

However, optimization and overfitting are still two concerns for deep learning. To accelerate convergence, many optimizers are proposed, e.g. SGD with momentum [7], Rmsprop [7], Adam [8]. To get rid of overfitting, regularization is a common approach to mitigate the complexity of the model.

3 Methods

3.1 Three-layer Network

We start from a Three-layer network, that is an MLP consisting of an input layer, a hidden layer and an output layer. Suppose the dimensionality of the output is K, of the hidden layer is J and of the input is I, the weight matrix from the hidden layer to the output layer is $W_2 \in \mathbb{R}^{K\times J}$ and the weight matrix from the input layer to the hidden layer is $W_1 \in \mathbb{R}^{J\times I}$, as well as bias vectors $\mathbf{b}_1 \in \mathbb{R}^J$ and $\mathbf{b}_2 \in \mathbb{R}^K$, the input $X \in \mathbb{R}^{I\times N}$, the targets $T \in \mathbb{R}^{K\times N}$, where N is the number of inputs. The forward propagation is

$$A_1 = W_1 X + \mathbf{b}_1 \mathbf{1}^\top \in \mathbb{R}^{J\times N} \tag{1}$$

$$H_1 = \text{Activation}(A_1) \in \mathbb{R}^{J \times N} \tag{2}$$

$$A_2 = W_2 H_1 + \mathbf{b}_2 \mathbf{1}^\top \in \mathbb{R}^{K \times N} \tag{3}$$

$$Y = \text{softmax}(A_2) \in \mathbb{R}^{K \times N} \tag{4}$$

where Activation(), softmax() are element-wise functions, $\mathbf{1}$ is a vector with all ones. We can rewrite the backward update rules in matrix/vector notation as follow:

$$W_2 = W_2 + \alpha(T - Y)H_1^\top \tag{5}$$

$$\mathbf{b}_2 = \mathbf{b}_2 + \alpha(T - Y)\mathbf{1} \tag{6}$$

$$W_1 = W_1 + \alpha W_2^\top (T - Y) \odot \text{Activation}'(A_1) X^\top \tag{7}$$

$$\mathbf{b}_1 = \mathbf{b}_1 + \alpha W_2^\top (T - Y) \odot \text{Activation}'(A_1)\mathbf{1} \tag{8}$$

where $\odot$ is element-wise multiplication and $\mathbf{1}_m$ is the matrix full of 1.

3.2 SGD with momentum

To accelerate the convergence of optimization, momentum is added. The update can be written as

$$v_{t+1} = \mu v_t + g_{t+1}, \tag{9}$$

$$w_{t+1} = w_t - \alpha v_{t+1}, \tag{10}$$

where w, v, g, μ denotes the parameters, velocity, gradients and momentum respectively. Usually the velocity v is initialized at zero. By involving momentum, the parameter vector will build up in any direction that has consistent gradient [2].

3.3 L_2-Regularization

With L_2-Regularization, our objective function can be written as:

$$J(W) = \sum_n E(t^n, y^n; W) + \lambda \sum_i ||W_i||_2^2, \tag{11}$$

where W is the weight matrix, and λ is a scalar to control the strength of penalty term. Then our goal becomes minimize Eq. 11. By taking the derivative of Eq. 11, we know that the only thing different from Eq. 7 and Eq. 5 is that we have an additional term w.r.t the L_2 norm, thus we have:

$$W_2 = W_2 + \alpha((T - Y)H_1^\top - 2\lambda W_2) \tag{12}$$

$$W_1 = W_1 + \alpha(W_2^\top (T - Y) \odot \text{Activation}'(A_1) X^\top - 2\lambda W_1) \tag{13}$$

for update.

