Model Design for Deep Learning in Practice

Yulin Wu (yw3376)
Department of Statistics
Columbia University in the City of New York
New York, NY 10027
yw3376@columbia.edu

Jiaqi Yuan (jy3021)
Department of Statistics
Columbia University in the City of New York
New York, NY 10027
jy3021@columbia.edu

Siqi Zhao (sz2866)
Department of Statistics
Columbia University in the City of New York
New York, NY 10027
sz2866@columbia.edu

Chuyang Zhou (cz2578)
Department of Statistics
Columbia University in the City of New York
New York, NY 10027
cz2578@columbia.edu

Abstract

In this project, we explore deep learning models to find well-performed models with high out-of-sample accuracy on MNIST dataset. We focus on three architectures, including baseline model, one layer CNN and two layer CNN. We compare these models by adding different regularization methods, and using different optimizations and initializations. We test the model robustness and generalization using Fashion-MNIST dataset. In conclusion, our best model is a two layer CNN model, with test accuracy equals 0.9953 in MNIST and 0.9179 in Fashion-MNIST, which is a desirable result.

1 Introduction

1.1 Problem overview

In this project, we design deep learning models (mainly Convolutional Neural Networks) using GPU-enabled Tensorflow and Keras API to achieve a high accuracy on the MNIST dataset. After we developed the model, we test our model using the Fashion-MNIST dataset to see model robustness and generalization. The choices of design included: network architecture, optimization, initialization, and regularization. We want to find the best performing model from the baseline model, one layer CNN and two layer CNN. With a particular interest in regularization, we would like to dig deeper in this direction.

1.2 Literature review

Our model design is informed by previous work. Based on Wang et al (2020)'s work on examining how different activation functions (ReLU and Elus is better than Sigmoid), learning rates(0.0001 and 0.0002) and the addition of the Dropout layer(randomly "delete" half of the hidden layer units) in front of the output layer will make the convergence speed different, weaken the influence of the initial parameters on the model, and improve the training accuracy of LeNet-5 model on MNIST dataset. Another work by Byerly and Kalganova (2020) gives us some intuition on weights initialization, batch normalization, data augmentation. Based on CNN, it also compared model performance on MNIST and Fashion-MNIST, which is a good reference. Moreover, we draw on the work of Greeshma and Sreekumar (2019), which outlines a series of regularization methods on Fashion-MNIST. We would like to use it as a guideline for conducting hyperparameter optimization and regularization on MNIST.

1.3 Data description

In this project, we use two datasets: MNIST and Fashion-MNIST (FMNIST). For each of the dataset, we preserve 20% as a validation set. The MNIST database consists of handwritten single digit numbers(0-9). Each image is 28*28 pixels, and the database consists of 60,000 training images and 10,000 testing images. The Fashion-MNIST is a dataset of clothing images. Each image is 28*28 pixels, and the database consists of 60,000 training images and 10,000 testing images. Before we run our models, we normalize and reshape data.

1.4 Methods overview

	Baseline	One Layer CNN	Two Layer CNN
Network Architecture	Fully Connected: Dense(16)-> Dense(16)-> Dense(10)	Basic One Layer CNN: Conv2D->Dense(128)->Dense(10)	(1) Basic Two Layer CNN Conv2D->Conv2D->BN+Maxpool+Dropout-> Dense(128)->Dense(10) (2) Double Hidden Units in Dense Layer as 256 Conv2D->BN+Maxpool+Dropout->Conv2D->BN+ Maxpool+Dropout->Dense(256)->Dense(10) (3) Double the number of activation maps in both Conv2D from filter=16 to filter=32
Optimization	N/A	Optimizer a. Adam - learning rate=0.001 - learning rate=0.0001 b. SGD c. RMSprop	N/A
Initialization	N/A	RandomNormal a. mean = 0 and std = 0.01 b. mean = 0 and std = 0.1 RandomUniform a. min = 0 and max = 1 b. min = -0.5 and max = 0.5	N/A
Regularization	N/A	[A]. max pooling layer (2*2) [B]. [A]+dropout layer (0.5) [C]. [A]+dropout layer (0.5) *2 layers [D,E]. [C]+weight decay 1e-3 vs 1e-5 [F]. [E]+batch normalization	[A] Add regularization to Two-layer CNN architecture (1) [B] Use data augmentation in Two-layer CNN architecture (3)

Figure 1.4.1: summary of methods

2 Model development using MNIST

In this section, we provide details on how we develope the final one layer CNN model and two layer CNN model through regularization, tuning hyperparameters, comparing different initialization. Our starting point is the baseline model.

2.1 Baseline model

The baseline model is an unregularized MLP with 2 hidden layers of 16 units and ReLU activation. We trained the model for 100 epochs with Adam optimize and minibatch size of 128. We used the default keras optimization and initialization hyperparameters. The test accuracy is 0.9514, test loss is 0.2361, validation accuracy is 0.9464, validation loss is 0.2551. Figure 2.1.1 show the training and validation accuracy and loss on MNIST with a clear sign of overfitting.

We will keep the baseline results and try to find better performing models in the following section.

