CS5330: Project 5: Recognition using Deep Networks

Created by Sida Zhang, last modified on Apr 20, 2022

@Sida Zhang and @Hongyu Wan

April 1, 2022

Introduction

This project is about building, training, analyzing, and modifying a convolutional neural network for a classification problem. We have used the MNIST digit recognition dataset for building and training the network. The MNIST digit dataset can effectively be used as the basis for learning and practicing how to develop, evaluate, and use convolutional deep learning neural networks for image classification from scratch. This includes how to develop a robust test harness for estimating the performance of the model, how to explore improvements to the model, and how to save the model and later load it to make predictions on our new data.

The operating system that we performed is Windows11 and the Integrated development environment is PyCharm and JupyterNotebook.

We have imported the following libs for this project.

os, cv2, sys, torch, torchvision, torch.nn as nn, torch.nn.functional as F,

torch.optim as optim, matplotlib.pyplot as plt, from torch.utils import data,

from torch.utils.data import DataLoader, from PIL import Image, import numpy as np

random, math. from torchvision import models, from torchvision import transforms

from collections import Counter

Please feel free to post any comment or suggestion on this page, and thank you for viewing our Wiki Page 👍.

Over all thinking

I think the biggest challenge was to recognize self-drawing Greek symbol data of alpha, beta, and gamma. We have correctly followed the steps to crop and to rescale the data set, but the program still doesn't match their corresponding symbol examples correctly (~20% correctness only). We eventually solved this issue by thresholding both datasets (Greek symbol collection from Dr. Maxwell and self-drawing additional Greek symbol collection from ourselves). We have converted both datasets to the binary images by setting the value to only 0 or 255 which enhanced the prediction rate to 80%, I think the biggest reason was we have very different drawing pen widths from the two data sets.

Anyway, I will now introduce our project to you, and please feel free to leave a comment or a like 😃.

Tasks and functionalities

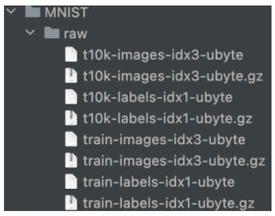
See Project Structure at the bottom for file and folder references

Task 1: Build and train a network to recognize digits

The first step of this project is to build and train a network to do digit recognition using the MNIST data. We have followed the tutorials that Dr. Maxwell provided on the Canvas site on building and training a convolutional neural network to solve the MNIST digit recognition task.

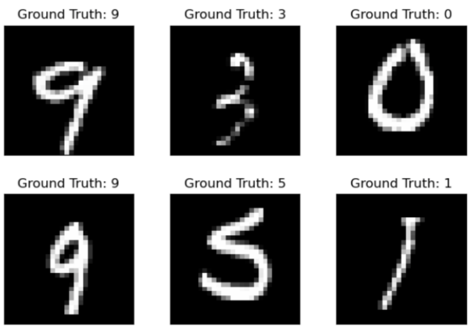
A. Get the MNIST digit data set

We have downloaded the data set directly from the torchvision package: torchvision.datasets.MNIST to local.



```
MNIST
  raw
    t10k-images-idx3-ubyte
    t10k-images-idx3-ubyte.gz
    t10k-labels-idx1-ubyte
    t10k-labels-idx1-ubyte.gz
    train-images-idx3-ubyte
    train-images-idx3-ubyte.gz
    train-labels-idx1-ubyte
    train-labels-idx1-ubyte.gz
```

We used pyplot subplot method to create the plots.



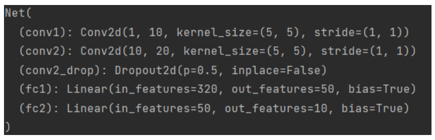
B. Make your network code repeatable
We have set the random set the seed to 42 to make the code repeatable and we have also turned off CUDA using

```
torch.backends.cudnn.enabled = False
```

for this project.

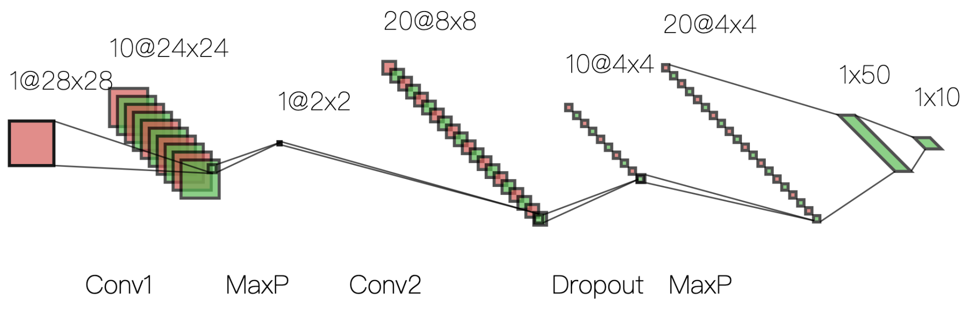
C. Build a network model

For this step, we have implemented our network with the given parameters. Below is a screenshot of our Network design in the terminal.