4 Experiments

4.1 Setup

The hidden layer consists of 50 units and this is applied to Section 4.2 -Section 4.5. The activation function is tanh() in Section 4.2 and Section 4.3. The parameters are initialized with `np.random.randn(num_in, num_out)*sqrt(2.0/num_in)` as [1] suggests. The batch size is 256, the model is trained for 100 epochs. The optimizer is SGD with momentum 0.9. Early stop criteria with threshold, 5 epochs, is used. Also, to achieve a stable and accurate optimal location, the learning rate α is initialized and is divided by 10 every 20 epochs after the 60-th epoch.

For Fasion MINST Dataset, we first shuffled and then split 10,000 images out of 60,000 images as our validation set. The validation loss is based on validation set.

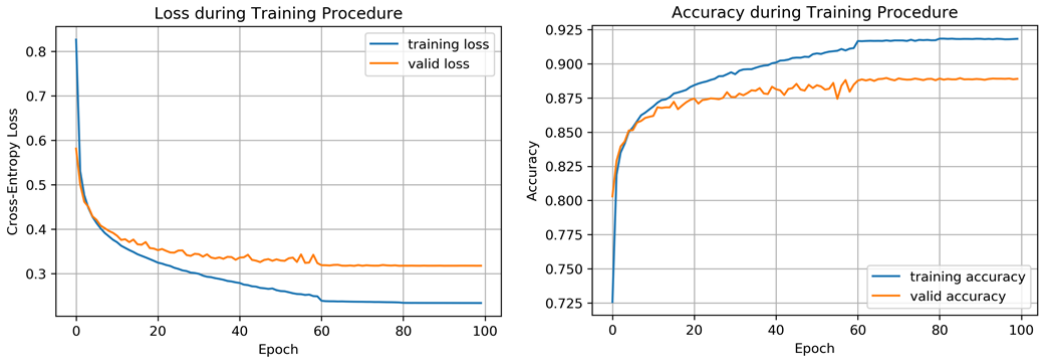

Figure 1: Loss (left) and accuracy (right) during training procedure.

4.2 Gradient Check with Numerical Approximation

To ensure the correctness of the gradient computing in our code, we computed the slope with respect to one weight using the numerical approximation:

$$\frac{d}{dw}E(w) \approx \frac{E(w+\epsilon) - E(w-\epsilon)}{2\epsilon} \tag{14}$$

As shown in Tabel 1, we have obtained two input to hidden weights, two hidden to output weights, one hidden bias and one output bias by performing our gradient-checking code, compared with the corresponding gradient results obtained by backpropagation. We used 10 examples, one from each category, as our checking dataset, and took $\epsilon = 0.01$. From the table, we can tell that the difference of the gradients obtained by these two ways is within big-O of ϵ^2, that is, 10^{-4}. Thus, our backpropagation is correct.

Table 1: Gradient checking results of $\epsilon = 0.01$

Parameters	**Numerical Approximation** $\Delta_\epsilon w$	**Backpropagation** dw	**The Difference:** $\lvert\Delta_\epsilon w - dw\rvert$
input_hidden_weight_1	-0.7946979106260299	-0.7946977316365482	$\approx 1.8 \times 10^{-7}$
input_hidden_weight_2	0.04437860331552024	0.04437847015382271	$\approx 1.3 \times 10^{-7}$
hidden_output_weight_1	-0.6083407776086291	-0.6083363651794743	$\approx 4.4 \times 10^{-6}$
hidden_output_weight_2	0.8439508755225811	0.8439512430038645	$\approx 3.6 \times 10^{-7}$
hidden_bias	-0.8617042853686918	-0.8616297615414158	$\approx 7.5 \times 10^{-5}$
output_bias	0.9734169711589358	0.9734794226098672	$\approx 6.3 \times 10^{-5}$

4.3 Experiment with Vanilla Three-layer MLP

We report our loss and accuracy curves under the learning rate $\alpha = 0.01$, which achieves the best performance on the test set. **The final accuracy is 87.81%.** From Figure 1, we can see that the training loss drops dramatically at the beginning, and then decreases slowly. However, after the 60-th epoch, when the learning rate is decreased, the training loss decreases again and converges. This indicates that the model is trapped at sub-optimal point when α is big. Fortunately, as the learning rate decreases, the model gets convergent after the 60-th epoch. From validation loss and accuracy, we can see that the model is slightly overfitting.

