USE DEEP NEURAL NETWORKS IN TRAINING IMAGES IN FASHIONMNIST DATASET

EDRIA LI

Applied Mathematics Department, University of Washington, Seattle, WA
yijueli@outlook.com

Abstract. We successfully trained the Fully Connected Deep Neural Networks (FCNs) to distinguish images in the FashionMNIST dataset. ReLU function was used to activate each layer. Cross Entropy Loss function was used as the target function to perform the classification. Several optimizers such as SGD, RMSprop, and Adam were implemented. We also performed dropout regularization, and considered different initializations and normalizations to see their effects on training. We started our baseline configuration with test accuracy of 85%; after hyperparameter tuning from the baseline, 90% accuracy rate were achieved.

1. Introduction and Overview

We are provided with FashionMNIST data with 60000 samples in training and 10000 samples in test sets. The training set is further split to training (90%) and validation (10%) where each image is represented in pixels of size 28×28.

Our task is to design a Fully Connected Deep Neural Network (FCN) model to classify the dataset. The model need to have an adjustable number of hidden layers and neurons in each layer with ReLU activation. Cross Entropy Loss function were used for training. Different optimizers such as Stochastic Gradient Descent (SGD), Adam, RMSprop were tried; multiple initialization methods such as Random Normal, Xavier, and Kaiming initialization were also examined. For training purpose, data were passed in through batches with batch size 512. We started with baseline configuration (test accuracy above 85%) using SGD optimizer with learning rate of 0.025. The number of layers, number of neurons in each layer, learning rate, number of training epochs, optimizers, initialization methods and batch normalization methods were all designed into adjustable parameters. Validation accuracy and loss function were visualized to aid our decision on hyperparameter tuning. The model was fined tuned to achieve 90.5469% testing accuracy on FashionMNIST and 98.3203% on MNIST.

2. Theoretical Background

2.1. **Deep Neural Network Basics.** Neural Networks have their names loosely inspired by neurobiology with similar structure modeling neurons [1]. One key design consideration for neural networks is the architecture, which refers to the overall structure of the network. The neural networks are organized into groups of units called layers, which are further arranged in a chain structure, where the first layer is $\mathbf{y}^1 = f^1(\mathbf{W}^{1\top}\mathbf{x}^1 + \mathbf{b}^1)$, and the second layer $\mathbf{y}^2 = f^2(\mathbf{W}^{2\top}\mathbf{y}^1 + \mathbf{b}^2)$ and so on, where weights $\mathbf{W}^\top\mathbf{x}$ can be elaborated itemwise in $\sum_{i=1}^n x_i w_i$. For forward propagation, the activation function as denoted above f can be represented in many forms, such as(1) Sigmoid function: $\sigma = f = \frac{1}{1+e^{-x}}$; (2) tanh function: $f = \tanh(x)$; (3) Rectified Linear Unit function (ReLU): $f = \max(x, 0)$; or (4) Leaky parametric ReLU: $f = \max(x, ax)$. ReLU is a default activation function recommended for use with most feedforward neural networks [1].

2.2. **Cross Entropy Loss and Gradient Descent.** Cross entropy is defined as $H(P, Q) = \mathbb{E}_{x\sim P}[-\log Q(x)]$ with P denotes distribution given to us where as Q denotes distribution we trained from model Q. And loss function is defined as $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$ where $\hat{y}$ denotes prediction and y is given. The accumulated loss can be combined into cost function which coins the term Cross-Entropy Loss: $\mathcal{J}^m(y, \hat{y}; x) = \frac{1}{m}\sum_{i=1}^m \mathcal{L}^i(\hat{y}, y)$ where index i traverse through each sample and m is the total number of samples. Gradient descent is than used as an iterative approach to optimize the given function $\mathcal{J}$. The weights

and bias can be updated as $\mathbf{W}_{k+1} = \mathbf{W}_k - \alpha\nabla_{\mathbf{W}}\mathcal{J}(\mathbf{W}_k; \mathbf{b})$ and $\mathbf{b}_{k+1} = \mathbf{b}_k - \alpha\frac{\partial}{\partial \mathbf{b}}\mathcal{J}(\mathbf{W}; \mathbf{b}_k)$ respectively, where α denotes learning rate.

