AMATH 482: HOME WORK 5

TIANRUI ZHOU

Amath Department, University of Washington, Seattle, WA
tianrz5@uw.edu

Abstract. This report is about Image Classification with Convolutional Neural Networks (CNN)[1], and its comparison with FCN. The content will be divided into five sections: Introduction and Overview, Theoretical Background, Algorithm Implementation and Development, Computational Results, and Summary and Conclusions. They will elaborate on how I used Neural Networks concept and Machine Learning concept to process the raw data FashionMNIST. I used Python (PyTorch) as my data processing tool and generated visual results with it. I will provide detailed explanations of my understanding of the mathematical foundations of the methods and algorithms. I will present my computational results and my conclusion for this project.

1. Introduction and Overview

I am working on the FashionMNIST dataset, which contains images depict articles of clothing, in order to train Fully Connected Deep Neural Networks (FCNs) and Convolutional Neural Networks (CNNs) to distinguish and classify the images in the dataset. The raw data has 60,000 training images and 10,000 testing images, and each of the clothing belongs to a class from one of 10 classes. I will train my classifier using the training set, and then, use validation set and testing set for evaluation and hyperparameter tuning for my FCN and CNN model. In terms of the testing accuracy, running time, and number of weights used, I will compare the performance of my two models using different number of weights.[1].

My goal of the project is designing my CNN and FCN models using different number of weights, such as 200K, 100K, 50K, 20K, and 10K. First, I will use 100K weights on FCN model and perform hyperparameter tuning to achieve over 88% on testing classification accuracy. Then, I will reduce the weights to 50K and also double the weights to 200K, in order to study the accuracy of these models by comparing the different FCN variants. I will test on 100K, 50K, 20K and 10K for CNN model, and compare the performance with all the FCN models[1].

2. Theoretical Background

My goal of the whole algorithm is using math foundation of FCN and CNN model concept and Machine Learning concept to experiment on models with different number of weights and compare their performances.

For loading the dataset from the FashionMNIST dataset, we separate the data in to two parts: training data and testing data. And in training data, we separate this data in to two sets: training set and validation set. The testing data is distinct from the training and validation data in order to evaluate the final model's performance.

Date: March 15, 2024.

In order to fairly compare FCN and CNN, I decided to do dropout with p equals to 0.5 and Kaiming initialization for both of the models. The **dropout regularization** effectively spreads the weights, and prevents overfitting in neural networks. Random neurons are dropped with probability p. **Initializations** initializing weights from various distributions plays a role in training. Some frequently used initialization procedures are Random Normal, Xavier and Kaiming.[13].

Considering efficiency and accuracy, I did hyperparameter tuning and set epochs to be 25. We also include the activation formula in both of the models: The **ReLU** activation function, defined as $f(x) = \max(0, x)$, and tanh activation function, defined as $f(x) = \frac{e^{2x}-1}{e^{2x}+1}$.

We first construct a class to set up my **FCN 100K** model, which means a constraint of using at most 100K number of weights. I set two hidden layers and with 112 and 106 neurons respectively, and end up with 99898 number of weights. The formula1 below shows the way of **computing the number of weights for a FCN model** using math formula, where L stands for the total number of layers in the neural network, and n_i means the number of neurons in layer i. I use the same formula 1 to calculate and design the FCN 50K with two layers (59 and 49 neurons) in total 49255 weights, and FCN 200K with two layers (205 and 189 neurons) in total 199859 weights.

$$W_{\text{total}} = \sum_{i=1}^{L} (n_i + 1) \times n_{i+1} \tag{1}$$

For constructing a **CNN** model, the **convolutional filter** works by moving around the input data, performing dot product between the input data (image) and the filter. The result of this whole process results in an output feature map. The formula 2 shows how the output feature map is created. $F[m, n]$ is the value of the output feature map at position (m, n), $I[m, n]$ represents the input data, $w[i, j]$ means the weights of the convolutional filter, f_h and f_w are the height and width of the filter respectively.

$$F[m, n] = \sum_{i=0}^{f_h-1} \sum_{j=0}^{f_w-1} I[m + i, n + j] \times w[i, j] \tag{2}$$

The **kernel size** stands for the dimensions of the filter; The **stride** parameter determines how much steps the filter shifts between the dot product computations between the input data; **Padding** in CNN involves adding additional pixels around the edges of input images or feature maps, with the purpose of mitigating the bias inherent in the convolution.

For the **maxpooling layers**, they reduce the size of the feature map and speed up the computation. A window, with dimension determined by the kernal size, is sliding over the feature map. Only the maximum value within each window is retained, in order to preserve the most significant information [12].