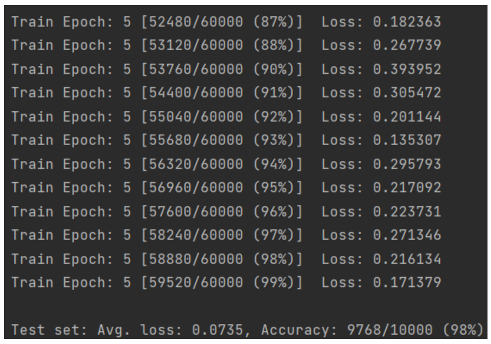
```
Net(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

Our convolutional neural network diagram is shown below (Draw by NN-SV):



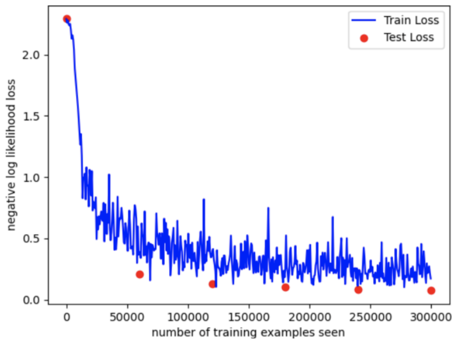
D. Train the model

This step is trained for five epochs, we have evaluated the model on both the training and test sets after each epoch (below image is a partial screenshot of the fifth, the last epoch):



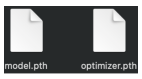
```
Train Epoch: 5 [52480/60000 (87%)]  Loss: 0.182363
Train Epoch: 5 [53120/60000 (88%)]  Loss: 0.267739
Train Epoch: 5 [53760/60000 (90%)]  Loss: 0.393952
Train Epoch: 5 [54400/60000 (91%)]  Loss: 0.305472
Train Epoch: 5 [55040/60000 (92%)]  Loss: 0.201144
Train Epoch: 5 [55680/60000 (93%)]  Loss: 0.135307
Train Epoch: 5 [56320/60000 (94%)]  Loss: 0.295793
Train Epoch: 5 [56960/60000 (95%)]  Loss: 0.217092
Train Epoch: 5 [57600/60000 (96%)]  Loss: 0.223731
Train Epoch: 5 [58240/60000 (97%)]  Loss: 0.271346
Train Epoch: 5 [58880/60000 (98%)]  Loss: 0.216134
Train Epoch: 5 [59520/60000 (99%)]  Loss: 0.171379

