DeepLearning
Term Project

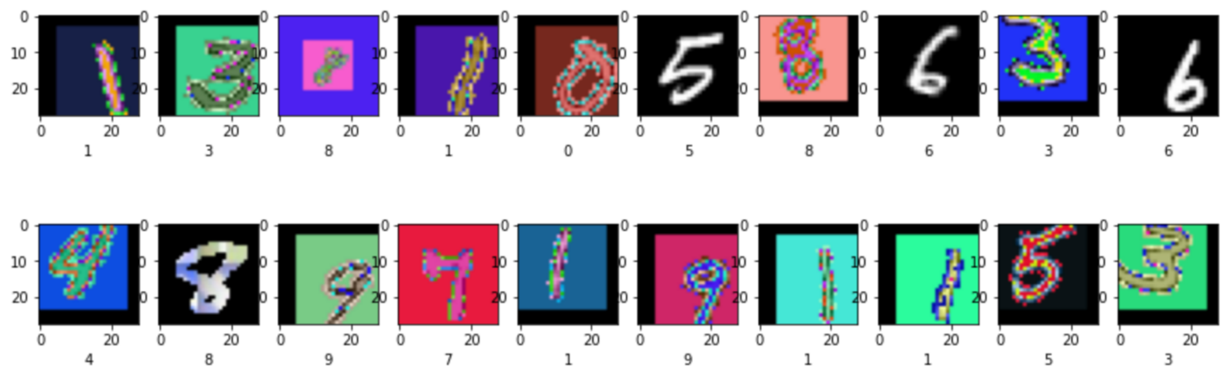
소프트웨어학부 컴퓨터전공
2017012479 이하영

1. Data

The base data is MNIST which is handwritten digits image data, and it has total 70,000 samples. Each image of it is 28x28x1, gray channel. To make my classifier general, I augmented the data with several ways.

a. Data Augmentation

I created new data, printed digits image data which has total 50 samples. I augmented both data using functions which are explained below. Plus, I used patterned data downloaded from HY-IN. Thus, the total size of the samples is 423,700 -MNIST: 360,000, new data: 3,700, patterned data: 60,000.



▲ Fig 1. Augmented data

i. Rotation

I implemented rotation function using getRotationMatrix2D function and wrapAffine function of opencv-python. The rotation function rotates an image left or right randomly in range 10 to 15.

ii. Coloring

There are two ways to color the gray images: the colouring function and the coloring function. Both functions change values of pixels, but the different is the first one adds random value to some pixels located in "digit" part. Thus, the coloured image remains the border like, while the colored one is solid.

iii. Scale

There are also two ways to scale the images: the expand function and the shrink function. As their name, the expand function increases images size 120%, and, on the other hand, the shrink function decreases its size 50% and fills border random value. Both functions are implemented with resize function of opencv-python.

iv. Shift

The shift function moves images into several directions: up, down, left, right, diagonal. The range of distance shifted is 3 to 6 because the image size (28x28) is too small to move a lot.

b. Preprocessing

I believe proper preprocessing method help a model perform better, especially if an input is image data. I experienced preprocessing MNIST data with deskewing or centralizing. Due to the great number of samples, however, I didn't do any preprocessing.