2.3. **Regularization.** Regularization terms can be further added to the cost function $\mathcal{J}$ as the follows: $\mathcal{J}^m(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}^i(\hat{y}, y) + \frac{\lambda}{2m}\sum_{i=1}^{L}\|\mathbf{w}^l\|_F^2$ where $\|\mathbf{w}^l\|_F^2 = \sum_{i=1}^{l}\sum_{j=1}^{l}(\mathbf{w}_{ij}^l)^2$, i.e. the squared sum of all weight terms. Dropout regularization provides a computationally inexpensive but powerful method in large neural networks [1], and prevents the co-adaptation of neurons of feature detectors by randomly zeroes some input elements with probability p chosen independently for each forward call and sampled from Bernoulli distribution [2].

2.4. **Initialization.** Initialization strategies in neural network optimization is not yet well understood, and typically we set the biases for each unit to heuristically chosen constants and initialize only the weights randomly to "break symmetry" between different layers [1]. Random Normal initializes the inputs with values drawn from normal distribution $\mathcal{N}(\mu,\, \sigma^2)$; Xavier Normal uses distribution sampled from $\mathcal{N}(0, \sigma^2)$ where $\sigma = \text{gain} \times \sqrt{\frac{2}{\texttt{fan\_in}+\texttt{fan\_out}}}$; and Kaiming Uniform uses distribution that have values sampled from $\mathcal{U}$(-bound, bound) where bound = gain $\times\sqrt{\frac{3}{\texttt{fan\_mode}}}$ where `fan_in` preserves the magnitude of the variance of the weights in the forward propagation whereas `fan_out` preserves magnitudes in the backward propagation [2].

2.5. **Batch Normalization.** Batch normalization is an adaptive reparametrization method used especially in very deep models [1]. Let $\mathbf{H}$ be a minibatch of activations of the layer to normalize, to normalize $\mathbf{H}$, we replace it with $\mathbf{H}' = \frac{\mathbf{H}-\mu}{\sigma}$, where at training time $\mu = \frac{1}{m}\sum_i \mathbf{H}_i$ and $\sigma = \sqrt{\delta + \frac{1}{m}\sum_i(\mathbf{H}-\mu)_i^2}$. We backwardly propagate through these operations for computing mean μ and standard deviation σ and for applying them to normalize $\mathbf{H}$ [1]. Practically we normalize inputs through formula $y = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}\cdot\gamma + \beta$, where mean μ and variance σ^2 are calculated over the minibatches and γ and β are learnable parameter vectors [2].

3. Algorithm Implementation and Development

we used `Python` in which we import `numpy`, `torch`, and `torchvision` for computing, `sklearn.model_selection` for data split; we also import `matplotlib` and `seaborn` for plotting.

Here are the detailed algorithms for setting up FCN Model (Algorithm 1) and Train/Test call function (Algorithm 2):

4. Computational Results

4.1. **Baseline Configuration.** We first setup our baseline configuration using a 3 hidden-layer FCN (each layer has nodes 1200, 2000, and 800, respectively), `SGD` as optimizer with learning rate .025, and trained in 20 epochs. The training started with validation accuracy of 68.0098% and plateaued around 87% after 15 epochs. We can observe the training loss function dropped down until the 15th epoch with a slight trend of increase at the end. A test accuracy of 86.0742% was achieved. We will use this FCN layers and number of epochs as our baseline configuration.

4.2. **Optimizers with Different Learning Rate.** We experimented SGD, RMSprop, and Adam with different sets of learning rate. After numerous experiments, we observed with confidence that SGD and RMSprop were sensitive to learning rate greater than .03 or smaller than .001 as the accuracy either dropped significantly or reached a low plateau. Adam in general exhibited a more robust performance especially with learning rate around .001. We observed Test Accuracy of 88.1641% for RMSprop and 88.8672% for Adam with learning rate of .001 (same layers and number of nodes as in baseline configuration). Results are shown in tables for three optimizers with learning rate of .001 (Table 1) and .005 (Table 2) respectively. Adam clearly won and we furthered our experiments with Adam and gradually decrease the learning rate, results as shown in Table 3: From Table 3, although Adam reached the highest test accuracy when learning rate is .0005, we may observe chances of overfitting because the highest validation accuracy was achieved at 12th epoch and a slight increase of train loss curve at the final few epochs. We observed the most smooth learning curve for learning rate of .0001 as shown in Figure 1b. With very close testing accuracy among the learning rates, we opted for the most smooth learning curve ($lr = .0001$) for less oscillations due to luck.