4.4 Experiment with Regularization

We trained the model with L_2 penalty term and used $\lambda \in \{0.001, 0.0001\}$ to control the penalty strength. The loss and accuracy curves are shown in Figure 2 and Figure 3. **The final test set for $\lambda = 0.001$ is 87.41% and for $\lambda = 0.0001$ is 87.62%.**

As we can see from Figure 2 and Figure 3, both the curves are nearly the same. The reason for this is a) with appropriate α and initialization, the model is convergent completely, and b) with small λ,

the strength of regularization is trivial. This can also be explained by the test accuracy, since it does not vary much compared to Section 4.3 (87.81%).

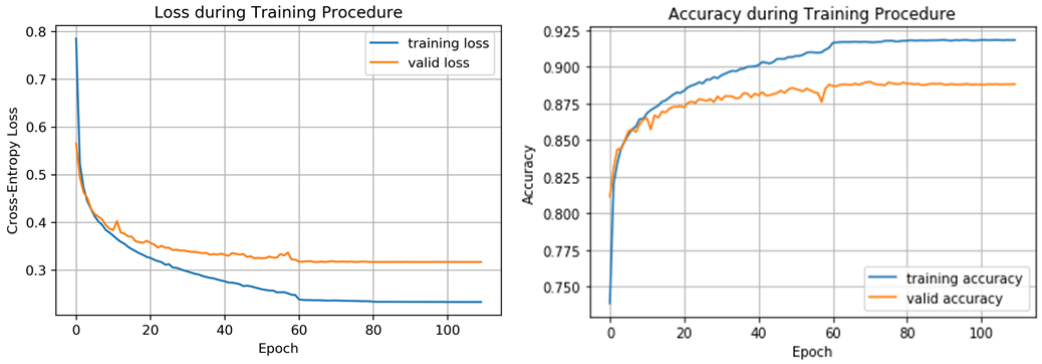

Figure 2: Loss (left) and accuracy (right) during training procedure while $\lambda = 0.001$.

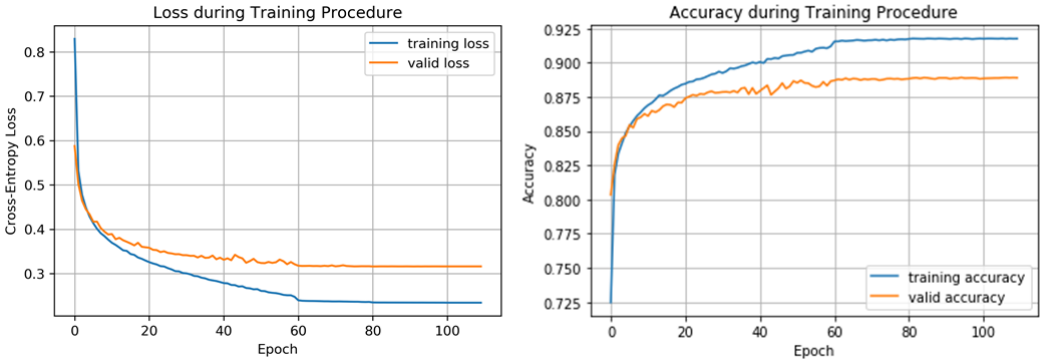

Figure 3: Loss (left) and accuracy (right) during training procedure while $\lambda = 0.0001$.

4.5 Experiment with Activations

Experiment with Sigmoid. By our experiment, the best learning rate for Sigmoid function is $\alpha = 0.1$. We report loss and accuracy curves in Figure 4. **The final test accuracy is 87.89%**, which is slightly better than tanh. Compared to tanh function, the training loss achieves lower value, while the validation loss is nearly the same. From Figure 4 Sigmoid function causes more serious overfitting than tanh function under the best learning rate.

