Project 5: Recognition using Deep Networks

Name: Seunghan, Lee / Maik Katko

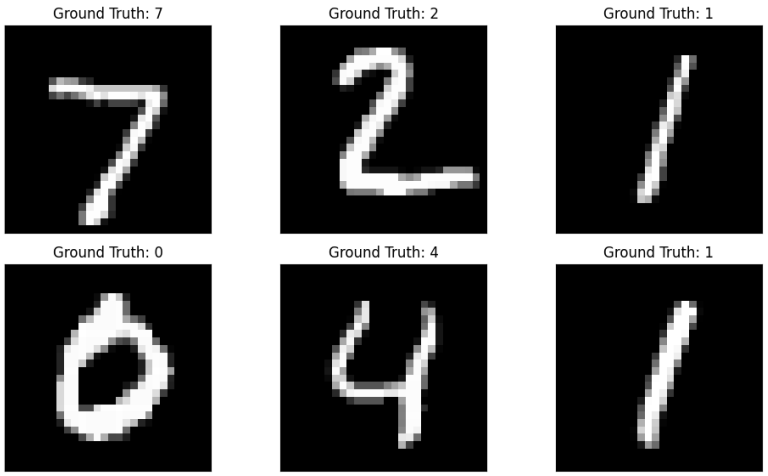
Short Description: This project is to get familiar with using deep learning framework pytorch for computer vision. First task was to build a custom network that classifies digits from the MNIST digit dataset, and train the model and test it with the test dataset. Next task was to explore the first convolution network that we have trained and understand how each filter look like. We also visualized the filter applied images. 3rd task was to apply transfer learning to a Greek letter dataset. Lastly we have implemented automated testing script that allowed us to find optimal hyper parameters for training MNIST fashion dataset.

Reflection: Working with state of the art PyTorch Framework to train and test deep learning task was exciting. I have read about transfer learning and had an understanding of how it works but never attempted to do transfer learning on any models that I previously played with. Also, exploring the weight filter and understanding them in the perspective of traditional computer vision was fun and helped me with intuitive understanding of deep learning in CV. Lastly, we learned that training image quality is very important. When trying to train for additional Greek letters, the performance was very poor because the images used for the additional letters were too small, resulting in mostly incorrect predictions.

Acknowledgements: We have used mainly PyTorch Docs, github pages, stackoverflow as main source of learning. We also used a free Greek letter dataset on Kaggle, but the images were too small. Working in python in IDE allowed us to easily navigate to the source code and read docs and the function parameters.