Algorithm 1 Setup FCN Parametrized Model

Setup `Class` using `nn.Module`.
Initialize model layers, pass in dataset input dimension (`input_dim`), number of neurons for each layer (`hidden_dims`), dropout rate (`dropout`), initialization methods(`init_wb`), and batch normalization switch (`batch_norm`) as parameters.
for i in range(length of `hidden_dims` -1) **do**
    append the new layer matrix by `append(nn.Linear())`
end for
Append the output layer to the model
Load dropout, initialization, and batch normalization parameter values into the model.
Define model Forward Propagation `forward`
for each layer in `self.layers` **do**
    apply each layer matrix onto input x
    if `batch_norm == True` **then** apply `nn.BatchNorm1d` to layer output x (`x.shape[1]`)
    end if
    apply ReLU activation
    if `dropout` $\neq$ `None` **then** apply `nn.Dropout` to current layer
    end if
end for
Define initialization function (`initialize`) for weight and bias initialization
if x `isinstance` of `nn.Linear` **then**
    Apply `Random Normal`, `Xavier`, or `Kaiming` initialization for weight and bias pursuant to parameter value
end if

Algorithm 2 Construct Training and Testing Call Function

Define `train_and_test` function that calls FCN Model in Algorithm 1
Call FCN model
Create Lost function and validation accuracy array length of the number of epoch.
Use `nn.CrossEntropyLoss` as our target loss function.
Choose Optimizer (`optim.SGD`, `optim.RMSprop`, or `optim.Adam`) depending on parameter value
for each epoch **do**
    for training sets with labels in each batch **do**
        Reshape `train_features` into columns of length 784
        Train training data by apply target loss function backward and optimize for each step
    end for
    Update `train_loss_list` for this epoch.
    Re-initialize validation accuracy to 0 for each epoch
    for validation sets with labels in each batch **do**
        Disable gradient descent calculation for validation purpose
        Reshape `val_features` into columns of length 784
        Train validation set
    end for
end for
Disable gradient descent calculation for testing
for testing sets in each batch **do**
    Reshape `test_features` into columns of length 784
    Test testing sets
    Compute Test Accuracy by accumulated number of correct classifications
end for

Optimizer	Test Acc.	Starting Val. Acc.	Highest Val. Acc.	Lowest Train Loss
RMSprop	88.1641%	81.8211%	89.9379%, 17^{th} Epoch	0.1159, 18^{th} Epoch
Adam	**88.8672%**	85.8929%	90.3759%, 12^{th} Epoch	0.0985, 18^{th} Epoch
SGD	66.5918%	26.2221%	68.1966%, 20^{th} Epoch	0.9949, 18^{th} Epoch

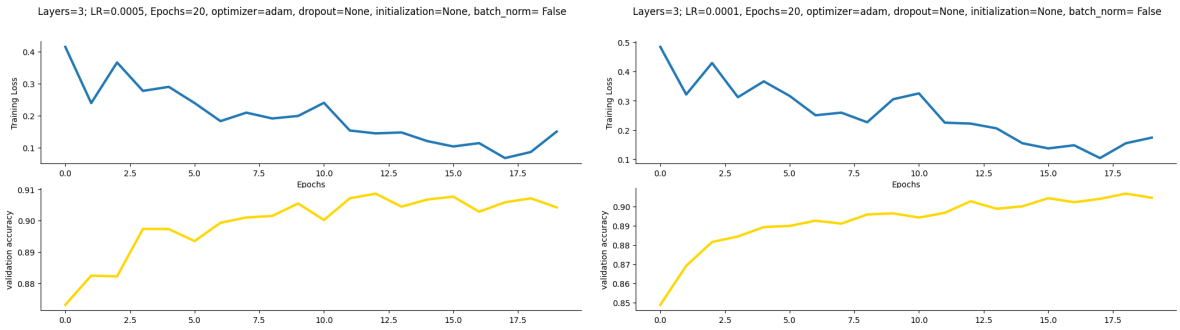
TABLE 1. Experiments for different optimizers with learning rate of .001.

Optimizer	Test Acc.	Starting Val. Acc.	Highest Val. Acc.	Lowest Train Loss
RMSprop	86.6113%	74.3129%	88.1468%, 20^{th} Epoch	0.2355, 18^{th} Epoch
Adam	**87.1777%**	83.8075%	89.1212%, 18^{th} Epoch	0.1363, 18^{th} Epoch
SGD	81.4941%	55.3952%	82.8238%, 19^{th} Epoch	0.4645, 15^{th} Epoch

TABLE 2. Experiments for different optimizers with learning rate of .005.