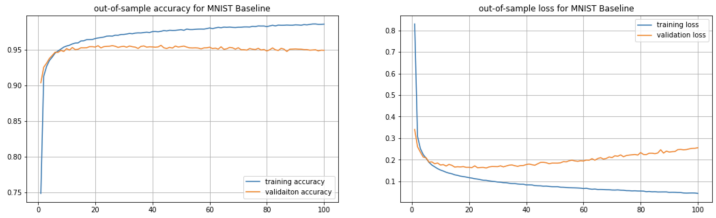


Figure 2.1.1: baseline model out-of-sample results

2.2 One layer CNN

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
dense (Dense)                (None, 128)               3211392
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 3,213,514
Trainable params: 3,213,514
Non-trainable params: 0
_________________________________________________________________
```

Figure 2.2.1: basic one layer CNN

Step 0: simple one layer CNN As shown in Figure 2.2.1, we start off by building a simple model with the detailed architecture in order: convolutional layer, fully connected layer, and chose "ReLu" as activation function to prevent exploding and vanishing gradients. This is shown in Figure 2.2.1.4 below. This model has test accuracy as 0.9888 and the validation loss as 0.1134. However, from step 0 in Figure 2.2.1.4, we saw that the model is overfitting, as there is a rising trend in the validation loss as epochs increase. We diagnose overfitting through looking at the trend of the graphs in the table. If the training accuracy is improving while the testing accuracy is not, or if the validation loss is increasing as the epochs increases, we conclude that the model is overfitting.

In the following section, we will talk about several regularization methods we used to resolve overfitting. We compared a stacked them to get our best one layer CNN model.

2.2.1 Regularization

Step 1: add max pooling In our CNN, we add a max pooling layer with pool size equal to 2*2 (step 1 in Figure 2.2.1.4). The pooling layer makes the representations smaller and more manageable. It operates over each activation map independently. Here, we take the maximum number from each 2*2 window as out output. After this layer, the height and width of pixels become half of before while the number of channels stays the same. Then, our testing accuracy has increased to 0.9901 and the validation loss has decreased to 0.078. From the graph, we see that the overfitting is ameliorated, but it still exists.

Step 2 and 3: add dropout Due to the overfitting problem after Step 1, we decide to try adding a dropout layer with dropout rate as 0.5 to the convolutional layer (step 2 in Figure 2.2.1.4). Dropout is a popular regularization technique to reduce overfitting in neural networks. It randomly drops out some units in the network. The dropout rate is chosen by comparing different commonly used dropout rates. This returns a 0.9899 testing accuracy and decreased validation loss (0.0577). The overfitting issue is significantly getting better, even though it still exists.

To further reduce overfitting , we add another dropout layer in the fully connected layer with the same dropout rate 0.5 (step 3 in Figure 2.2.1.4). After adding this layer, the test accuracy is 0.9918 and the validation loss is 0.0368. What is noticeable is that the overfitting is greatly reduced after adding each of the two dropout layers, though it still presented.

Step 4 and 5: add weight decay Next, to improve based on Step 3 and reduce overfitting, we consider weight decay, which is used to penalize complexity. During the training process, we desire a lower cost, which requires providing penalty for more and larger parameters. As shown in the previous paper, weight decay can improve neural network through suppressing irrelevant components of the weight vector and suppressing some effects of static noise on targets.

In the Conv2D layer, we compare two weight decay hyperparameters 10^{-3} and 10^{-5}. When we used kernel regularizer l2(1e-3), the test accuracy is 0.9917 and the validation loss is 0.039. When we used kernel regularizer l2(1e-5), the test accuracy is 0.9912 and the validation loss is 0.0387.

We notice that their test accuracy and loss were fairly close. But their accuracy and loss did not give the same conclusion on which model is better. We thought this loss-vs-accuracy discrepancy occurred because when a model is over-confident in its predictions, a single false prediction will increase the loss unproportionally compared to the minor drop in accuracy. Since overfitting(over-confident) is an issue in all previous models, shown in Figure 2.2.1.4., we would like to choose loss as the reference to make choices. Thus we go for weight decay 10^{-5} as it has lower loss.

Also, we notice that there are fluctuation in the plots. This is acceptable since different batches contain different samples of images.

Step 6: add batch normalization Finally, we try to add batch normalization (step 6 in Figure 2.2.1.4). Batch normalization works through normalizing the input layer through re-centering and re-scaling, which will make the neural networks more stable and faster. Our batch size is 128, which is not small. Batch normalization probably could help here. We wanted to see what result we would get if we include Batch Normalization.

Following the same logic as in Step 4 and 5, we focus on loss values. We find that adding Batch Normalization improved the loss value from 0.0387 to 0.0385 in step 5. Also, from the graph, we see that the there is no noticeable sign of overfitting. We achieve a high test accuracy of 0.9907.

We conclude with this model sequential_8, and the architecture is shown below (Figure 2.2.1.1). The out-of-sample accuracy is 0.9907, and the validation loss is 0.0385. Besides table 2.2.1.4, which includes detailed information for each step, we also included two graphs of all the regularization methods we used (Figure 2.2.1.2 and 2.2.1.3).

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_8 (Conv2D)            (None, 28, 28, 32)        832
_________________________________________________________________
batch_normalization (BatchNo (None, 28, 28, 32)        128
_________________________________________________________________
activation (Activation)      (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
dropout_11 (Dropout)         (None, 14, 14, 32)        0
_________________________________________________________________
flatten_8 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_16 (Dense)             (None, 128)               802944
_________________________________________________________________
dropout_12 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_17 (Dense)             (None, 10)                1290
=================================================================
Total params: 805,194
Trainable params: 805,130
Non-trainable params: 64
_________________________________________________________________
```

Figure 2.2.1.1: one layer model - final model summary after regularization

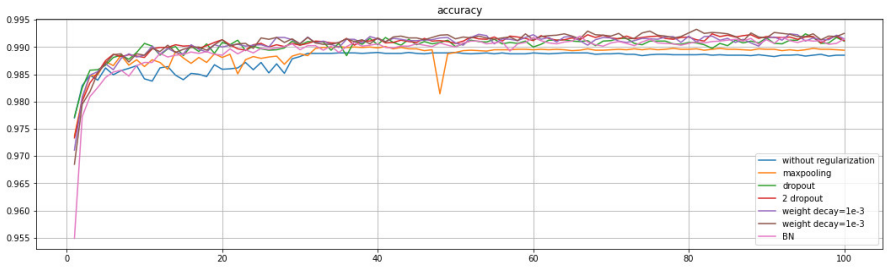


Figure 2.2.1.2: one layer model - validation accuracy

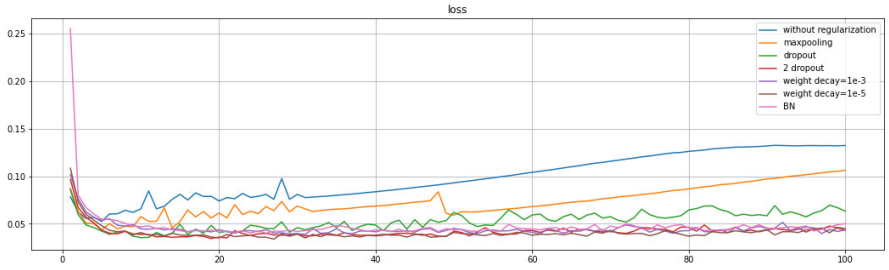
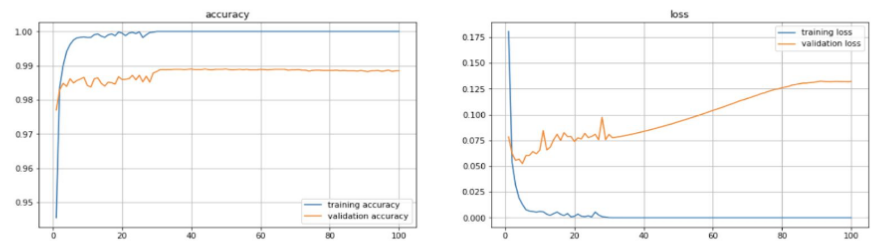
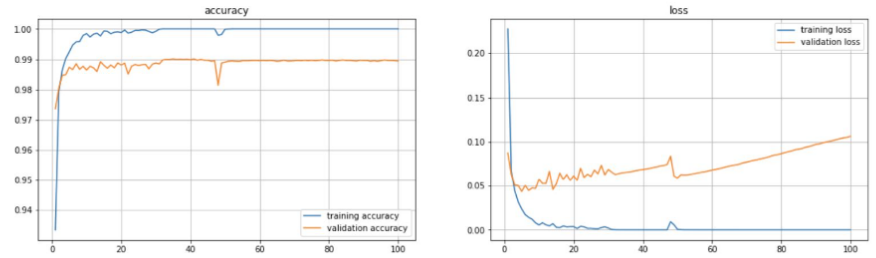
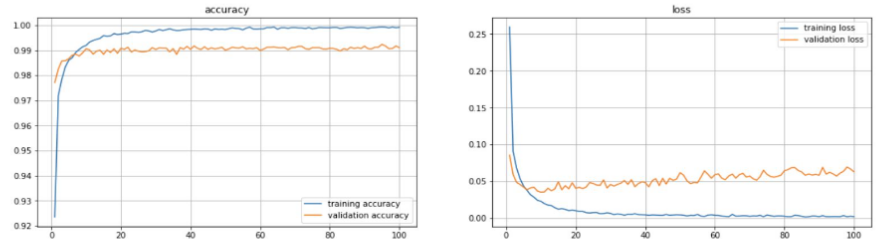
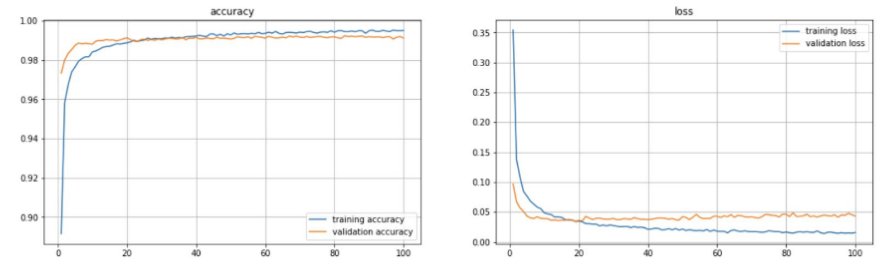