The **Fully Connected layer** is typically used at the end of the CNN to do the classification task. It takes the flattened output of from the previous layers and transform it to the final output, which is 10 different classes in our case.

In order to build CNN models with 100K, 50K, 20K, and 10K weights, I need to compute the number of weights. The math formula for **computing the weights** is the formula3 below:

$$\text{Number of weights in convolution} = (c_{\text{in}} \times k \times k + 1) \times c_{\text{out}} \tag{3}$$

c_{in} is the number of input channels, and c_{out} is the number of output channels. $k \times k$ determined the kernal size. The total number of weights is calculated by summing up the number of weights across all convolutional layers, and number of weights in fully connected layers.

Then, for the training and validation process, we iterate over the epochs, batches. we train and validate the FCN and CNN models respectively by using **gradient descent**.

$$W_{k+1} = W_k - \alpha \cdot \nabla J(W_k)$$

In this formula, W_{k+1} represents the updated weights, W_k is the current weights, α is the learning rate, and $\nabla J(W_k)$ represents the gradient of the cost function J. In the loop, we also employ the forward propagation, which is computing the loss through forward pass for a single training example, and also the back propagation, which is computing the gradients of parameters through backward pass for a single training example [11].

3. Algorithm Implementation and Development

In the python code, I used some software libraries and packages to help me do the calculation and analysis.

`Torch` is an open-source machine learning library that provides many tools for training neural networks, such as activation functions, optimizer, gradient computation, tensor operation and so on. `torch.nn.Module` is the base class for all neural network modules [6]. `optimizer.zero_grad()` resets the gradients of all optimized `torch.Tensor` s [8]. `torch.no_grad()` is the context-manager that disables gradient calculation [7]. `nn.Conv2d` module should have the input tensor with shape `[N, C, H, W]`, which is respectively `[number of samples in the batch, number of channels, the height of the input image, the width of the input image]`.

The `NumPy`[2] library is used for numerical operations and array manipulations. Some built-in functions in this package are really crucial to my program. `np.zeros()` returns a new array of given shape and type, filled with zeros [5]. `np.sum` sum of array elements over a given axis [4].
The `tqdm` is a library that provides a simple and smart way to add progress bars to and iterable objects, and it can also predict the remaining time [14].
The package `Matplotlib`[3] is used to make visualization of my data. `Seaborn` is a Python data visualization library based on `matplotlib`. It provides a high-level interface for drawing attractive and informative statistical graphics [10].
The `sklearn.model_selection` module is intended to assist with cross-validation. `train_test_split` split arrays or matrices into random train and test subsets [9].

4. Computational Results

In this project, I am experimenting on designing my FCN model and CNN model on the Fashion-MNIST dataset, using different number of weights, and I will compare their performances according to the testing accuracy and training time. To control my variables, all my FCN and CNN models will use **epochs = 25**, **dropout(p = 0.5)**, and **Kaiming initialization**.

First, I perform hyperparameter tuning on **FCN** model to achieve **over 88%** of testing classification, with the constraint of incorporating up to **100K** weights. With input 784 and output 10, I used two hidden layers with number of neurons of 112 and 106, so the whole number of weights is **99898**, which is around 100K. After training and testing, I got the testing classification accuracy of **88.35%** with standard deviation of 1.80%, which is above 88% accuracy.

Then, I reduce by half the number of weights to 50K, and also increase the number of weights to 200K. I train these models the same as the way for FCN 100K, and record my accuracy and training time results in Table 1.

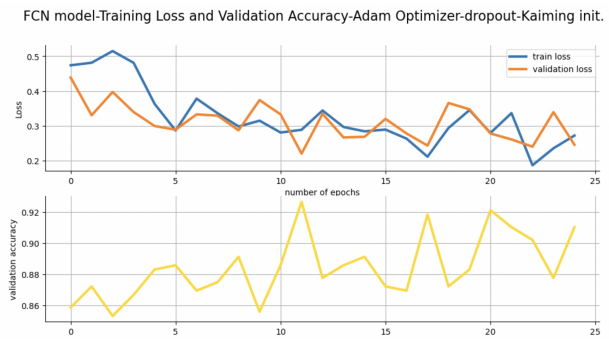
TABLE 1. FCN model comparison for different number of weights

Model	Validation Accuracy (%)	Training Time	Testing Accuracy (%)
FCN 50K	89.95	1:00	87.96 ± 2.09
FCN 100K	91.03	1:02	88.35 ± 1.80
FCN 200K	92.39	1:05	88.95 ± 1.75