Learning Rate	Test Acc.	Highest Val. Acc.	Lowest Train Loss
0.001	88.1641%	90.3759%, 12^{th} Epoch	0.0985, 18^{th} Epoch
0.00075	88.3594%	90.7828%, 12^{th} Epoch	0.0881, 16^{th} Epoch
0.0005	89.2871%	90.8599%, 13^{th} Epoch	0.0671, 18^{th} Epoch
0.00025	89.1504%	90.8812%, 16^{th} Epoch	0.0702, 18^{th} Epoch
0.0001	88.9941%	90.6745%, 19^{th} Epoch	0.1046, 18^{th} Epoch

TABLE 3. Experiments for Adam with learning rate of .001, .00075, .0005, .00025 and .0001 respectively.



(A) Adam with learning rate 0.0005.

(B) Adam with learning rate 0.0001

FIGURE 1. 1a Validation accuracy and Training loss curve with learning rate of .0005. 1b with learning rate of .0001.

4.3. **Dropout Regularization.** We also experimented with dropout regularization, to our surprise, with a mere .1 probability of dropout would lead to a decrease in performance. We suspect that this may due to low number of data and training epochs which leads to potential underfitting situation of our model. This may give us direction towards training for higher (above 90%) test accuracy by increasing number of epochs, as we will discuss later in Subsection 4.5. Although a slight increase of performance may be observed with .01 dropout (as shown in Table 4), we speculate that such evanescent boost may merely due to the stochastic nature of probability. Therefore, we concluded that due to small number of epochs and training data, dropout may not present more desirable performance but instead lead to potential underfitting of the model. We decided to further approach other hyperparameters with *no dropout*.

4.4. **Initialization and Batch Normalization.** We continued our experiments on initializations such as Random Normal, Xavier Normal and Kaiming He Uniform and paired with batch normalization with no dropout, learning rate of .0001 using Adam optimizer as results shown in Table 5. We tested all combinations

Dropout	Test Acc.	Highest Val. Acc.	Lowest Train Loss
0.2	88.7305%	90.3271%, 20^{th} Epoch	0.1401, 19^{th} Epoch
0.1	88.5938%	90.4573%, 18^{th} Epoch	0.1384, 19^{th} Epoch
0.01	89.0625%	90.3936%, 19^{th} Epoch	0.1088, 19^{th} Epoch
0.005	88.8379%	90.5394%, 19^{th} Epoch	0.1135, 19^{th} Epoch
No dropout	88.9941%	90.6745%, 19^{th} Epoch	0.1046, 18^{th} Epoch

TABLE 4. Experiments for Adam (lr = .0001) with different dropout rates of .1, .2, .01, and .005 respectively.

of initialization and batch normalization, and found that all three initializations increase overall performance in test accuracy. For Random Normal, the training process starts with a huge drop in initial validation accuracy (drop from 84.8873% to 81.1792%) and a higher lowest train loss (increase from 0.1046 to 0.1561), and ends with an upward tail for train loss manifesting overfitting which was more significant when batch normalization was applied. For Xavier, the training process yields a less smooth validation accuracy curve but a bumpy train loss; batch normalization shockingly decrease the accuracy drastically when applied together with Xavier. For Kaiming Uniform, we observed a steady decrease of train loss function with a smooth increase in validation accuracy, batch normalization further improves the performance and a reasonable converging train loss function. However, if we want to achieve the best accuracy, we would strategically apply Random Initialization or Batch Normalization, either alone or with both.