Figure 2.2.1.3: one layer model - validation loss

Steps	Graph
0	One Layer: test accuracy: 0.9880, Loss: 0.1134 
1	Add Maxpool Layer(2*2) : test accuracy: 0.990, Loss: 0.078 
2	Add Dropout Layer: test accuracy: 0.9899, Loss: 0.0577 
3	Add Another Dropout Layer: test accuracy: 0.9918, Loss: 0.0368 

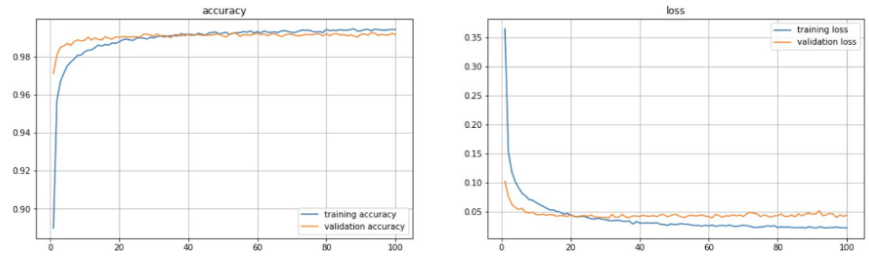
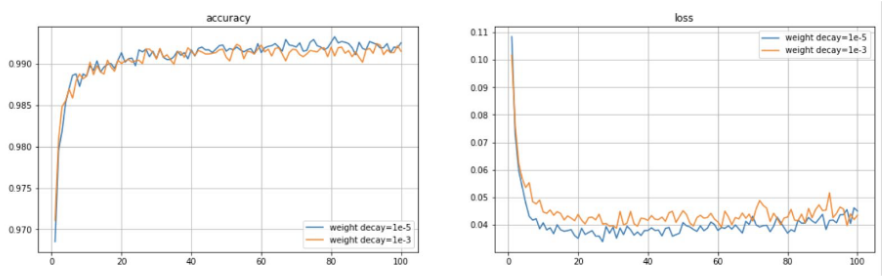
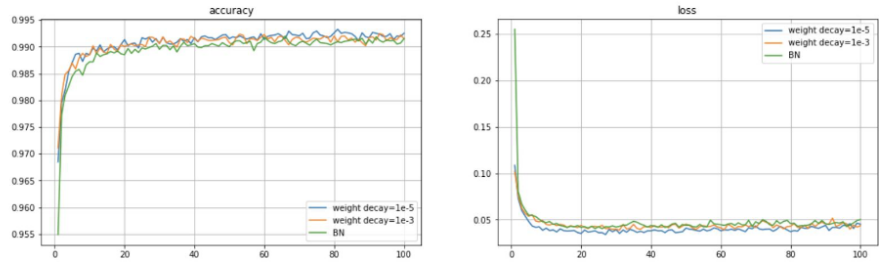
4	Add Weight Decay (with l2 kernel regularizer with weights default to be 1e^-3) : test accuracy: 0.9917, Loss: 0.0390 
5	Tune Weight Decay (l2(1e^-5)): test accuracy: 0.9912, Loss: 0.0387 
6	Add Batch Normalization: test accuracy: 0.9907, Loss: 0.0385 

Figure 2.2.1.4: one layer model - table of regularization

2.2.2 Tuning

After we finish regularization, we obtain model sequential_8 as shown in Figure 2.2.1.1 above. The parameters in the model has convolutional layer output depth as 32, kernel size as 5*5, and optimizer as Adam. Here, we tried to tune these parameters and learning rates.

We start off by trying additional two output depth, 16 and 64, and we compare their performance with output depth equals to 32. Output depth is the number of filters used. For output depth equals to 16, the testing accuracy is 0.9920 and validation loss is 0.0274. For output depth equals to 64, the testing accuracy is 0.9916 and validation loss is 0.9916. Compared with output depth 32 in model_sequence 8 (testing accuracy as 0.9907 and loss as 0.0385), output depth equals 16 has higher testing accuracy as well as a smaller loss. Thus, we decided to use output depth as 16.

After decide on the output depth(16), we tune the kernel size. We try an additional kernel size, 3*3. For kernel size equals to 3*3, the testing accuracy is 0.9905 and has a lower validation loss is 0.0430. Thus, we keep the original choice with kernel size as 5*5.

After fixed output depth as 16 and kernel size as 5*5, we experiment with different optimizers: SGD and RMSprop. For SGD, the testing accuracy is 0.9893 and validation loss is 0.0326. For RMSprop, the testing accuracy is 0.9845 and loss is 0.0554. Comparatively, the optimizer Adam in model_sequence 8 has higher testing accuracy as well as a smaller validation loss. Thus, we decide to keep Adam.