Test set: Avg. loss: 0.0735, Accuracy: 9768/10000 (98%)
```

The batch size we have chosen for this step is 64. The plot of this training is shown below:



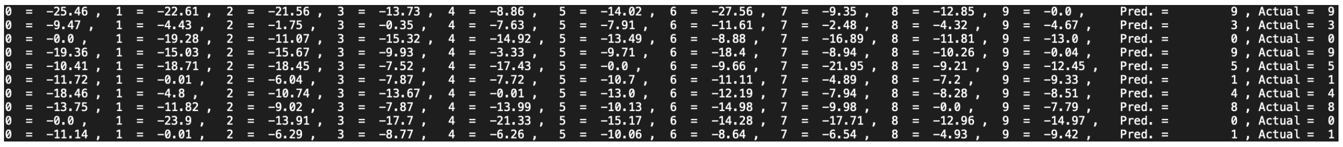
E. Save the network to file

After the network is trained, we have saved the network to ../results/networkTrained/model.pth, and ../results/networkTrained/optimizer.pth for future uses.



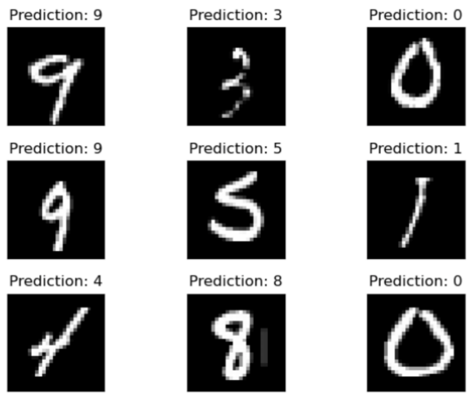
F. Read the network and run it on the test set

For this step, we have written the code in a separated py file, the program read the network that had been saved from the previous step and runs the model on the first 10 examples in the test data set. We have set the mode to evaluation before we run the samples. The program would print out the 10 output values with 2 decimal places, the index of the max output value, and the correct label of the digit.



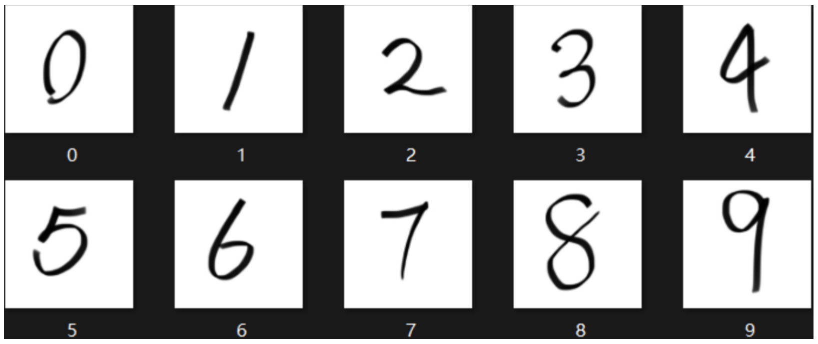
```
0  =  -25.46 ,  1  =  -22.61 ,  2  =  -21.56 ,  3  =  -13.73 ,  4  =  -8.86 ,   5  =  -14.02 ,  6  =  -27.56 ,  7  =  -9.35 ,   8  =  -12.85 ,  9  =  -0.0 ,     Pred. =         9 , Actual =  9
0  =  -9.47 ,   1  =  -4.43 ,   2  =  -1.75 ,   3  =  -0.35 ,   4  =  -7.63 ,   5  =  -7.91 ,   6  =  -11.61 ,  7  =  -2.48 ,   8  =  -4.32 ,   9  =  -4.67 ,    Pred. =         3 , Actual =  3
0  =  -0.0 ,    1  =  -19.28 ,  2  =  -11.07 ,  3  =  -15.32 ,  4  =  -14.92 ,  5  =  -13.49 ,  6  =  -8.88 ,   7  =  -16.89 ,  8  =  -11.81 ,  9  =  -13.0 ,    Pred. =         0 , Actual =  0
0  =  -19.36 ,  1  =  -15.03 ,  2  =  -15.67 ,  3  =  -9.93 ,   4  =  -3.33 ,   5  =  -9.71 ,   6  =  -18.4 ,   7  =  -8.94 ,   8  =  -10.26 ,  9  =  -0.04 ,    Pred. =         9 , Actual =  9
0  =  -10.41 ,  1  =  -18.71 ,  2  =  -18.45 ,  3  =  -7.52 ,   4  =  -17.43 ,  5  =  -0.0 ,    6  =  -9.66 ,   7  =  -21.95 ,  8  =  -9.21 ,   9  =  -12.45 ,   Pred. =         5 , Actual =  5
0  =  -11.72 ,  1  =  -0.01 ,   2  =  -6.04 ,   3  =  -7.87 ,   4  =  -7.72 ,   5  =  -10.7 ,   6  =  -11.11 ,  7  =  -4.89 ,   8  =  -7.2 ,    9  =  -9.33 ,    Pred. =         1 , Actual =  1
0  =  -18.46 ,  1  =  -4.8 ,    2  =  -10.74 ,  3  =  -13.67 ,  4  =  -0.01 ,   5  =  -13.0 ,   6  =  -12.19 ,  7  =  -7.94 ,   8  =  -8.28 ,   9  =  -8.51 ,    Pred. =         4 , Actual =  4
0  =  -13.75 ,  1  =  -11.82 ,  2  =  -9.02 ,   3  =  -7.87 ,   4  =  -13.99 ,  5  =  -10.13 ,  6  =  -14.98 ,  7  =  -9.98 ,   8  =  -0.0 ,    9  =  -7.79 ,    Pred. =         8 , Actual =  8
0  =  -0.0 ,    1  =  -23.9 ,   2  =  -13.91 ,  3  =  -17.7 ,   4  =  -21.33 ,  5  =  -15.17 ,  6  =  -14.28 ,  7  =  -17.71 ,  8  =  -12.96 ,  9  =  -14.97 ,   Pred. =         0 , Actual =  0
0  =  -11.14 ,  1  =  -0.01 ,   2  =  -6.29 ,   3  =  -8.77 ,   4  =  -6.26 ,   5  =  -10.06 ,  6  =  -8.64 ,   7  =  -6.54 ,   8  =  -4.93 ,   9  =  -9.42 ,    Pred. =         1 , Actual =  1
```

The program also plots the first 9 digits as a 3 x 3 grid with the prediction result for each example:

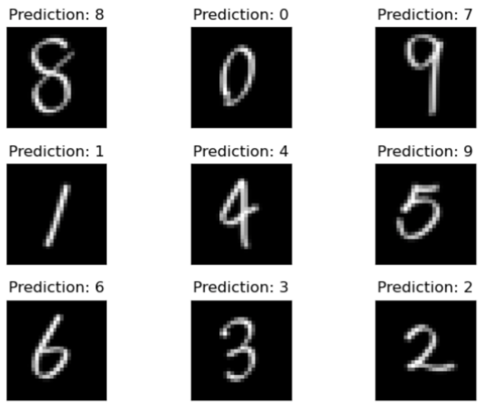


G. Test the network on new inputs

For this step, we have written out our own ten digits from 0 - 9 on iPhone. We have cropped each digit in 1284 x 1284 pixels.



We have written the code in a separate file to read the image, convert it to greyscale, resized it to 28 x 28, and run them through the network. The result of this step is below:



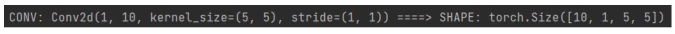
As you can tell from the above image, we have 80% accuracy in the prediction. I think "7" does look like "9" in some way, and if we reverse "5" it does look similar to "9". We will now examine our network and analyze how the network processes the data in the next task.

Task 2: Examine your network

In this task, we will examine our convolutional neural network by analyzing the hidden layers to see how the network processes the data in the following steps

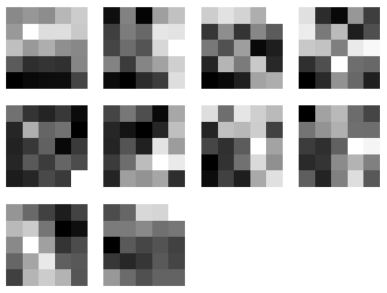
A. Analyze the first layer

For this step, the program will get the weights of the first layer in a tensor object of [10, 1, 5, 5]:



```
CONV: Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1)) ====> SHAPE: torch.Size([10, 1, 5, 5])
```

We can visualize the ten filters with pyplot (color in gist_gray) as a 3 x 4 grid image below:

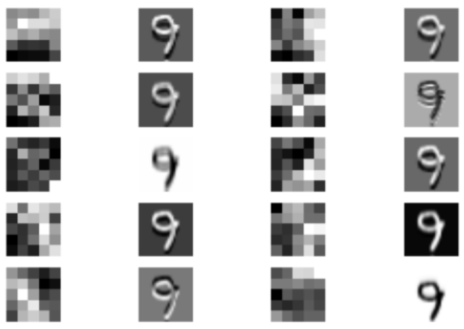


B. Show the effect of the filters:

In this step, we have used filter2D function from OpenCV to apply the 10 filters to the first training example ("9" in this case) image. We have also used

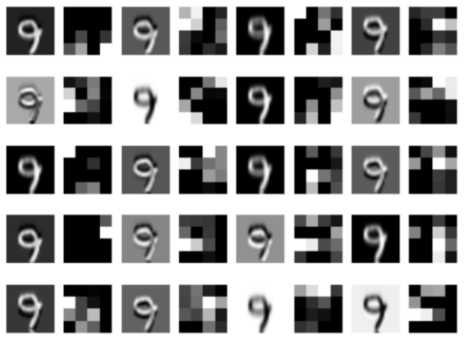
`torch.no_grad()`

function to skip calculating on gradients:



C. Build a truncated model

In this step, we have built a new sub-model but with only the first two convolutional layers. We have inherited the original model for this Submodel. The weights of the model are the same, but only the first two layers are applied to the model. We can visualize the 20 4 x4 channels with pyplot (color in gist_gray) as a 5 x 8 grid image below:



Task 3: Create a digit embedding space

This task is about using the trained network as an embedding space for images of greek symbols. The program will differentiate images of the Greek letters "alpha", "beta", and "gamma".

A. Create a greek symbol data set

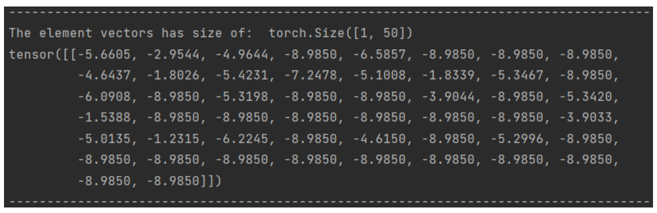
For this step, we have downloaded the collection from Canvas as our training data set. We have scaled them to 28 x 28, converted them to greyscale, inverted the intensities, and converted them to binary images. We have written the data set to two CSV files under ../results/ and created a list for adding more Greek symbols for later use.

dataset.csv

label.csv

B. Create a truncated model

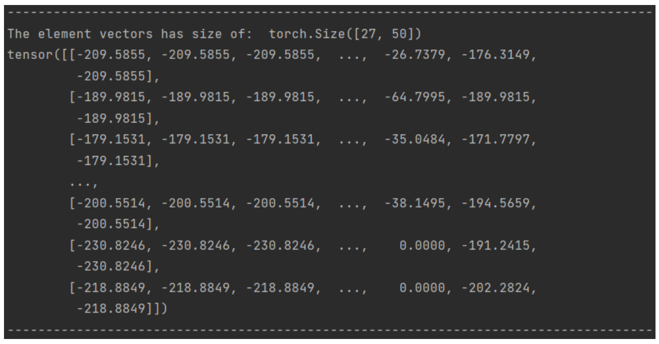
The program will read the trained model and build a new model that will stop training at the Dense layer with 50 outputs.



```
--------------------------------------------------------------------------------
The element vectors has size of:  torch.Size([1, 50])
tensor([[-5.6605, -2.9544, -4.9644, -8.9850, -6.5857, -8.9850, -8.9850, -8.9850,
         -4.6437, -1.8026, -5.4231, -7.2478, -5.1008, -1.8339, -5.3467, -8.9850,
         -6.0908, -8.9850, -5.3198, -8.9850, -8.9850, -3.9044, -8.9850, -5.3420,
         -1.5388, -8.9850, -8.9850, -8.9850, -8.9850, -8.9850, -8.9850, -3.9033,
         -5.0135, -1.2315, -6.2245, -8.9850, -4.6150, -8.9850, -5.2996, -8.9850,
         -8.9850, -8.9850, -8.9850, -8.9850, -8.9850, -8.9850, -8.9850, -8.9850,
         -8.9850, -8.9850]])
--------------------------------------------------------------------------------
```

C. Project the greek symbols into the embedding space:

We have applied the truncated network to the greek symbols to get a set of 27 x 50 element vectors by reading from the CSV file:



```
--------------------------------------------------------------------------------
The element vectors has size of:  torch.Size([27, 50])
tensor([[-209.5855, -209.5855, -209.5855,  ...,  -26.7379, -176.3149,
         -209.5855],
        [-189.9815, -189.9815, -189.9815,  ...,  -64.7995, -189.9815,
         -189.9815],
        [-179.1531, -179.1531, -179.1531,  ...,  -35.0484, -171.7797,
         -179.1531],
        ...,
        [-200.5514, -200.5514, -200.5514,  ...,  -38.1495, -194.5659,
         -200.5514],
        [-230.8246, -230.8246, -230.8246,  ...,    0.0000, -191.2415,
         -230.8246],
        [-218.8849, -218.8849, -218.8849,  ...,    0.0000, -202.2824,
         -218.8849]])
--------------------------------------------------------------------------------
```

D. Compute distances in the embedding space

Now we have the set of 27 x 50 element vectors, we can compute the sum-squared distance in the 50-dimensional embedding space between each example and all 27 examples.

For this step, we have randomly generated a number from 0 - 27 by using

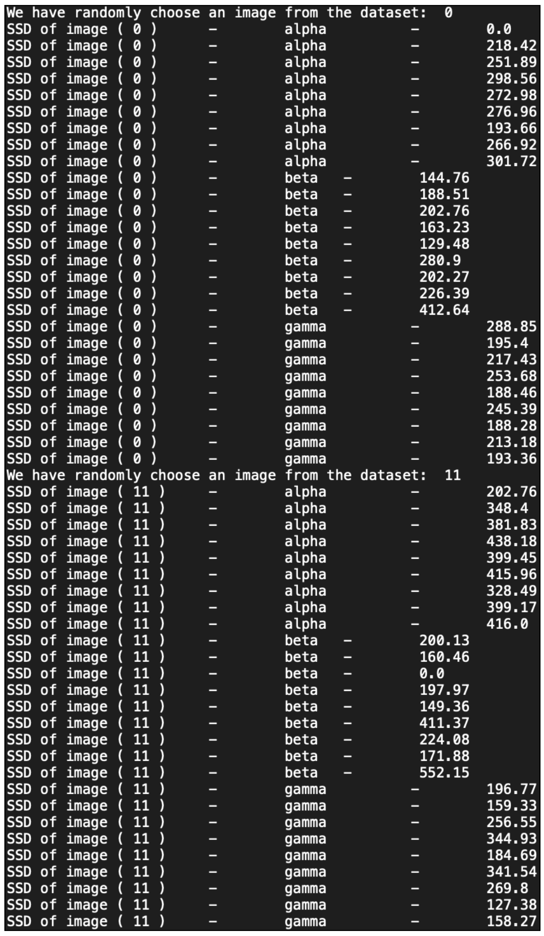
`random_randint()`

to pick one example from each set of nine:

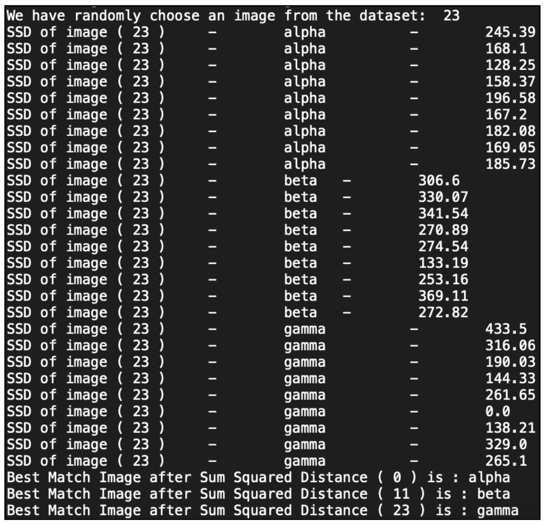
*: Because we will add more symbols to this data set, we have written a function to randomly pick a number from each set of nine symbols:

`for i in (m): rand_index.append(random.randint(n / m * i, n / m * (i + 1) - 1)) where n is size of symbols and m is size of categories.`

The result is shown below in the terminal:



```
We have randomly choose an image from the dataset:  0
SSD of image ( 0 )      -       alpha          -       0.0
SSD of image ( 0 )      -       alpha          -       218.42
SSD of image ( 0 )      -       alpha          -       251.89
SSD of image ( 0 )      -       alpha          -       298.56
SSD of image ( 0 )      -       alpha          -       272.98
SSD of image ( 0 )      -       alpha          -       276.96
SSD of image ( 0 )      -       alpha          -       193.66
SSD of image ( 0 )      -       alpha          -       266.92
SSD of image ( 0 )      -       alpha          -       301.72
SSD of image ( 0 )      -       beta   -       144.76
SSD of image ( 0 )      -       beta   -       188.51
SSD of image ( 0 )      -       beta   -       202.76
SSD of image ( 0 )      -       beta   -       163.23
SSD of image ( 0 )      -       beta   -       129.48
SSD of image ( 0 )      -       beta   -       280.9
SSD of image ( 0 )      -       beta   -       202.27
SSD of image ( 0 )      -       beta   -       226.39
SSD of image ( 0 )      -       beta   -       412.64
SSD of image ( 0 )      -       gamma          -       288.85
SSD of image ( 0 )      -       gamma          -       195.4
SSD of image ( 0 )      -       gamma          -       217.43
SSD of image ( 0 )      -       gamma          -       253.68
SSD of image ( 0 )      -       gamma          -       188.46
SSD of image ( 0 )      -       gamma          -       245.39
SSD of image ( 0 )      -       gamma          -       188.28
SSD of image ( 0 )      -       gamma          -       213.18
SSD of image ( 0 )      -       gamma          -       193.36
We have randomly choose an image from the dataset:  11
SSD of image ( 11 )     -       alpha          -       202.76
SSD of image ( 11 )     -       alpha          -       348.4
SSD of image ( 11 )     -       alpha          -       381.83
SSD of image ( 11 )     -       alpha          -       438.18
SSD of image ( 11 )     -       alpha          -       399.45
SSD of image ( 11 )     -       alpha          -       415.96
SSD of image ( 11 )     -       alpha          -       328.49
SSD of image ( 11 )     -       alpha          -       399.17
SSD of image ( 11 )     -       alpha          -       416.0
SSD of image ( 11 )     -       beta   -       200.13
SSD of image ( 11 )     -       beta   -       160.46
SSD of image ( 11 )     -       beta   -       0.0
SSD of image ( 11 )     -       beta   -       197.97
SSD of image ( 11 )     -       beta   -       149.36
SSD of image ( 11 )     -       beta   -       411.37
SSD of image ( 11 )     -       beta   -       224.08
SSD of image ( 11 )     -       beta   -       171.88
SSD of image ( 11 )     -       beta   -       552.15
SSD of image ( 11 )     -       gamma          -       196.77
SSD of image ( 11 )     -       gamma          -       159.33
SSD of image ( 11 )     -       gamma          -       256.55
SSD of image ( 11 )     -       gamma          -       344.93
SSD of image ( 11 )     -       gamma          -       184.69
SSD of image ( 11 )     -       gamma          -       341.54
SSD of image ( 11 )     -       gamma          -       269.8
SSD of image ( 11 )     -       gamma          -       127.38
SSD of image ( 11 )     -       gamma          -       158.27
```

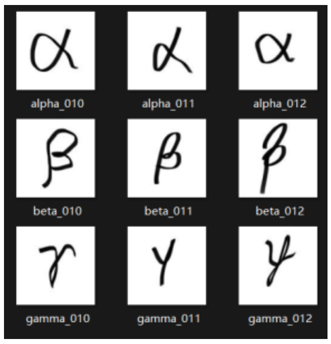