2. Implementation

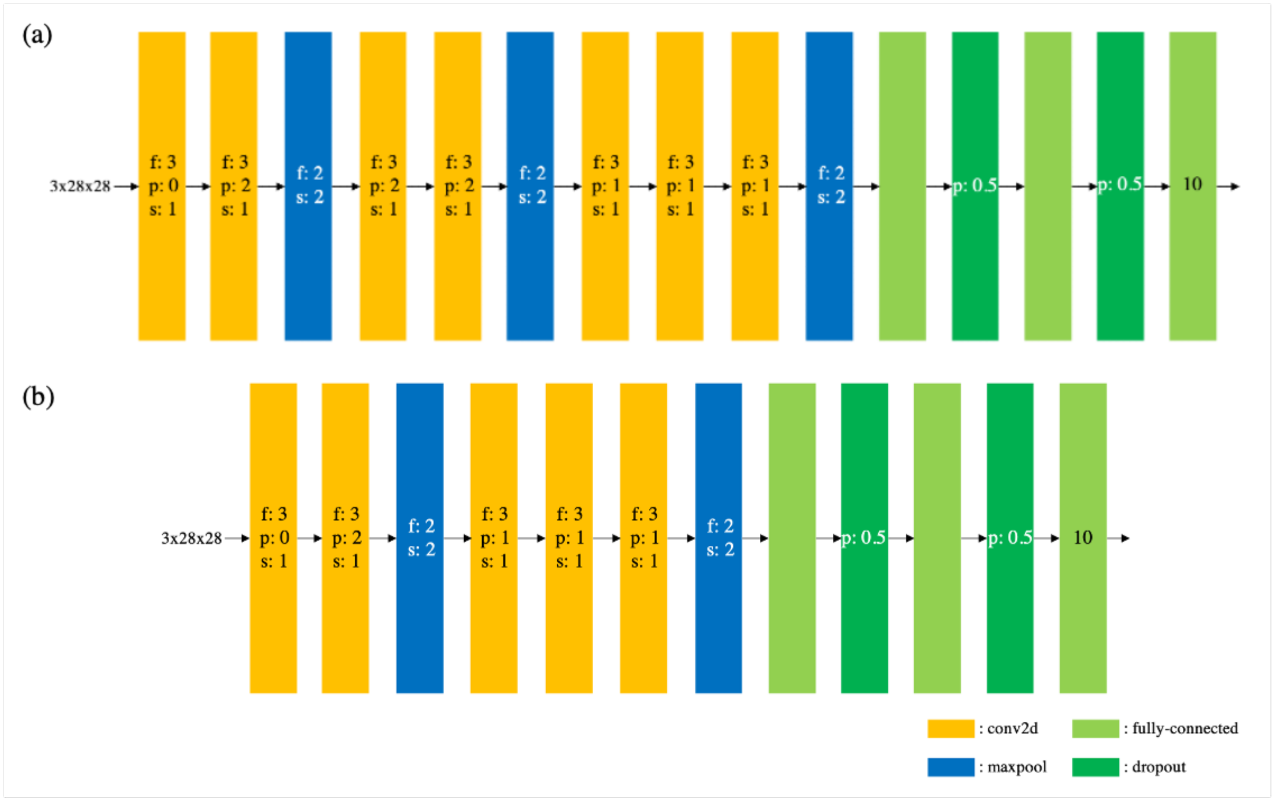
a. Environment and Dependencies

- Environment: Windows 10, GeForce RTX 3090/GeForce RTX 2070 mobile, CUDA 11.2
- Dependencies: torch==1.8.1+cu111, torchvision==0.9.1+cu111, opencv-python==4.5.2.52

b. Dataset

I randomly split it into trainset(413,700) and testset(10,000), and again split trainset into trainset(330,960) and validset(82,740).

c. Model



▲ Fig.2 Model Structures: The f, p, and s in conv2d layer means a filter size, a padding size, and a value of stride respectively. The p in dropout layer means a probability of an element to be zeroed.

I designed two digits-classifiers using convolution neural network by referencing to the AlexNet structure. See Fig2, (a) has 7 convolutional layers, 3 maxpooling layers, 2 fully-connected layers (except output layer), and 2 dropout layers. Each convolutional layer contains 3x3 filters and 1 stride value. Because the input image size is small as 28x28, I set padding value in each layer, so the size does not decrease greatly during convolution steps. Instead, through maxpooling layers with stride values 2, the machine specifies the features. Channel sizes of convolutional layers and the number of nodes in fully-connected layers are considered hyperparameters.

The activation function is the ReLU function, and it exists behind every convolutional layer and fully-connected layers except the output layer. I initialized weights as He initial value due to ReLU function.

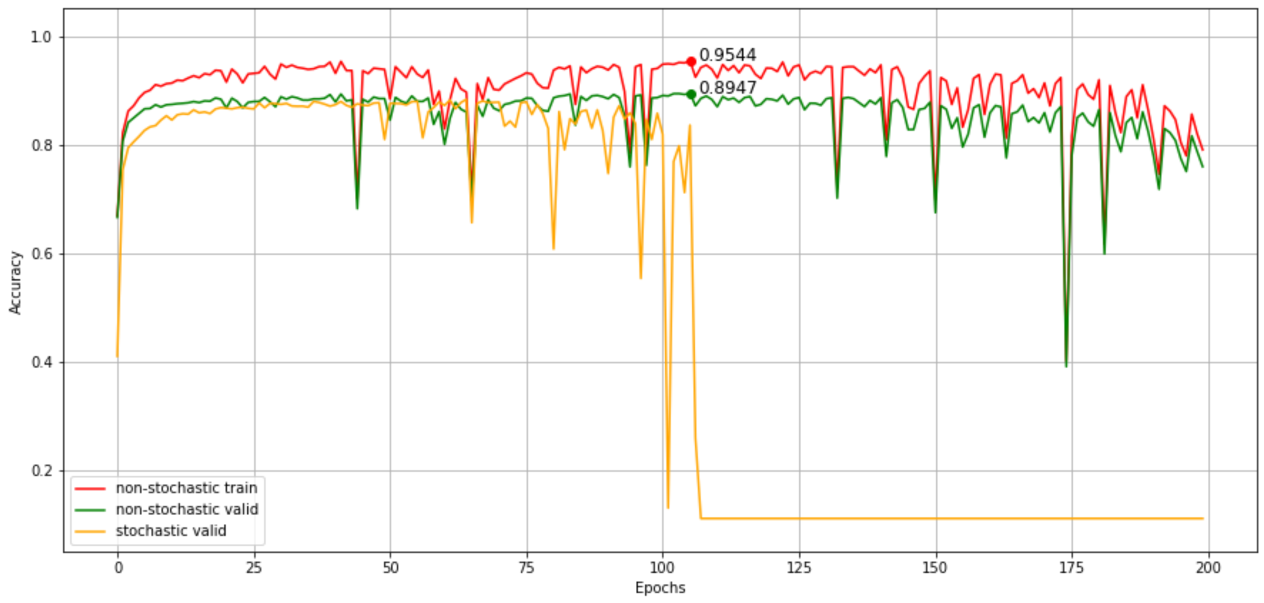
I selected the cross-entropy loss and the Adam optimizer. I decided the learning rate is 0.001 considering the running time. To avoid overfitting, I set 0.1 in the weight_decay argument of Adam optimizer. I added a trick to keep the best model which has the best accuracy of validation dataset and to save the best one. Thus, I could attain the best one even if the graph fluctuated so much or the accuracy of the end is less than the middle of.