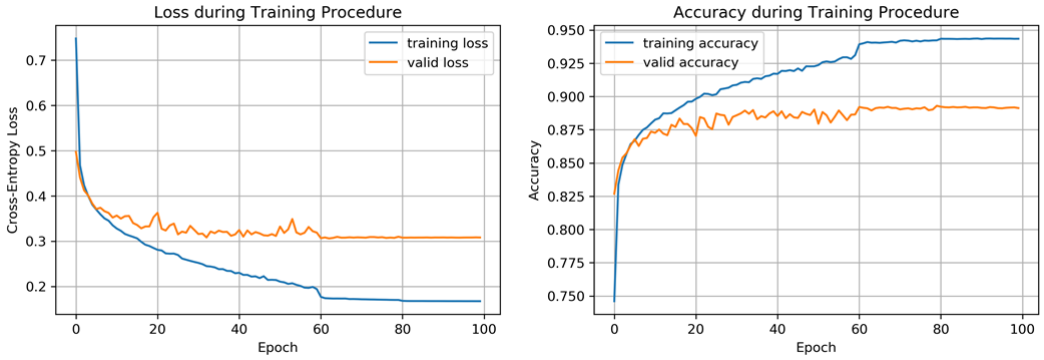

Figure 4: Loss (left) and accuracy (right) during training procedure with Sigmoid.

Experiment with ReLU. By our experiment, the best learning rate for ReLU activation function is $\alpha = 0.01$. We report loss and accuracy curves in Figure 5. **The final test accuracy is 87.78%**, which is slightly worse than other activation functions. Both training loses and validation loses are higher than using sigmoid, while is similar to tanh.

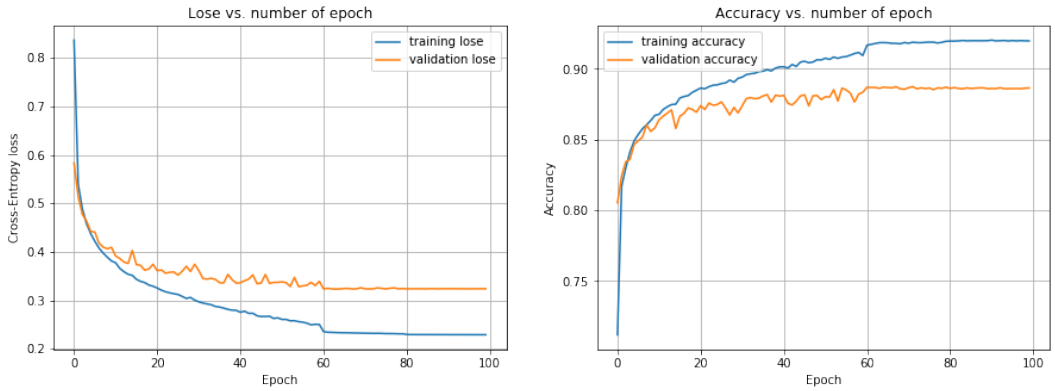

Figure 5: Loss (left) and accuracy (right) during training procedure with ReLU.

From our experiments, we found that by using the sigmoid activation function, the network converges slower but is able to achieve higher accuracy with larger learning rate. ReLU and tanh make the network converge faster but the drawback is easily overshooting.

4.6 Experiment with Network Topology

Experiment with doubling the number of hidden units. By our experiment, doubling the number of hidden units improves the performance. We report loss and accuracy curves in Figure 6. **The final test accuracy is 88.08%**, which is slightly better than the vanilla network. Although they are converging in a nearly same rate, more parameters allow the network to be trained on more epoches without suffering from overfitting.