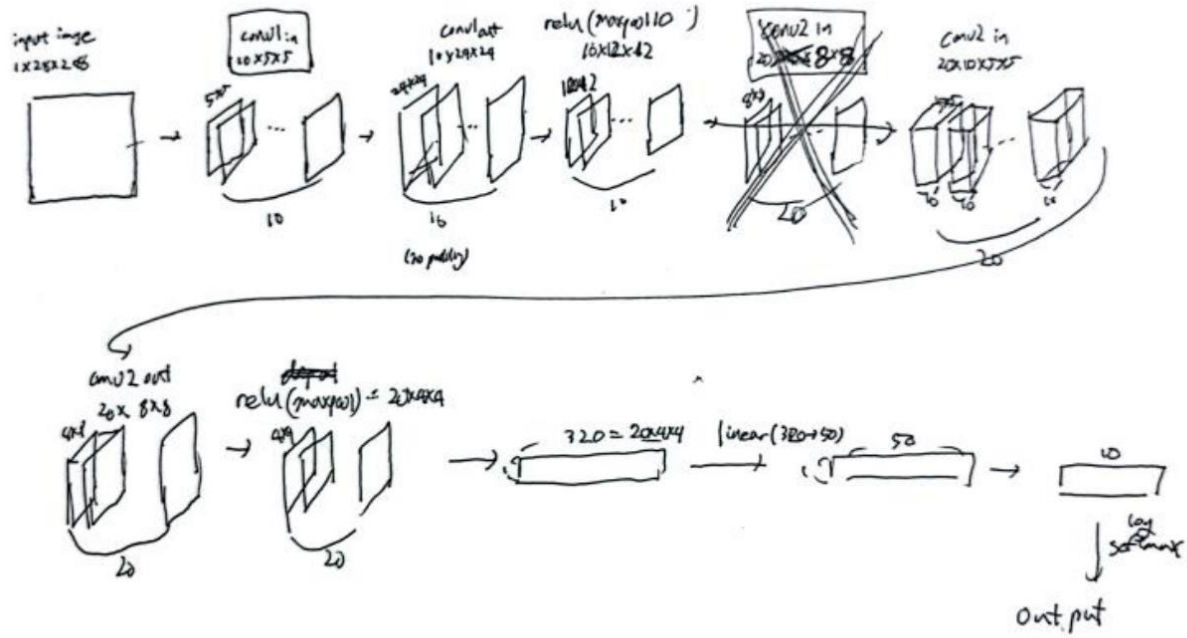
1.A: Get the MNIST digit data set

Below is the first 6 example.



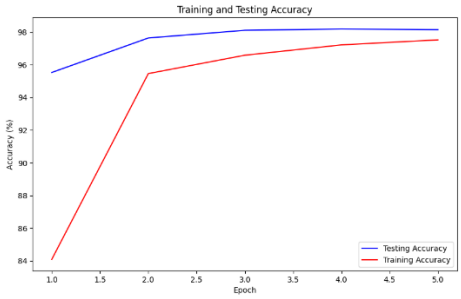
1.B: Build a network model

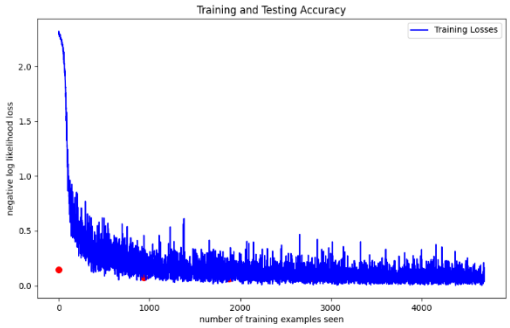
Handwritten architecture is included below.



1.C: Train the model

Based on the plot, training accuracy has exponentially increased as well as the test accuracy. It seems like there is no overfitting. We could say it as an example of benign overfitting as model we built generalizes well to the test set.

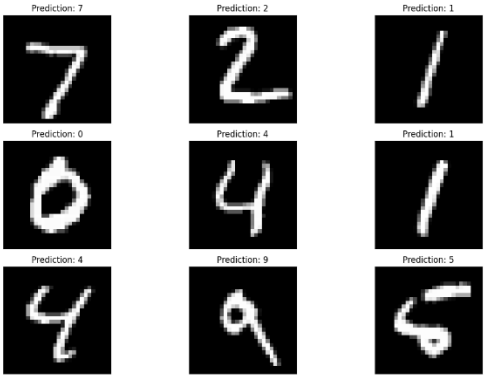




1.E: Read the network and run it on the test set

Our model has learned the pattern of the handwritten numbers quite well. For the 9th image, it seems like it can be both

interpreted as 5 or 6, and our model has classified it as 5 instead of 6.

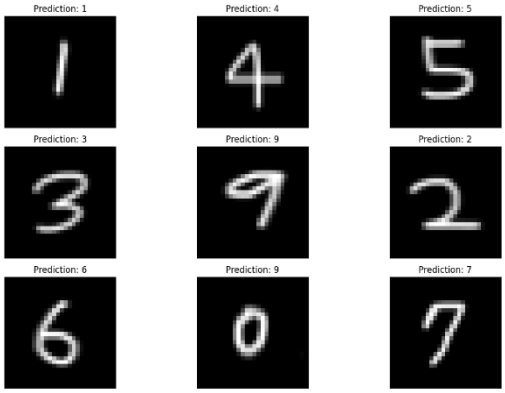


1.F: Test the network on new inputs

Our model has successfully classified all the digits except for the 0 digit. It classified it as 9 instead of 0. It seems likely