```
We have randomly choose an image from the dataset:  23
SSD of image ( 23 )     -       alpha           -       245.39
SSD of image ( 23 )     -       alpha           -       168.1
SSD of image ( 23 )     -       alpha           -       128.25
SSD of image ( 23 )     -       alpha           -       158.37
SSD of image ( 23 )     -       alpha           -       196.58
SSD of image ( 23 )     -       alpha           -       167.2
SSD of image ( 23 )     -       alpha           -       182.08
SSD of image ( 23 )     -       alpha           -       169.05
SSD of image ( 23 )     -       alpha           -       185.73
SSD of image ( 23 )     -       beta    -       306.6
SSD of image ( 23 )     -       beta    -       330.07
SSD of image ( 23 )     -       beta    -       341.54
SSD of image ( 23 )     -       beta    -       270.89
SSD of image ( 23 )     -       beta    -       274.54
SSD of image ( 23 )     -       beta    -       133.19
SSD of image ( 23 )     -       beta    -       253.16
SSD of image ( 23 )     -       beta    -       369.11
SSD of image ( 23 )     -       beta    -       272.82
SSD of image ( 23 )     -       gamma           -       433.5
SSD of image ( 23 )     -       gamma           -       316.06
SSD of image ( 23 )     -       gamma           -       190.03
SSD of image ( 23 )     -       gamma           -       144.33
SSD of image ( 23 )     -       gamma           -       261.65
SSD of image ( 23 )     -       gamma           -       0.0
SSD of image ( 23 )     -       gamma           -       138.21
SSD of image ( 23 )     -       gamma           -       329.0
SSD of image ( 23 )     -       gamma           -       265.1
Best Match Image after Sum Squared Distance ( 0 ) is : alpha
Best Match Image after Sum Squared Distance ( 11 ) is : beta
Best Match Image after Sum Squared Distance ( 23 ) is : gamma
```

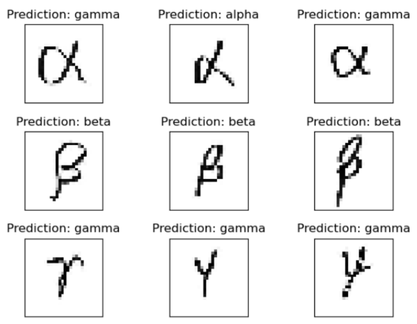
Since the result's Sum-Squared Distance with the matching itself should be 0, the results here are very accurate (100%) because the input data is indeed from the training data. I have found that the average Sum-Squared Distance of each category does not follow a specific rule, and I think the K-Nearest Neighbor classifier would give a much better result. We will have the K-Nearest Neighbor classifier implemented later in the extensions.

E. Create your own greek symbol data

We have added more greek symbols to our data set under ../data/greek_input/:



The matching result of their corresponding symbol example is shown below:



Task 4: Design your own experiment

The final task we have implemented is to try some experimentation with the deep network for the MNIST task.

A. Develop a plan

Plan A (81 combinations): For this task, we have picked 4 dimensions (L * M * N * T), in a total of 81 different combinations with automation in order to optimize the network performance. The variations we have used in this task are listed below:

Batch Size	Epochs	Learning Rate	Dropout Rate
64	1	0.001	0.125
128	5	0.01	0.25
256	10	0.1	0.5

Plan B (three convolutional layers): We have also tried a different experiment on changing from two convolutional layers to three convolutional layers.

B. Predict the results

Before we start running the evaluation, we want to come up with some hypothesizes for our plan:

Plan A:

Batch size: I believe batch size is of the easiest parameters to adjust. If the batch size is too small, it would be too late to converge the network; and if the batch size is too big, it would take a lot of memory spaces.

Epochs: I personally think the more epochs we have the better results we would get.

Learning rate: we think the lower the learning rate is, the slower the loss function changes. While using a lower learning rate ensures that we don't miss anything, it also means a longer time to converge. However, if the learning rate is too high, the network would be too difficult to converge.

Dropout Rate: I would often adjust this hyperparameter to tunning the results but I am really interested in seeing the difference the dropout rate can make in the network.

Plan B:

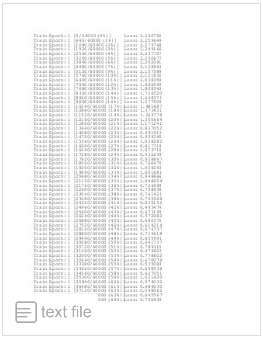
We assume three convolutional layers with the same parameters will have a better training result with lower average loss and better accuracy.

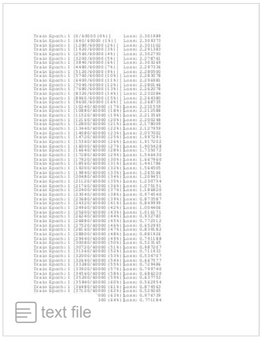
C. Execute your plan

Plan A: After ~3 hours of running our plan (mostly because of the 10 epochs), we finally have the result printed out and saved to a CSV file below:

exp_result.csv

Plan B: The best result is to have a 256 batch size, 10 epochs, 0.1 learning rate, and 0.125 dropout rate in this case. However, after averaging out, I found that more epochs do not promise better results. Every parameter plays their roles in training a network. And I believe this is the reason why we do tun on network training all the time in deep network learning. See results in the below txt files (conv_original.txt, conv_multi3.txt):


text file


text file

Extensions

Extension 1: Live video digit recognition application

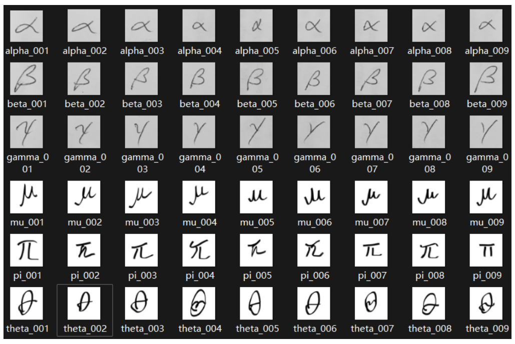
For this extension, we have implemented a live video digit recognition application that inputs the frames and outputs the prediction in the terminal. The application will print the prediction in the terminal every 10 frames. Please watch the below video link for more descriptions.

digit_recognition_p5.mp4

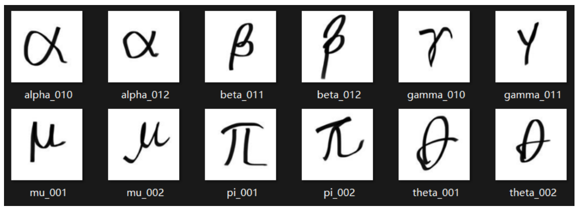
Extension 2: More greek letters and with K-nn classifier

We have selected three more Greek letters "Pi", "Theta", and "Mu", to our training data set and the test data set for this extension. We implemented a K-Nearest Neighbor classifier system where parameter K is 3. The classifier first calculates the Sum-Squared Distance from the training sets and stores them in an array. Then we calculated the frequency (up to 3) by using "Counter()" python built-in function and return the matching with the most frequencies.

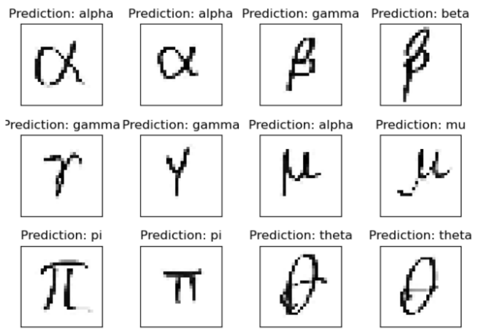
Our training data set (55 images) :



Our testing data set(12):



K-Nearest Neighbor classifier result with (80% accuracy):



K-Nearest Neighbor classifier indeed provides better accuracy in our case.

CSV files for K-Nearest Neighbor classifier:

dataset.csv

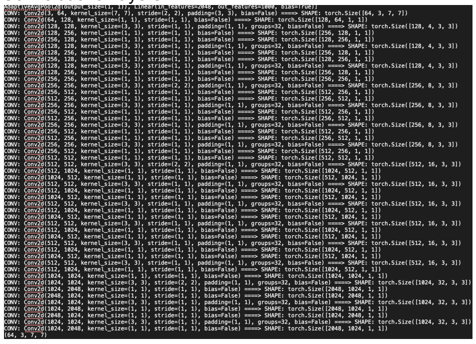
label.csv

Extension 3: Pre-trained networks available in the PyTorch package

For this extension, we have tried the ResNet pre-trained networks from torchvision package. We first included from torchvision import models and we have also downloaded the model from url

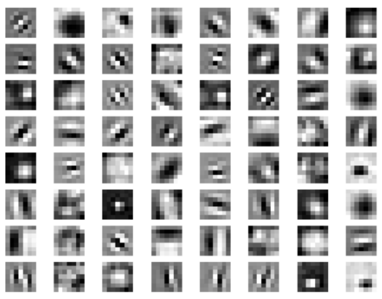
```
'resnext50_32x4d': 'https://download.pytorch.org/models/resnext50_32x4d-7cdf4587.pth'
with
torch.hub.load()
```

Convolutional layers:

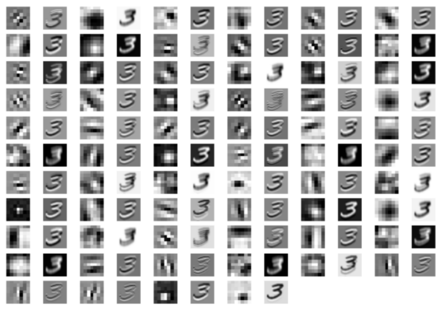