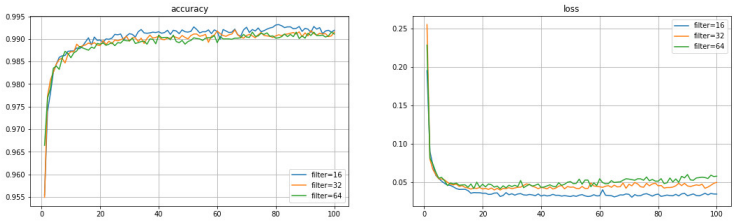


Figure 2.2.2.1: one layer model - different filters for MNIST

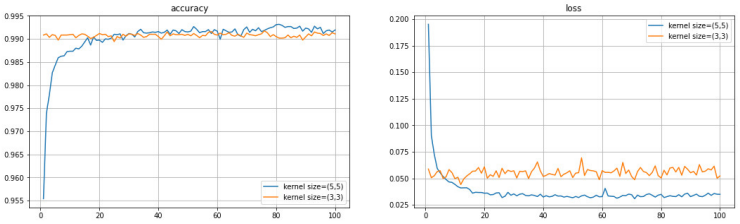


Figure 2.2.2.2: one layer different kernel sizes for MNIST

After fixed output depth as 16 , kernel size as 5*5 and optimizer as Adam, we try different learning rates: 0.001, 0.0001. For learning rate as 0.001, the testing accuracy is 0.9916 and the validation loss is 0.0355. For learning rate as 0.0001, the testing accuracy is 0.9906 and the validation loss is 0.0300. Comparing these two learning rates, we see that the learning rate as 0.001 has higher testing accuracy as well as a smaller validation loss. Thus, we adopt learning rate 0.001.

2.2.3 Initialization

Afterwards, we try different initializations, including RandomNormal class and RandomUniform class.

We use random normal with mean equals 0 and std equals 0.01. The testing accuracy is 0.9912 and the validation loss is 0.0301. We then try random normal with mean equals 0 and std equals 0.1. The testing accuracy is 0.9920 and validation loss is 0.0292. We also try a random uniform with min equals 0 and max equals 1. The testing accuracy is 0.9908 and the validation loss is 0.0339. For random uniform with min equals -0.5 and max equals 0.5, the testing accuracy is 0.9910 and loss is 0.0330.

From all the results above, we find that the RandomNormal class performed better than the RandomUniform class in MNIST dataset, based on our limited two choices above. And RandomNormal with larger standard deviation of 0.1 works best here. As there is no single best way nor guidance to select the initial weights of a neural network(as referenced in Deep Learning, 2016 page 301), our above result might be dataset specific [2].

For model simplicity, we keep the default initialization(glorot uniform) in the rest of the project.

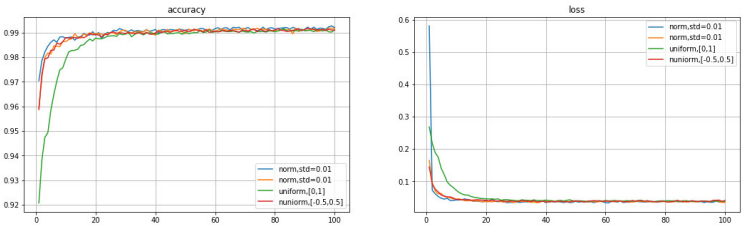


Figure 2.2.3.1: one layer model - different initializations for MNIST

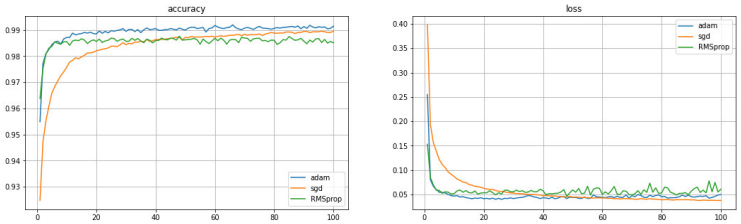


Figure 2.2.2.3: one layer model - different optimizers for MNIST

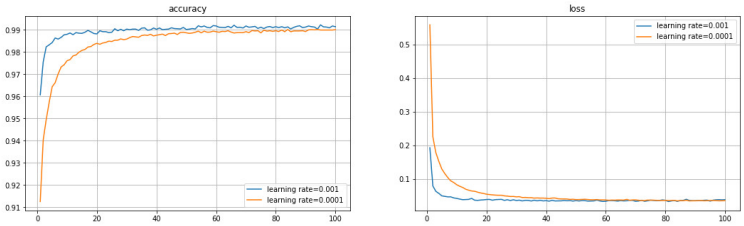


Figure 2.2.2.4: one layer model - different learning rates for MNIST

2.3 Two layer CNN

After we try single convolutional layer architectures, we want to see if adding one more convolutional layers would improve model performance further.

2.3.1 Architecture: adjacent convolutional layers

First, we add one more convolutional layer to the best performing one layer CNN model $adam\_lr1$ above to see if we can further improve our model. To start off, we put two convolutional layers adjacent while keeping all other layers the same as $adam\_lr1$. We notice that the test accuracy of the two models are around the same from 0.9917 to 0.9918, and there is a decrease in the validation loss from 0.355 to 0.311. This improvement happened because we introduced more parameters by adding one convolutional layer. However, we notice that, as epoch increases, there is a very slight sign of overfitting from the validation set result in Figure 2.3.1.1. We consider reducing it in the next step.

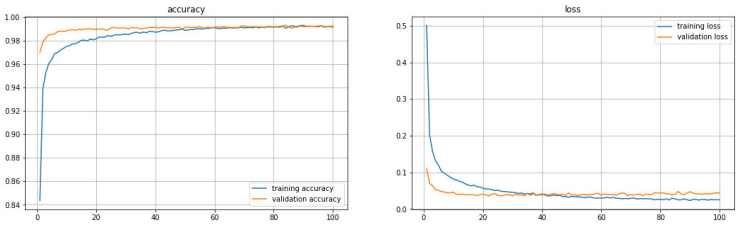


Figure 2.3.1.1: validation accuracy and loss of model twocnnA

2.3.2 Add regularization after each convolutional Layer

To add regularization to the above $twocnnA$ model, we modify the architecture that each convolutional layer follows by a set of regularization methods. Not surprisingly, this model has a clear improvement as shown in Figure 2.3.2.2 over the $twocnnA$ model: though it had a lower training accuracy from 0.9929 to 0.9822, it had the same validation accuracy of 0.9912 and higher test accuracy from 0.9918 to 0.99.37. Also, we notice from Figure 2.3.2.1 that there is no overfitting. This is because the newly added set of regularization drops many parameters to avoid overfitting. Meanwhile, the model performance is not affected due to the already complex structure with such a large number of parameters.