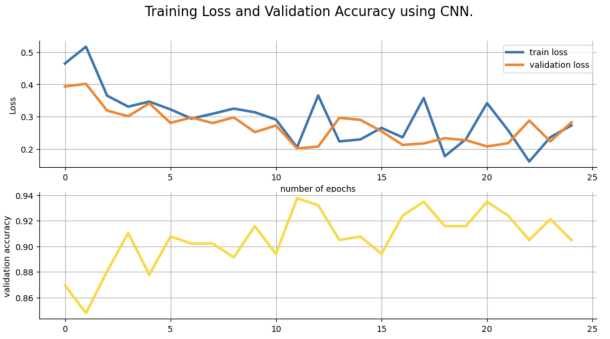
By analyzing Table 1, we can study the accuracy and training time of these models by comparing the different FCN variants. The **FCN 200K** model achieves the highest testing accuracy of 88.95%, followed by **FCN 100K** with 88.35% and **FCN 50K** with 87.96%. The FCN 200K also has the smallest standard deviation of 1.75%, and the FCN 50K has the largest standard deviation of 2.09%. This trend suggests that increasing the number of weights may allow the model to extract more information from the data, leading to improved testing accuracy. For FCN, it may be worth to increase the weights to 200K compared to 100K, because the training time only increase 3 seconds, but the testing and validation accuracy both increase and the standard deviation decreases.

For constructing Convolutional Neural Networks (CNN) model, I will hold convolutional layer parameter of kernel_size = 5, stride = 1, padding = 2; maxpooling kernel_size = 2. As FCN, I also add dropout and Kaiming initialization to my CNN model.

Now I implement and train a **CNN 100K** model with two sets of convolutional, max pooling, and FC layers with up to 100K weights, which is specifically 96474 weights. According to my record in Table 2, my CNN 100K model ends up having testing accuracy of **91.36%** with standard deviation of 1.75%. **The testing accuracy of CNN 100K is 3.01% higher than that of FCN 100K**, the validation accuracy is also higher, and CNN 100 has lower standard deviation. Even using FCN 200K, which means using twice number of weights, its testing accuracy of 88.95% is still lower than that of CNN 100K. Figure 1 shows the plot for training loss curve and validation accuracy for FCN 100K and CNN 100K.



(A) Training loss curve, validation loss curve, and validation accuracy curve for FCN 100K



(B) Training loss curve, validation loss curve, and validation accuracy curve for CNN 100K

FIGURE 1. Both using 100K weights, the performance curve comparison for FCN and CNN model

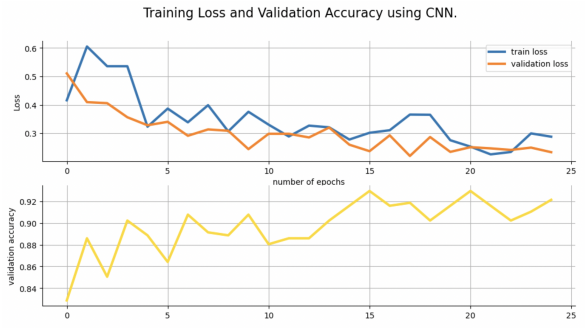
Then, I reduce the number of weights in the CNN 100K model to create **CNN 50K** (49644 weights), **CNN 20K** (19306 weights), and **CNN 10K** (9516 weights). I train them the same way

TABLE 2. CNN model comparison for different number of weights

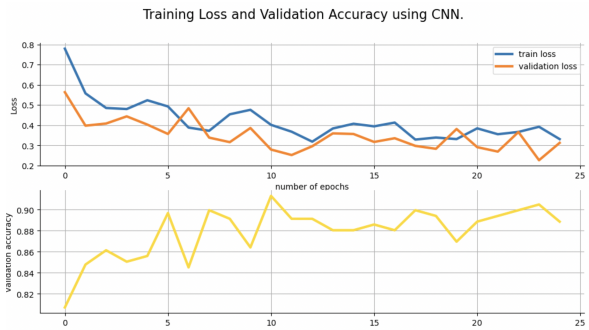
Model	Validation Accuracy (%)	Training Time	Testing Accuracy (%)
CNN 10K	88.59	5:18	87.73 ± 2.39
CNN 20K	88.86	7:12	89.38 ± 1.76
CNN 50K	92.12	12:19	91.04 ± 1.49
CNN 100K	90.49	19:16	91.36 ± 1.75

as the CNN 100K model does.