```
AdaptiveAvgPool2d(output_size=(1, 1)), Linear(in_features=2048, out_features=1000, bias=True)]
CONV: Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False) ====> SHAPE: torch.Size([64, 3, 7, 7])
CONV: Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 64, 1, 1])
CONV: Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([128, 4, 3, 3])
CONV: Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 128, 1, 1])
CONV: Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 256, 1, 1])
CONV: Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([128, 4, 3, 3])
CONV: Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 128, 1, 1])
CONV: Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 256, 1, 1])
CONV: Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([128, 4, 3, 3])
CONV: Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 128, 1, 1])
CONV: Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 256, 1, 1])
CONV: Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([256, 8, 3, 3])
CONV: Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 256, 1, 1])
CONV: Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 512, 1, 1])
CONV: Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([256, 8, 3, 3])
CONV: Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 256, 1, 1])
CONV: Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 512, 1, 1])
CONV: Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([256, 8, 3, 3])
CONV: Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 256, 1, 1])
CONV: Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 512, 1, 1])
CONV: Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([256, 8, 3, 3])
CONV: Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 256, 1, 1])
CONV: Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 512, 1, 1])
CONV: Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([512, 16, 3, 3])
CONV: Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([1024, 512, 1, 1])
CONV: Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 1024, 1, 1])
CONV: Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([512, 16, 3, 3])
CONV: Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([1024, 512, 1, 1])
CONV: Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 1024, 1, 1])
CONV: Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([512, 16, 3, 3])
CONV: Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([1024, 512, 1, 1])
CONV: Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 1024, 1, 1])
CONV: Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([512, 16, 3, 3])
CONV: Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([1024, 512, 1, 1])
CONV: Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 1024, 1, 1])
CONV: Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([512, 16, 3, 3])
CONV: Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([1024, 512, 1, 1])
CONV: Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 1024, 1, 1])
CONV: Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([512, 16, 3, 3])
CONV: Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([1024, 512, 1, 1])
CONV: Conv2d(1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([1024, 1024, 1, 1])
CONV: Conv2d(1024, 1024, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([1024, 32, 3, 3])
CONV: Conv2d(1024, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([2048, 1024, 1, 1])
CONV: Conv2d(2048, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([1024, 2048, 1, 1])
CONV: Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([1024, 32, 3, 3])
CONV: Conv2d(1024, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([2048, 1024, 1, 1])
CONV: Conv2d(2048, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([1024, 2048, 1, 1])
CONV: Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False) ====> SHAPE: torch.Size([1024, 32, 3, 3])
CONV: Conv2d(1024, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([2048, 1024, 1, 1])
(64, 3, 7, 7)
```

Then we have plotted the result with pyplot:



And the result of the first MNIST example:



Project Reflection:

I think most of the class materials were very complicated, but this assignment definitely helps on understand the concepts. The biggest thing I have learned from this project is that all parameters play their roles in training a network and all of them are important. And I believe this is the reason why we do tun on network training all the time in deep network learning. Also, in this project, I have adapted much different knowledge and concept from previous projects and classes. I believe the combination of all the course works; we can have better learning of Computer Vision.

Project Structure