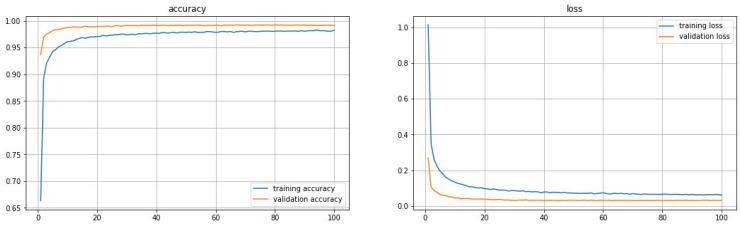


Figure 2.3.2.1: validation accuracy and loss of model twocnnB

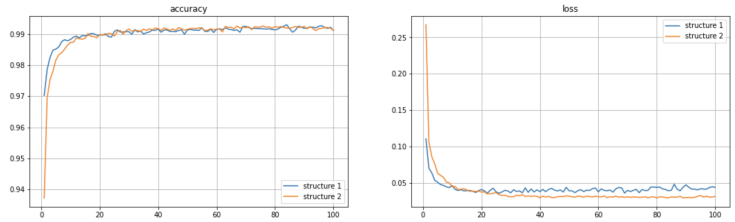


Figure 2.3.2.2: twocnnA vs twocnnB: adding regularization in two layer CNN

2.3.3 Architecture: double hidden units in dense layer

As adding regularization works well in the above model without noticeable overfitting, we tried to double the hidden units in the last fully connected layer to see if we can get better performance by adding parameters.

Even though we saw from the result of $twocnnC$ in Figure 2.3.3.1 that there is no sign of overfitting, the performance of this model is not better than the previous model $twocnnB$ as show in Figure 2.3.2.2 based on the validation and test results. It has a slight drop in test accuracy from 0.9937 to 0.9936.

This model is not a good choice.

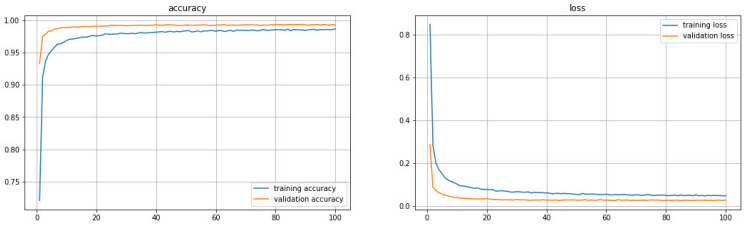


Figure 2.3.3.1: validation accuracy and loss of model twocnnC

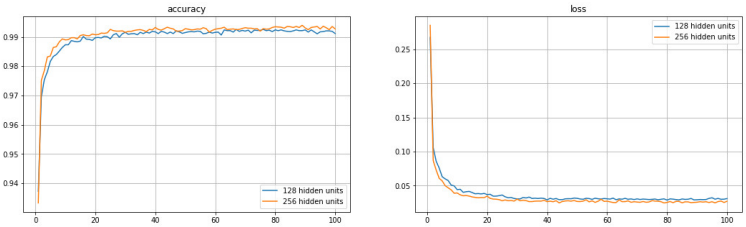


Figure 2.3.3.2: compare twocnnB and twocnnC

2.3.4 Architecture: double the number of activation maps in convolutional layers

Though $twocnnC$ did not perform well, it did not overfit. So we wanted to increase the number of parameters from $twocnnC$ to see if we can boost the model performance.

Increasing the number of activation maps (i.e. the output depth) increases the units in hidden layers and also increases the number of parameters.

We tried to double the number of activation maps from 16 to 32 based on the $twocnnC$ model. Results are shown in Figure 2.3.4.1.

In this case, compared to $twocnnC$ in Figure 2.3.4.2, we notice that the validation loss of different output depths are quite close. But when the number of activation maps equals 32, we achieved higher test, validation and training accuracies, with test accuracy equals 0.9953. Thus, $twocnnD$ is a better model than $twocnnC$ and a better model for MNIST.

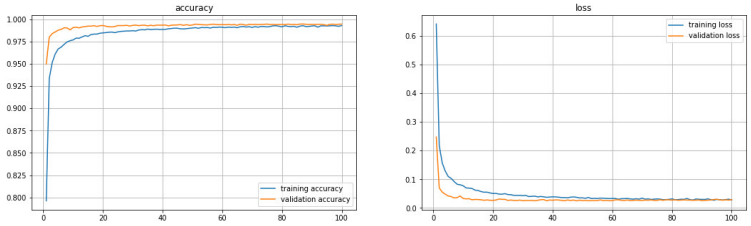


Figure 2.3.4.1: validation accuracy and loss of model twocnnD

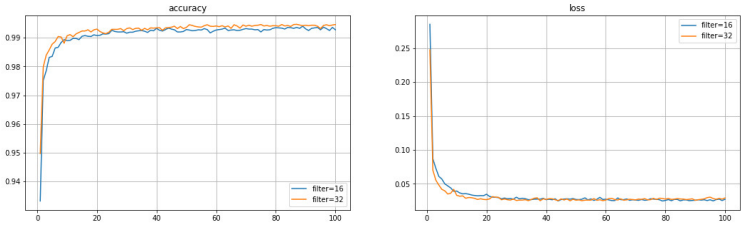


Figure 2.3.4.2: compare twocnnC and twocnnD

To sum up our two layer CNN architecture exploration, the best model is twocnnD which has filter 32 and regularization follows each convolutional layer. The model architecture is shown in Figure 2.3.4.3.

```
Model: "sequential_40"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_43 (Conv2D)           (None, 28, 28, 32)        832
_________________________________________________________________
batch_normalization_25 (Batc (None, 28, 28, 32)        128
_________________________________________________________________
activation_25 (Activation)   (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d_32 (MaxPooling (None, 14, 14, 32)        0
_________________________________________________________________
dropout_59 (Dropout)         (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_44 (Conv2D)           (None, 14, 14, 32)        25632
_________________________________________________________________
batch_normalization_26 (Batc (None, 14, 14, 32)        128
_________________________________________________________________
activation_26 (Activation)   (None, 14, 14, 32)        0
_________________________________________________________________
max_pooling2d_33 (MaxPooling (None, 7, 7, 32)          0
_________________________________________________________________
dropout_60 (Dropout)         (None, 7, 7, 32)          0
_________________________________________________________________
flatten_40 (Flatten)         (None, 1568)              0
_________________________________________________________________
dense_80 (Dense)             (None, 256)               401664
_________________________________________________________________
dropout_61 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_81 (Dense)             (None, 10)                2570
=================================================================
Total params: 430,954
Trainable params: 430,826
Non-trainable params: 128
_________________________________________________________________
```

Figure 2.3.4.3: best model architecture

2.4 Regularization: data augmentation

Then we consider data augmentation as a regularization method to see if we can continue improving the model performance based on the $twocnnD$ model. We added new augmented images to the training set to increase data diversity, including images that were flipped horizontally, vertically and adjusted contrast. The results for training and validation sets are shown in Figure 2.4.1. There is no clear sign of overfitting which is the same as $twocnnD$ which did not use the data augmentation.