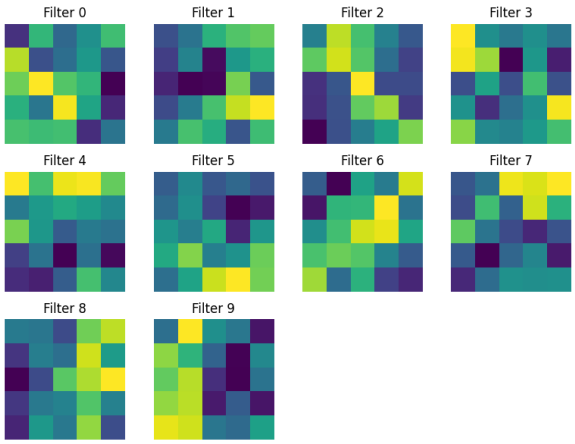
that if the digit was written thicker the result would have been more accurate. Also, I believe the model weight that we trained on

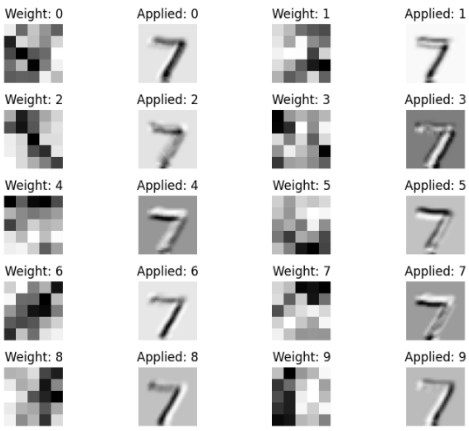
had less stronger edge detection on the lower part where the digit 9 and 0 makes difference.



2.A: Analyze the first layer

Each filter is an output of the first convolution layer of size 5x5 where each patch in the original image had 10 different channel filters, meaning each channel filter had different functionality of gathering the feature of that specific patch. It seems like for each filter, some of the areas have higher contrast then rest of the area(i.e: filter1 (3,3)). It is similar to the sobelX, sobelY-like gradient filters for edge detection.

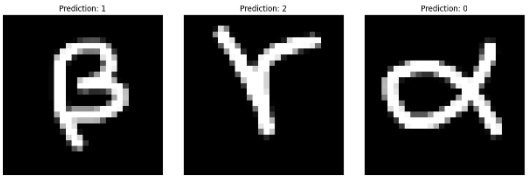


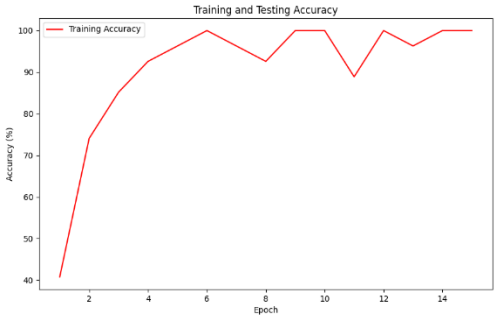


3. Transfer Learning on Greek Letters

After running 15 epochs, we have learned that something around 5~6 epochs were enough to do the transfer learning. For running the experiment on our training data, it has successfully classfied all three variation as can be seen in the image.

```
$ python greek_network.py
MyNetwork(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (dropout): Dropout2d(p=0.25, inplace=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=3, bias=True)
)
```



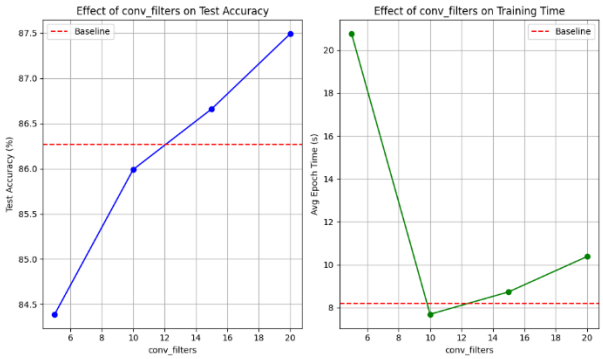


4. MNIST Network Architecture Experiment

This experiment aims to analyze how changes to the neural network architecture affect the performance of MNIST digit recognition. We'll systematically vary three key dimensions of the network architecture to understand their impact on both accuracy and training. We analyze number of convolutional filters, dropout rate, and batch size to see the impact on performance. Hypothesis for each of these is as follows:

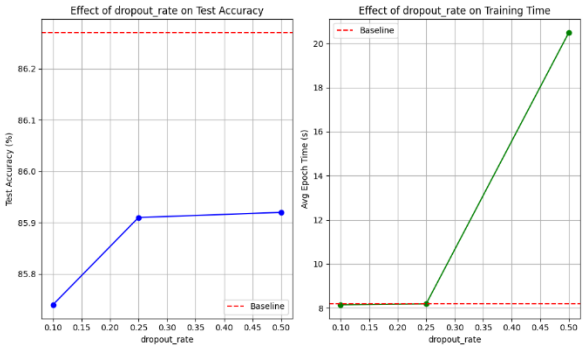
Number of Convolutional Filters

Hypothesis: Increasing the number of convolutional filters will improve model accuracy up to a point, after which we'll see diminishing returns. However, more filters will always increase training time approximately linearly.