```
- Project 5:

       - data

          - greek_data

               - 27 (3x9) png images

          - greek_input

               - 9 (3x3) png images

          - handwritting

               - 9 (1x9) png images

       - extension

          - extension_test

               - 12 (2x6) png images

          - extension_train

               - 66 (6x11) png images

       - results

          - extension_csv

               - dataset.csv

               - label.csv

          - greek_symbol_csv

               - dataset.csv

               - label.csv

          - networkTrained

               - model.pth

               - optimizer.pth

          - conv_multi3.txt

          - conv_original.txt

          - customized_results.csv

       - src

          - extension1_classifier.py

          - extension2_livevideo.py

          - extension3_resnet.py

          - task1_handwritting.py

          - task1_main.py

          - task1_testset.py

          - task2_analysis.py
```

- `task3_main.py`
- `task4_main.py`
- `task4_multiconv.py`

Acknowledgment:

Thanks to Dr. Maxwell and all the posts and discussions on Piazza.

Dr. Maxwell's CV notes and class materials

http://alexlenail.me/NN-SVG/LeNet.html

https://invideo.io/make/online-video-editor/

https://nextjournal.com/gkoehler/pytorch-mnist

https://medium.com/dataseries/visualizing-the-feature-maps-and-filters-by-convolutional-neural-networks-e1462340518e

https://datascience.stackexchange.com/questions/32651/what-is-the-use-of-torch-no-grad-in-pytorch

https://pytorch.org/vision/0.8/_modules/torchvision/models/resnet.html

https://pytorch.org/tutorials/beginner/basics/intro.html

No labels