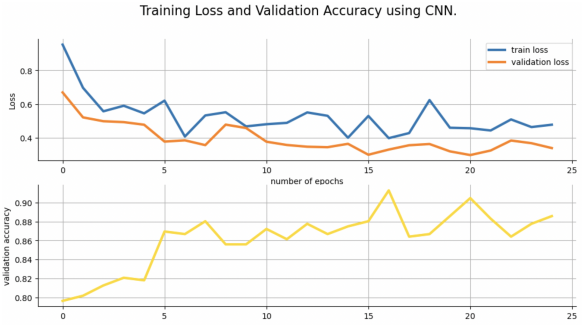
We can get useful information using the result we get in Table 2. Among the CNN models for different numbers of weights, the **CNN 100K** model exhibits the highest validation accuracy of 90.49% and testing accuracy of 91.36%, but requires the longest training time of 19 minutes and 16 seconds; the **CNN 50K** model has a shorter training time of 12 minutes and 19 seconds. Despite having 50K fewer weights compared to the CNN 100K model, it maintains a relatively similar testing accuracy of 91.04%; The **CNN 20K** uses only one-fifth the number of weights compared to the CNN 100K variant, but it still achieves decent testing accuracy of 89.38%, with training time of 7 minutes and 12 seconds; **CNN 10K**, has the lowest number of weights, achieves testing accuracy of 87.73%, using the shortest time to train.



(A) Training loss curve and validation accuracy curve for CNN 50K



(B) Training loss curve and validation accuracy curve for CNN 20K



(C) Training loss curve and validation accuracy curve for CNN 10K

FIGURE 2. Training loss curve and validation accuracy curve using three different number of weights of CNN model

Comparing these CNN variants with FCN variants, we can observe that **CNN** models tend to need longer time for training, but they are able to achieve higher testing classification accuracies even when using less number of weights than FCN model. For example, **CNN** can use only 20K weights to reach 89.38% of testing accuracy, which can not be achieved by **FCN** using even 200K weights (88.95%).

Overall, **FCN** tends to be more time-efficient, but it needs more weights in order to achieve higher testing accuracy; However, **CNN** tends to be time-consuming, but it is capable of using small amount of weights to achieve higher testing accuracy, compared to FCN.

5. Summary and Conclusions

I am working on the FashionMNIST dataset, which contains images depict articles of clothing. In order to classify all the data in to 10 classes, I train Fully Connected Deep Neural Networks (FCNs) and Convolutional Neural Networks (CNNs) to distinguish and classify the images in the dataset. The raw data consist of images of clothing belongs to a class from one of 10 classes. I train my classifier using the training set, and then, use validation set and testing set for evaluation and hyperparameter tuning for my FCN and CNN model.

For FCN model, I tested with 100K, 50K, and 200K weights; For the CNN model, I tested with 100K, 50K, 20K, and 10K weights. For each of them, I log their validation and testing accuracy, and also the running time. I compare them in terms of the efficiency and the accuracy performances. The conclusion of my program is that, **FCN** tends to be more time-efficient, but it needs more weights in order to achieve higher testing accuracy; However, **CNN** tends to be time-consuming, but it is capable of using small amount of weights to achieve higher testing accuracy, compared to FCN.

Acknowledgements

The author is thankful to Prof. Eli Shlizerman for useful information in Lectures and Recitation about some important details about this project, and the useful information about how to use CNN PyTorch. We are also thankful to peer Ziyi Zhao and all the classmates on Discord for meaningful discussion of logic of the coding part, and helping me debug my code successfully.

References

[1] Directions, reminders and policies. PDF, 2024. File: AMATH482582$_{H}omework5.pdf$.
[2] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.
[3] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
[4] NumPy. numpy.sum documentation. `https://numpy.org/doc/stable/reference/generated/numpy.sum.html`.
[5] NumPy. numpy.zeros documentation. `https://numpy.org/doc/stable/reference/generated/numpy.zeros.html`.
[6] PyTorch. torch.nn.module documentation. `https://pytorch.org/docs/stable/generated/torch.nn.Module.html`.
[7] PyTorch. torch.no_grad documentation. `https://pytorch.org/docs/stable/generated/torch.no_grad.html`.
[8] PyTorch. torch.optim.optimizer.zero_grad documentation. `https://pytorch.org/docs/stable/generated/torch.optim.Optimizer.zero_grad.html`.
[9] scikit-learn Developers. scikit-learn.train_test_split documentation. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html`.
[10] Seaborn Developers. Seaborn documentation. `https://seaborn.pydata.org/`.
[11] E. Shlizerman. Amath 482 lecture 19 notes, 2024.
[12] E. Shlizerman. Amath 482 lecture 24 notes, 2024.
[13] E. Shlizerman. Amath 482 practice3 pytorch optimization hyperparam, 2024.
[14] tqdm Developers. tqdm documentation. `https://tqdm.github.io/`.