Purvi Thakor – Individual Final Report

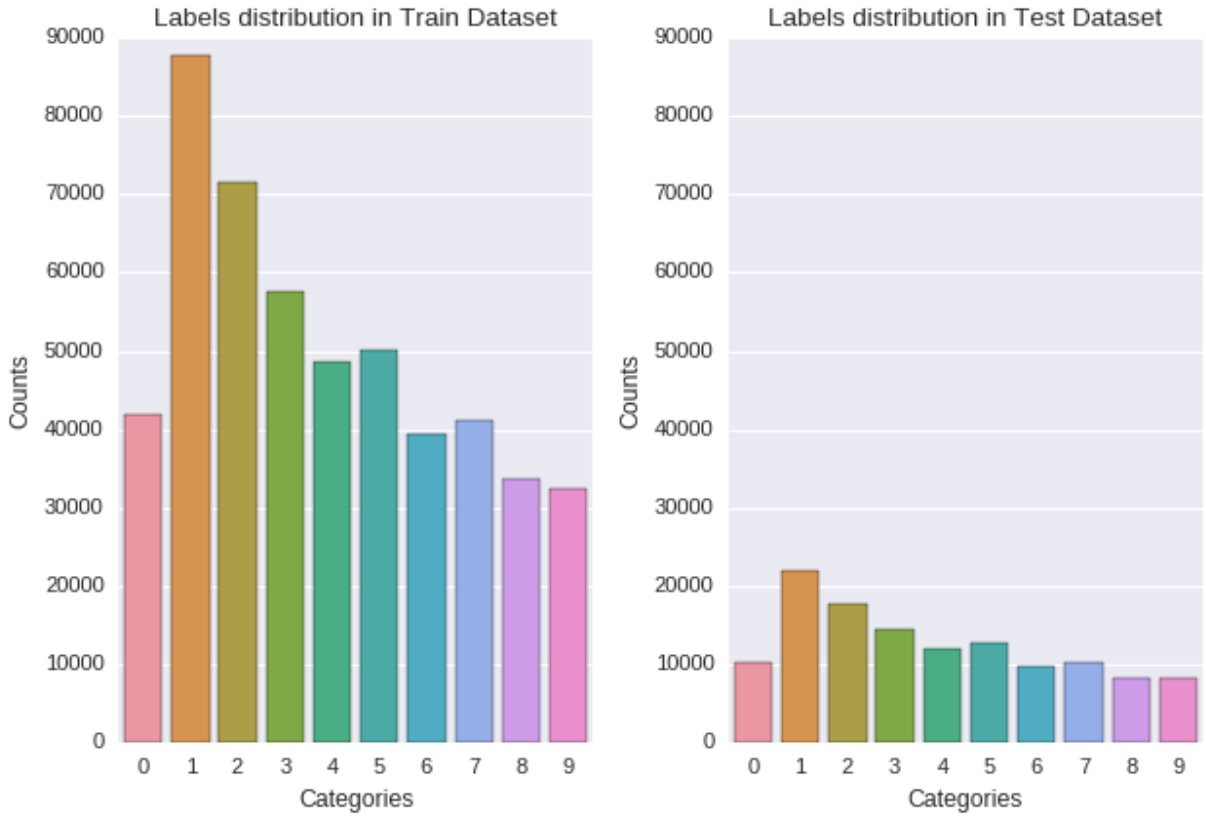
Introduction

We used the Street View House Numbers (SVHN) Datasets for our final project. This is a real-world image dataset for developing object detection algorithms. It is similar to the MNIST dataset but has more labeled data (over 600,000 images). The data has been collected from house numbers viewed in Google Street View. This dataset is 1.4 GB. It is large enough to train a deep network.

The datasource hosts 3 sets of data which have 10 unique classes, 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10. It has 73257 digits for training, 26032 digits for testing, and 531131 additional, somewhat less difficult samples, to use as extra training data It comes in two formats:

1. Original images with character level bounding boxes.
2. MNIST-like 32-by-32 images centered around a single character (many of the images do contain some distractors at the sides)

We decided to use the 32-by-32 images for our models. We did a bit of EDA on the dataset & found that the classes are unevenly distributed.