We then focused on the comparison between with and without data augmentation. After data augmentation, training accuracy boosted a little. The validation accuracy and loss, as shown in Figure 2.4.2, are very close to each other after 60 epochs. However, the test accuracy after data augmentation dropped 0.2%. The explanation we offer is that sometimes the newly added augmented images do not make sense. For example, flipping '4' upside down did not look similar to any digits from 0 to 9, but only added noises to the training dataset, which led to a lower test accuracy here.

So, data augmentation is not very helpful here.

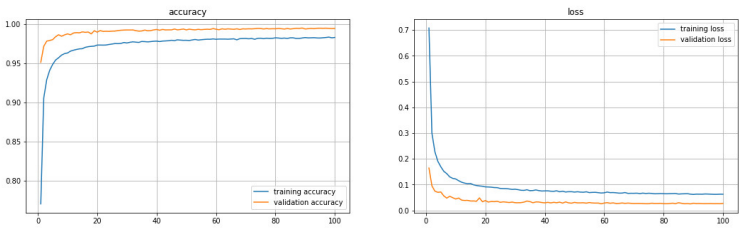


Figure 2.4.1: validation accuracy and loss with data augmentation

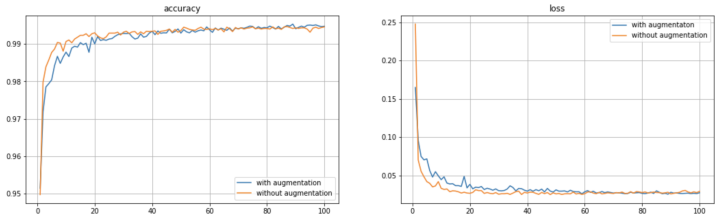


Figure 2.4.2: compare with and without data augmentation

3 Run models on Fashion-MNIST

After we build the models on MNIST, we follow the same steps and apply the same models to Fahion-MNIST (FMNIST) to compare the results with those on MNIST. We are interested in the performance of our fine-tuned models on FMNIST built in section 2.

3.1 Baseline model

The baseline model for the FMNIST is the same as the baseline model for MNIST. We train the model for 100 epochs with Adam optimize and minibatch size of 128. Here we list all the results. The test accuracy is 0.85883, test loss is 0.4715, validation accuracy is 0.8676, validation loss is 0.4363.

From the above results and Figure 3.1.2 we notice that this baseline model perform much worse for FMNIST compared to MNIST. The test accuracy has dropped nearly 10%. Yet, similar to MNIST, there is very clear overfitting for FMNIST dataset.

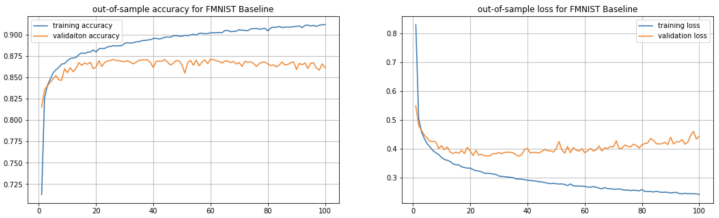


Figure 3.1.1: baseline model accuracy on FMNIST

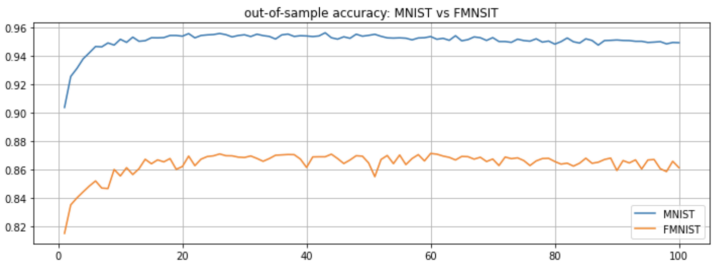


Figure 3.1.2: out-of-sample accuracy: MNIST vs F-MNIST

3.2 One layer CNN

Step 0: simple one layer CNN As shown in Figure 2.2.1, we start off by building a simple unregularized model with one convolutional and one fully connected layer. The test accuracy is 0.9089 and the loss is 1.4745, the validation loss is 1.4273.

Compared to MNIST with test accuracy of 0.9880, it still has worse performance. From the 3.2.1.2we could see that the model is overfitting even worse than MNIST, as there is a higher rising trend in the validation loss as epochs increase.

3.2.1 Regularization

For regularization, we follow the same steps as those in section 2 and try to add max pooling layer, dropout layer, batch normalization and weight decay. We also examine overfitting by looking at the trend of the graphs of loss.

Step 1: add max pooling In order to solve that, we try several regularization methods. We add a max pooling layer with pool size equal to 2*2. The testing accuracy increases to 0.9159 and the loss decreases to 0.7927, the validation loss decreases to 0.7455, and the overfitting is ameliorated from the graph.

Step 2 and 3: add dropout Afterwards, we decide to try adding a dropout layer with dropout rate as 0.5. This returns a 0.9163 testing accuracy and decreased loss (0.5014), and the validation loss decreases to 0.4705. The overfitting issue is significantly getting better, even though it still exists. We added another dropout layer with dropout rate as 0.5. After adding this layer, the test accuracy is 0.9195 and the loss is 0.2759. The validation loss is 0.2583. What is noticeable is that the overfitting is greatly reduced after adding the two dropout layers.

Step 4 and 5: add weight decay We add weight decay. In the Conv2D layer, we use kernel regularizer l2(0.001). The test accuracy is 0.9182 and the loss is 0.2839. The validation loss is 0.2669. We see that adding weight decay will improve the model. We also try different weights, l2(1e-5), and this returns a test accuracy of 0.9195 and decreased loss of 0.2842. The validation loss decreases to 0.2618. From the results, adding a weight decay l2(1e-5) could improve the model, which is the same as the result on MNIST in section 2.

Step 6: add batch normalization Finally, we try to add batch normalization. The test accuracy is 0.9157 and loss is 0.2699. The validation loss is 0.2446. The performance boosts more.

The performance of different regularization methods is shown below. What similar to the performance on MNIST is that regularization methods on FMNIST could also help prevent the problems of overfitting. After all six steps have added, there is no noticeable sign of overfitting. And each regularization step has similar effect each step in both datasets as we compare Figure 3.2.1.2 and Figure 2.2.1.3.

What different between the one-layer CNN model on the two datasets is that MNIST has a much better performance than FMNIST before and after regularization. This can be seen by comparing 3.2.1.1 and Figure 2.2.1.2.