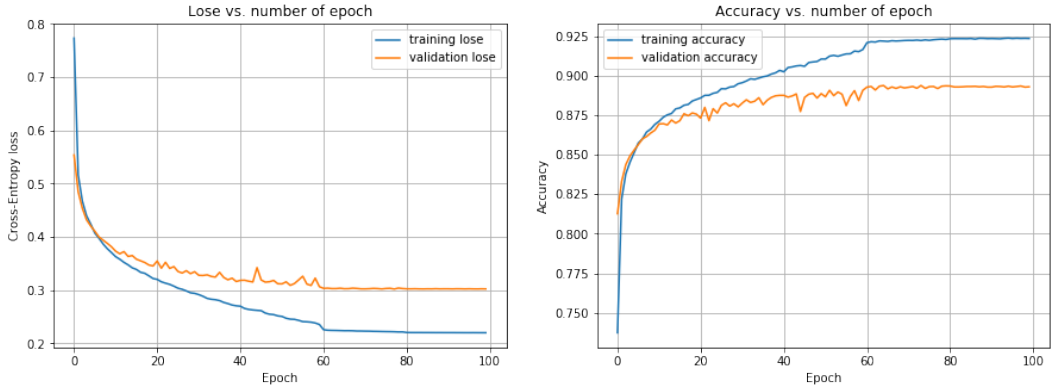

Figure 6: Loss (left) and accuracy (right) of the network with 100 hidden units.

Experiment with two hidden layers. By our experiment, adding more hidden layers also improves the performance. We report loss and accuracy curves in Figure 7. **The final test accuracy is 88.04%**, which is slightly better than the vanilla network. In this experiment, we have two hidden layers with each has 47 hidden units, which makes the total number of parameters approximately equal to the vanilla network. By observing the lose and accuracy curves, more hidden layers allows the network converge faster and to a lower lose and higher accuracy.

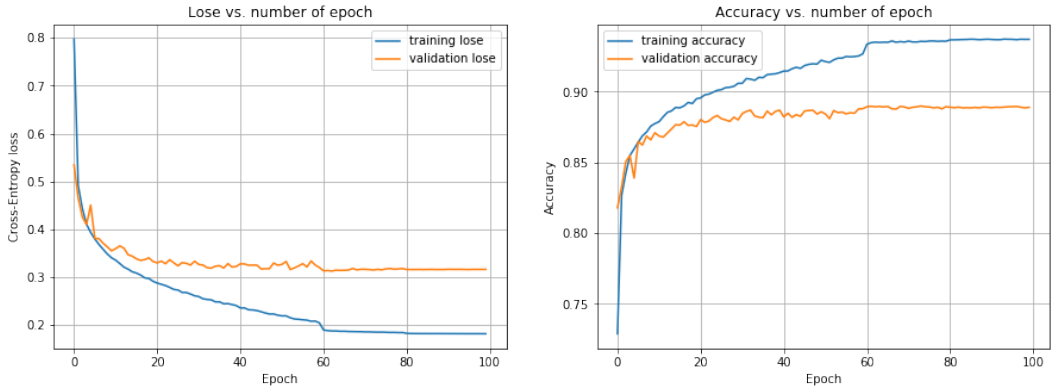

Figure 7: Loss (left) and accuracy (right) of the network with two hidden layers.

5 Contributions

Yongchuang Huang finished the code for data preprossing, layer class, activation class and gradient check. He also wrote Section 4.2. for this report.

Yiran Xu finished the part of SGD optimizer (including momentum) and L_2-Regularization. He also wrote Abstract, Introduction, Backgroud, Methods, Section 4.3 and Section 4.4 in this report.

Jingpei Lu finished the experiments and report of different activation functions and network topology.

References

[1] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the IEEE international conference on computer vision (pp. 1026-1034).

[2] CS231n Course Notes: http://cs231n.github.io/neural-networks-3/sgd

[3] Cortes, C., & Vapnik, V. (1995). Support-vector networks. Machine learning, 20(3), 273-297.

[4] Kamiński, B., Jakubczyk, M., & Szufel, P. (2018). A framework for sensitivity analysis of decision trees. Central European journal of operations research, 26(1), 135-159.

[5] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. nature, 521(7553), 436-444.

[6] Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009, June). Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition (pp. 248-255). Ieee.

[7] Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

[8] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.