3. Experiment

Train data size	Batch size	Epochs	Stochastic	Channels	Nodes	Accuracy	Running Time
150,000	24	100	O	[16, 32, 32, 64, 64, 128, 128]	[256, 128]	0.1127	About 2h 30min (RTX 3090)
150,000	24	100	O	[16, 16, 32, 64, 64, 96, 128]	[128, 128]	0.113	About 2h (RTX 3090)
100,000	100	200	O	[24, 32, 64, 64, 96]	[128, 128]	0.8037	About 1h (RTX 3090)
100,000	100	200	O	[16, 32, 64, 64, 128]	[128, 128]	0.1113	About 1h (RTX 3090)
330,960	100	200	O	[24, 32,64, 64, 96]	[128, 128]	0.8833	About 2h (RTX 2070 mobile)
330,960	100	200	X	[24, 32, 64, 64, 96]	[128, 128]	**0.8947**	About 2h (RTX 2070 mobile)
330,960	50	200	X	[24, 32, 64, 64, 96]	[128, 128]	0.8458	About 2h (RTX 2070 mobile)

▲ Table.1 Training case: The accuracies are results from validset. The cases with 7 channel sizes uses the model (a), and others uses the model (b).

Seeing the Table.1, each row shows the results and conditions while running the models to tune the hyperparameters. In the first and second row of the Table.1 I found that the model (a) is too big for the data, and the number of nodes in each fully-connected layer should be same even though the difference of the accuracy between them is tiny. I reduced the number of the train samples and increased the batch size due to the large running time. The model (b) whose size decreased performed better than the model (a). I also tried to run the model (b) with 256 nodes of both fully-connected layer. The parameter size of this case, however, was too large-its torch state_dict file size is more than 7MB- so I gave up although the performance was great. From the 3rd and 4th row, I got the best condition of the size of channels and nodes among various test conditions. I put entire train dataset I prepared and trained the model (b) with two cases: using randomly train data or not. As shown the Table.1, in the non-stochastic case (6th row) there is higher accuracy. I found two facts seeing the Fig.3. First, higher accuracy was obtained earlier in the non-stochastic case. Second, there was a moment that the accuracy dropped down and never recovered in the stochastic case.



▲ Fig.3 Comparison of three accuracies of dataset: The orange graph shows the accuracy of validset over epochs in the 5th row of the Table.1. The red graph shows accuracy of the trainset, and the green one shows accuracy of the validset in the 6th row.

Lastly, I ran the 6th row condition changing batch size as 50. I expected the accuracy of 50 batch size should be higher since I believe small batch size makes better model with less training time. In the case of 7th row in the Table.1, the model reaches its best accuracy earlier than the case of 6th row-the former one at 55th epoch and the latter one at 105th epoch. But the accuracy of 6th row is greater by about 5%points.

From the experiments, I finally found the best model with the condition of 6th row of the Table.1. I plotted accuracies of the trainset and the validset to check whether the model overfits. As following Fig.3, the difference at the best moment (105th epoch) is about 6%points, and I concluded the model is quite well learned.

I didn't figure out the reason of severe fluctuations in training loss over epochs. I guess that is because the maximum epoch is too large comparing with the model size.

4. Conclusion

I tested this model with the testset, and the accuracy is 0.8949. The slightly small difference of accuracies (0.02%point) between the validset and the testset implies the model seems fine. The model can classify both handwritten and printed digits with quite high accuracy.