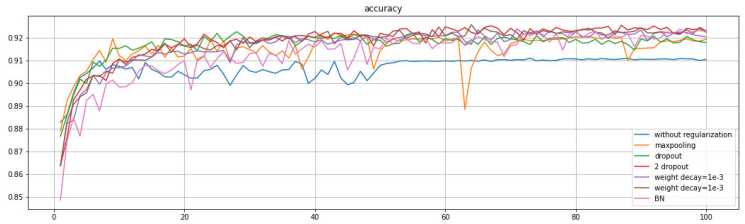


Figure 3.2.1.1: validation accuracy of different regularization methods on FMNIST

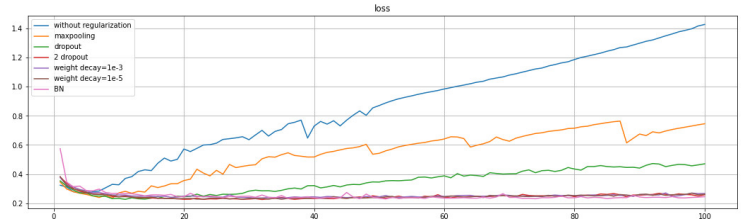


Figure 3.2.1.2: validation loss of different regularization methods on FMNIST

3.2.2 Tuning

Then, we tune parameters including output depth, kernel size, optimizer and learning rate. For FMNIST, we follow the same steps as those section 2 on MNIST.

For output depth equals 16, test accuracy is 0.9172 and loss is 0.2452. The validation loss is 0.2292. For output depth equals 64, testing accuracy is 0.9152 and loss is 0.2919. The validation loss is 0.2672. Compared with output depth equals 32 (test accuracy as 0.9157 and loss as 0.2699, validation loss as 0.2446), output depth equals 16 has higher training and test accuracy as well as a smaller loss.

Thus, we decide to use output depth as 16, which is the same as the selected output depth on MNIST.

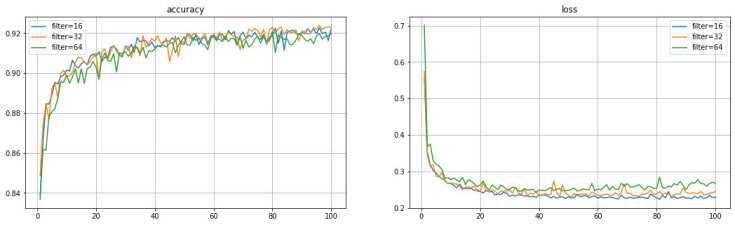


Figure 3.2.2.1: one layer model - different filters for FMNIST

After deciding on the output depth (16), we also tune the kernel size. In the above model, our kernel size is 5*5. We try an additional kernel size, 3*3. For kernel size equals to 3*3, testing accuracy is 0.9167 and loss is 0.2450. The validation loss is 0.2297.

Compared with kernel size 5*5 (testing accuracy as 0.9172 and loss as 0.2452, validation loss as 0.2292), kernel size 3*3 performs better on FMNIST. However, we choose kernel size 5*5 on MNIST in section 2. In neural networks, larger kernel size means larger receptive field, which means we have a larger overview of information of pictures and obtain better global features. Since FMNIST and MNIST are whole different dataset, the requirement of localization of a particular pattern is different, hence, the choice of kernel size may be different.

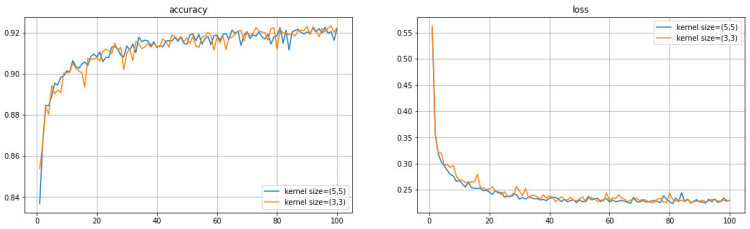


Figure 3.2.2.2: one layer model - different kernel sizes for FMNIST

After tuning filter and kernel size on FMNIST and comparing the results with MNIST, we try the performance of different optimizers: SGD and RMSprop. For SGD, testing accuracy is 0.9073 and loss is 0.2590. The validation loss is 0.2454. For RMSprop,testing accuracy is 0.9045 and loss is 0.3163. The validation loss is 0.2855. Compared with optimizer adam (testing accuracy as 0.9172 and loss as 0.2452, validation loss as 0.2292), as adam has higher training and testing accuracy as well as a smaller loss. Thus, we decide to choose adam, which is the same as MNIST in section 2.

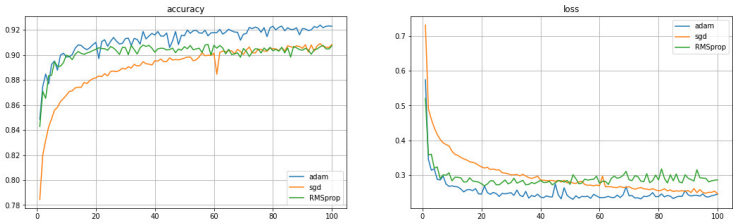


Figure 3.2.2.3: one layer model - different optimizers for FMNIST

Then, we also try different learning rate: 0.001, 0.0001. For learning rate as 0.001, testing accuracy is 0.9169 and loss is 0.2581. The validaiton loss is 0.2400. The validation loss is 0.2400. For learning rate as 0.0001, testing accuracy is 0.9161 and loss is 0.2332. The validation loss is 0.2217. Comparing these two learning rates, we see that the learning rate as 0.001 has higher training and testing accuracy but higher testing and validation loss, which is slightly different from the results on MNIST.

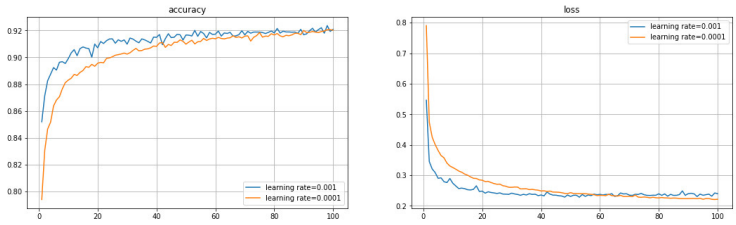


Figure 3.2.2.4: one layer model - different learning rates for FMNIST

3.2.3 Initialization

Afterwards, we also try different initializations on FMNIST, including RandomNormal class and RandomUniform class. We try random normal with mean equals 0 and std equals 0.01. Test accuracy is 0.9192 and loss is 0.2456. The validation loss is 0.2240. We then try random normal with mean equals 0 and std equals 0.1. Test accuracy is 0.9179 and loss is 0.2416. The validation loss is 0.2258. We also tried a random uniform with min equals 0 and max equals 1. Test accuracy is 0.9119 and loss is 0.2490. The validation loss is 0.2302. For random uniform with min equals -0.5 and max equals 0.5, test accuracy is 0.9106 and loss is 0.2485. The validation loss is 0.2379. All their performance is quite close to each other.

Yet, similar to MNIST, RandomNormal class performed better than RandomUniform class based on our choice of model parameters.