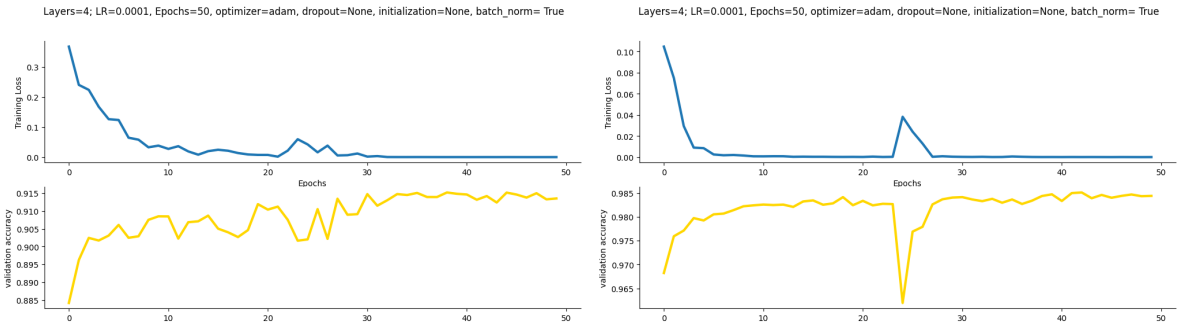
Initialization	Batch Norm.	Test Acc.	Highest Val. Acc.	Lowest Train Loss
No initialization	No	88.9941%	90.6745%, 19^{th} Epoch	0.1046, 18^{th} Epoch
Random Normal	No	89.3359%	90.3695%, 20^{th} Epoch	0.1561, 13^{th} Epoch
Xavier Normal	No	89.1504%	90.7779%, 17^{th} Epoch	0.0619, 20^{th} Epoch
Kaiming Uniform	No	89.1211%	90.9059%, 20^{th} Epoch	0.0446, 15^{th} Epoch
No initialization	Yes	89.3359% $\uparrow$	90.9385%, 15^{th} Epoch	0.0063, 19^{th} Epoch
Random Normal	Yes	89.3359% $\leftrightarrow$	91.1267%, 14^{th} Epoch	0.0279, 18^{th} Epoch
Xavier Normal	Yes	89.1406% $\downarrow$	91.0751%, 12^{th} Epoch	0.0062, 19^{th} Epoch
Kaiming Uniform	Yes	89.2773% $\uparrow$	90.7283%, 19^{th} Epoch	0.0116, 19^{th} Epoch

TABLE 5. Experiments for Initialization and batch normalization combinations.

4.5. **Fine Tuning and Testing for MNIST.** After numerous unsuccessful hyperparameter tuning to achieve $> 90\%$, we expand our model to include 4 hidden FCN layers for training with each layer having 1000, 2000, 1000, and 1000 neurons respectively. We then successfully achieve test accuracy of 90.5469% using Adam with learning rate .0001 and Batch Normalization in 50 epochs as shown in Figure 2a with specific configurations listed in Table 6. Slight fluctuation of training loss during the 20-30th epoch and we see a converging loss function reside at .0001. Validation accuracy started strong at 88.4192% and stayed within the range of 90.5% to 91.3%. We also tested the same configuration on the MNIST dataset and successfully achieved 98.3203% test accuracy; the validation accuracy started at 96.8240% and the train loss function tails near 0.0001 after 40 epochs; and we observed a deep wave for around 25th epoch. We observed such fluctuations in training loss and validation accuracy in both FashionMNIST and MNIST dataset. We contemplate that such trend may due to the difficult classification cases embed in the original data, and if we have enough epochs for training, overfitting can be overcome by finding *a local minimum which performs nearly as well as a global minimum* ([1]) rather than stopping at a poor local minimum around 25th epoch. In that sense, such local minima are acceptable because they correspond to significantly low values of cost function , as in both our cases here .0001.

5. Summary and Conclusions

We designed a FCN model with adjustable number of hidden layers, neurons in each layer, and other hyperparameters. We used Cross Entropy loss function to perform classification on FashionMNIST dataset.



(A) 90.5469% test accuracy on FashionMNIST (B) 98.3203% test accuracy on MNIST

FIGURE 2. 2a (left)Validation accuracy (yellow) and Training loss curve (blue) for FashionMNIST dataset. 2b (right) same configurations for original MNIST dataset.

Layers	**No. of Nodes**	**Epochs**	**Optim.**	**L.R.**	**Drop**	**Init.**	**B.Norm.**
4	1000, 2000, 1000, 1000	50	Adam	0.0001	None	None	Yes

TABLE 6. Specific configurations for 90.5469% accuracy on FashionMNIST and 98.3203% on MNIST: (1) 4 hidden FCN layers; (2) With Number of Nodes 1000, 2000, 1000, 1000 respectively; (3) 50 Epochs; (4) Adam optimizer; (5) Learning rate of 0.0001; (6) No dropout; (7) No Initialization; (6) With Batch Normalization.

We experimented three optimizers (SGD, RMSprop, and Adam), tried numerous learning rates for each optimizers and opted with Adam. We then found undesirable effects of dropout possibly due to underfitting, but observed benefits in performance with initialization and batch normalization. We finally successfully fine tuned our model for test accuracy of 90.5469% in FashionMNIST and 98.3203% in MNIST datasets. We visualized the results by logging train loss and validation accuracy, organizing those into tables, and plotting the curves for each epoch.

ACKNOWLEDGEMENTS

The author is thankful to Prof. Eli for useful discussions and code samples about Deep Neural Networks, Regularization, and Normalization.

REFERENCES

[1] I. J. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.