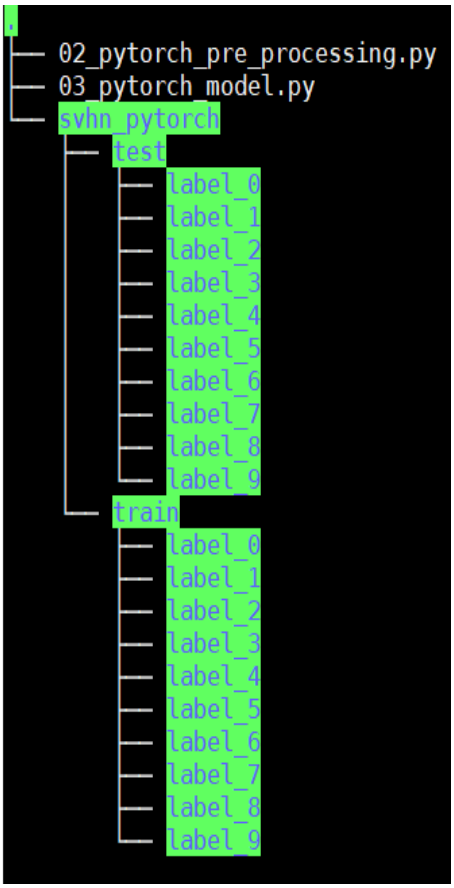


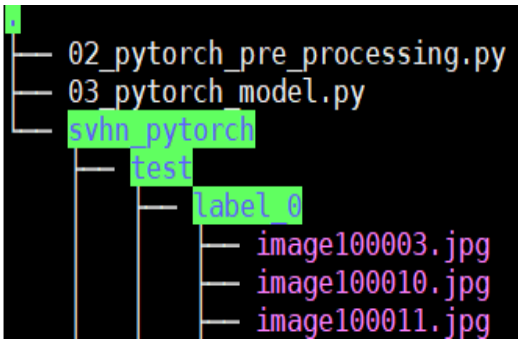
Data pre-processing:

Since we had 3 datasets (Training, Testing & Validation) we decided to concatenate them together into one single dataset. Because we combined the 3 datasets before splitting it into train and test, we reshuffled the data so that the splits would have random images. After reshuffling, we split it into train (80%) and test (20%), used histogram equalization to improve the contrast of our images & exported the outputs into test and train .mat files.
We then uploaded our data to the cloud and moved onto working on different frameworks.

PyTorch:

Pytorch requires that the directories be setup in the following structure so that it can associate the images with the correct label.





Data loading in Pytorch is easy. First, I applied a data transformation to convert the scale of the image from [0,255] to [0,1] and also convert it from .jpg to a torch tensor in a single step using '*transforms.ToTensor()*'. Next, I loaded the images using ImageFolder. Dataloader basically takes care of loading the images in the specified batch sizes which is later used to enumerate through the train and test loader.