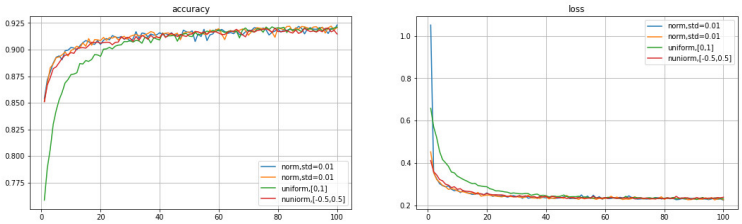


Figure 3.2.3.1: one layer model - different initializations for FMNIST

3.3 Two layer CNN

3.3.1 Architecture: adjacent convolutional layers

We add one layer of Conv2D to the best performed One-Layer CNN above to see if we can improve our model on FMNIST.

We first put two convolutional layers adjacent $twocnnA$ while keeping all other layers and parameters the same as the previous model. We notice a slight improvement in the accuracy in the test set from 0.9172 to 0.9224, and a decrease in the loss function from 0.2452 to 0.2399. This improvement is because we introduced more parameters by adding one convolutional layer, which is similar to the results on MNIST. Meanwhile, we notice that, as the epoch increases, there is a very slight sign of overfitting. We will consider reducing it in the next section.

3.3.2 Add regularization after each convolutional layer

Then, we also modify the architecture that each Convolutional layer follows by a set of regularization methods $twocnnB$. However, the test accuracy does not increase on FMNIST: 0.9224 $twocnnA$ and 0.9106 $twocnnB$, which is different from the results on MNIST. This is because MNIST and FMNIST have different image features, some improvements on MNIST do not apply to FMNIST.

3.3.3 Architecture: double the hidden units in dense layer

Following the same step in section 2, we try to add more parameters by doubling the hidden units in the last fully connected layer to see what happened.

We could see the performance of $twocnnC$ (256 units) performs (test loss 0.2405, test accuracy 0.9128) better than $twocnnB$ (128 units) (test loss 0.2483 test accuracy 0.9106) on FMNIST. While in section 2, $twocnnC$ (256 units) does not perform better than $twocnnB$ on MNIST. It is because FMNIST has more features, adding the dense units could have relatively better performance compared with MNIST.

3.3.4 Architecture: double the number of activation maps in convolutional layers

Then, we also choose to double the output depth in both Convolutional layers to compare the results on FMNIST with the results on MNIST. The test accuracy is 0.9170 and the test loss is 0.2386.

In conclusion, $twocnnA$ and $twocnnD$ have relatively higher test accuracy and lower test loss, which perform better than the other two layer Convolutional models.

There are many reasons that accounts for this phenomenon. First, MNIST and FMNIST are whole different datasets, different features are captured when training. Second, test accuracy depends on the test set, if the test dataset does not have a good coverage of the data distribution the model is trained on, the final test accuracy may be affected.

Although the result for two layer CNN on MNIST and FMNIST varies, we think this result is acceptable. And our best model is robust and generalizes well.

3.4 Data augmentation

Then we also consider Data Augmentation as a regularization method to see if we can continue improving the model performance. Like section 2, we add new augmented images to the training set to increase data diversity, which are flipped horizontally, vertically and adjusted contrast. After data augmentation, the test accuracy is 0.9247. The results for training and validation sets are below. We could see that there are dramatic fluctuations on FMNIST after data augmentation.

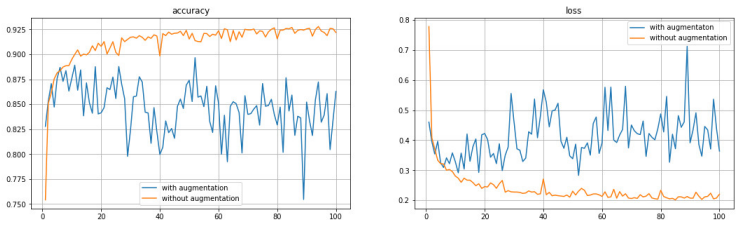


Figure 3.4.1: data augmentation for FMNIST

There are several explanations for fluctuations. First, as said in MNIST, flipping pictures horizontally and vertically is not a reasonable way to do data augmentation for FMNIST. This fails to add useful

information and instead increasing the noise. Second, after data augmentation, the validation dataset may not have a good coverage of the data distribution the model is trained on and fails to be equally distributed. After all, using data augmentation seems not to be a good idea in this dataset, or we need to find another reasonable to do data augmentation.

In general, the varied performance on MNIST and FMNIST are mainly due to the datasets themselves. The best model that built based on MNIST are robust and generalized well on FMNIST.

4 Discussion

4.1 Summary

In general, the best model we built for one layer (sequential 8) and two layer (twocnnD) based on MNIST generalized well on FMNIST. But there are some noticeable points listed below.

- For our best model (twocnnD), the test accuracy of FMNIST (about 0.92) is lower than that of MNIST (about 0.99). Since FMNIST contains more complex image features, which requires a more complex(deeper) NN architecture than MNIST.
- The optimal kernel size is 3*3 on FMNIST. However, the optimal kernel size is 5*5 on MNIST. There are some possible explanations for this. It could be that different features in different dataset are captured when training. It could also be that the test dataset and training dataset have different coverage of digits/clothes types.
- Validation accuracy and loss have more dramatic fluctuations on FMNIST than on MNIST. One of the possible explanation is that flipping pictures horizontally and vertically might not add useful information, but added noise. Another research is that the the validation dataset have different coverage of digits/clothes types.
- In the one-layer CNN model, we see that adding dropout layers significantly reduce the overfitting. Hence, dropout is one of the most efficient ways to solve overfitting.

4.2 Future Work

Since both MNIST and FMNIST are of size 28*28*1 and images are presented in the center, it is relatively easy to train the model. However, in reality, the pictures are more complex, such as 400*400 pixels with RGB channel, and the theme may not be presented in the center. In such cases, the model will have a more complex architecture if we want a high accuracy.

References

[1] Adam Byerly & Tatiana Kalganova & Ian Dear (2020) A Branching and Merging Convolutional Network with Homogeneous Filter Capsules. **arXiv:2001.0916**

[2] Jason Brownlee. "Why Initialize a Neural Network with Random Weights?", Machine Learning Mastery. Website URL: https://machinelearningmastery.com/why-initialize-a-neural-network-with-random-weights/

[3] Greeshma K.V. & Sreekumar K (2019) Hyperparameter Optimization and Regularization on Fashion-MNIST Classification. **DOI: 10.35940/ijrte.B3092.078219**

[4] Sergey Ioffe & Christian Szegedy (2015) Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. **arXiv:1502.03167**

[5] Wang Yifan & Fenghou Li & H.Sun & W. Li & Cheng Zhong & Xuelian Wu &Hailei Wang & Ping Wang (2020). Improvement of MNIST Image Recognition Based on CNN. **DOI:10.1088/1755-1315/428/1/012097**

[6] "Developer Guides." Keras Website. Website URL: https://keras.io/guides/.