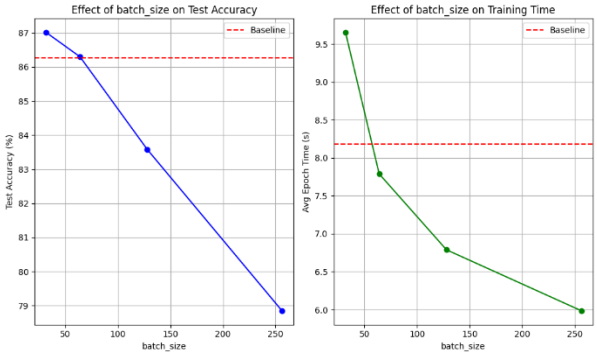
Dropout Rate

Hypothesis: A moderate dropout rate (around 0.25) will yield the best test accuracy. Too low (0.1) and the model might overfit; too high (0.5) and it might struggle to learn effectively. Training time should be relatively unaffected by dropout rate.



Batch Size

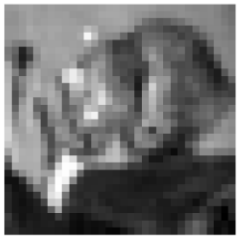
Hypothesis: Larger batch sizes will train faster in wall-clock time but may achieve slightly lower accuracy. The optimal batch size for accuracy will likely be in the middle range (64-128).



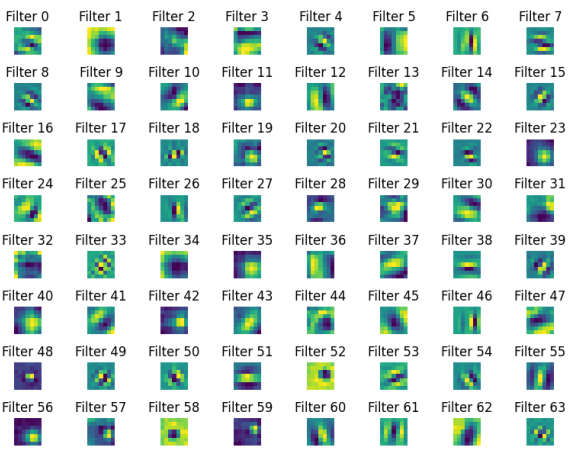
Extension:

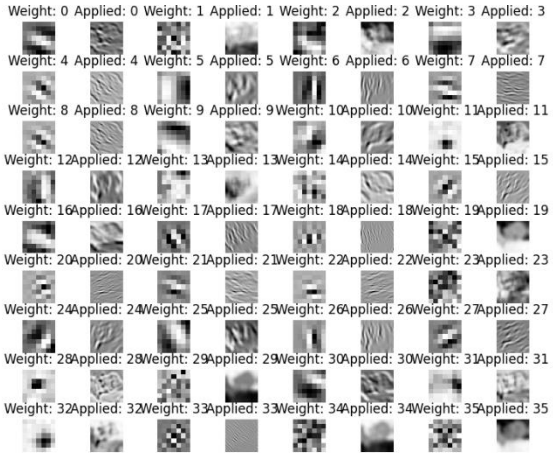
1) **Resnet conv1 layer analysis**

   We have conducted filter weight analysis(64x7x7) on the first image of the CIFAR dataset (3x32x32). CLI of the whole network was too long so we have only included several lines. It is intriguing to see the apparent



```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
```





2) Additional Greek Letters Analysis

We created functionality to evaluate delta, epsilon, eta, theta, and zeta in addition to alpha, beta, and gamma. The results ended up being very poor because the dataset we used for the extended letters had images that were too small and thus most of them were misidentified as seen in the results below. In order to improve this in the future, we will need to use larger images for training and testing.