I trained the 504,336 training images using two layers comprising of Convolution, ReLU, and Max pooling. Using the below architecture with 10 number of epochs, a batch-size of 500, and a learning rate of 0.001.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding =0))
        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding =0))
        nn.Dropout(p=0.2),
        self.fc1 = nn.Linear(in_features=6*6*32, out_features=10)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        return out
```

Here, in_channels in the first layer mean the depth of the channels (i.e 3 for color images), out_channels are the number of feature maps produced, in the second layer these out channels are the input to the second convolution layer.

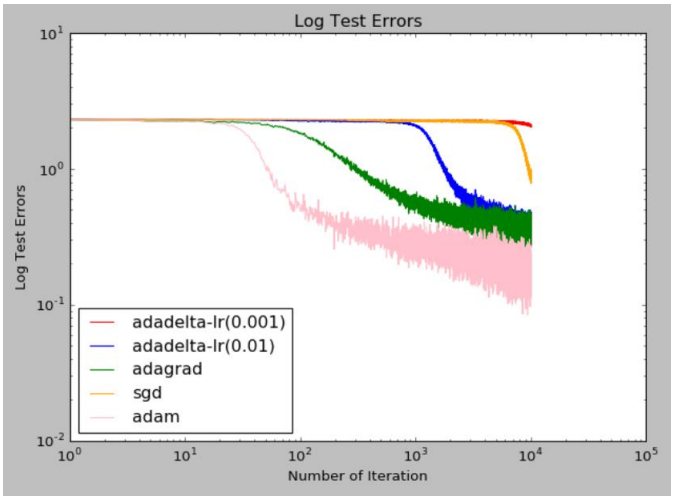
The choice of using Cross-Entropy Loss function was straightforward because it combines Log Softmax and Negative Log Likelihood Loss function. The way that Cross-Entropy works in Pytorch is that it converts the outputs to probabilities using the log softmax function. The Loss functions expect a class index in the range of 0 to C-1 where C is the number of classes. This was also one of the reasons to convert the label 10 in our data to label 0.

I liked working on Pytorch mainly because one can explicitly define the network parameters. Understanding how the math works during forward propagation is crucial to supplying those values in the CNN network.

Optimizers:

Comparing the various optimizers, it turns out that Adam optimizer converged faster than others. To update the network weights, I chose the Adam (Adaptive Moment Estimation) optimizer algorithm. Compared to Stochastic Gradient Descent which maintains a single learning rate for weight updates, the Adam algorithm is based on adaptive learning rates which in practice requires less tuning and gives better, quicker results.

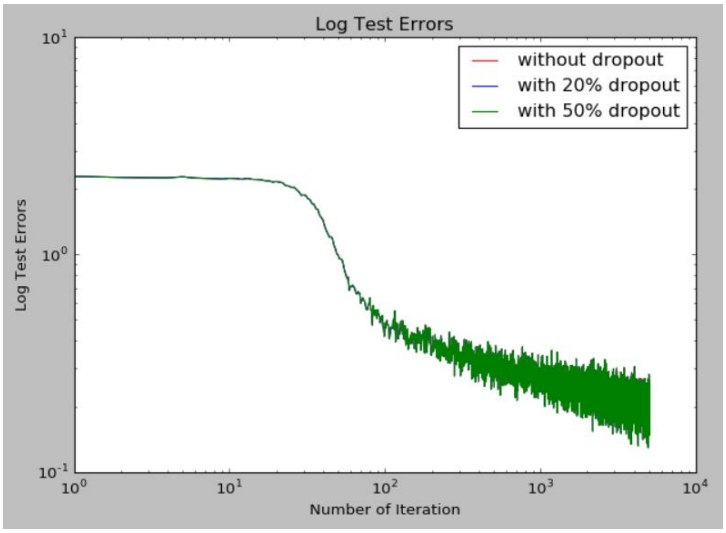
Optimizer	Test Accuracy	F1 score	Time(mins)
Adam	95%	0.948	27.19
Adagrad	89%	0.891	26.8
Adadelta(lr = 0.01)	89%	0.892	26.65
SGD	77%	0.748	26.61
Adadelta	29%	0.136	26.62



Dropouts:

Drop out is done to prevent over-fitting and dependency of neurons on each other. Drop-out means to ignore or shut off some neurons (along with their connections) during the training phase. During dropout, these nodes are not considered in forward or backward pass. Nodes are dropped with a probability of p. This leaves a training network with less number of neurons in the network. Testing is done with the full network without dropout nodes.
As can be seen in the below figure, adding a dropout of 20% and 50% keep probability did not affect the log errors. This meant that the model was not overfitted.

Dropouts	Test Accuracy	F1 score
0.2	95%	0.948
0.5	95%	0.948
0	92%	0.918



Confusion Matrix:

The confusion matrix is such that for every element in **ith** row and **jth** column, the number is equal to total number of images that belong to label i but are classified in label j. Thus, the diagonal elements give the number of images that are correctly classified by the model and off diagonal elements represent misclassifications. The actual labels are represented as rows and the predicted values are represented as columns.
Predictions are based on 4 classification parameters (general definitions):

1. TN / True Negative: case was negative and predicted negative
2. TP / True Positive: case was positive and predicted positive
3. FN / False Negative: case was positive but predicted negative
4. FP / False Positive: case was negative but predicted positive

Precision: Accuracy of correctly classifying a label
Precision = TP/ (TP +FP)

Recall: Fraction of labels that were correctly classified.
Recall = TP/(TP+FN)

		PREDICTED										
		label 0	label 1	label 2	label 3	label 4	label 5	label 6	label 7	label 8	label 9	RECALL
TARGET	label 0	9868	133	39	17	53	12	137	21	24	114	0.95
	label 1	55	21235	75	51	228	32	44	164	23	27	0.97
	label 2	22	107	17217	79	102	40	30	69	53	111	0.97
	label 3	9	128	135	13532	66	239	76	57	100	79	0.94
	label 4	11	224	74	31	11771	16	51	33	13	48	0.96
	label 5	10	50	41	185	40	12004	176	11	33	41	0.95
	label 6	60	51	19	60	65	121	9233	15	93	15	0.95
	label 7	10	338	157	51	59	11	16	9593	14	29	0.93
	label 8	36	68	48	139	55	67	152	12	7744	94	0.92
	label 9	77	82	74	65	59	53	26	28	62	7667	0.94

Our dataset had class imbalances and probably that is why some of the labels have higher recall as compared to others.

Conclusion:

This project was indeed a good learning experience to apply all the things I learned this semester. Though my part on this project seems similar to the exams, there were quite a few challenges that we came across when working on a dataset of our choice. The first and foremost was pre-processing the data. It took us quite some time to come up with appropriate pre-processing that made sense to us. Once we managed that, loading a custom data in Pytorch was something that I had never done before (We used built-in datasets in the exams). Nevertheless, that seems like an easy part now. I still need to have a better understanding of how each loss and optimizer function works.

Code:
I worked on similar lines of Exam1 and therefore, most of my code is from Dr. Amir's github. The rest of the code is about data manipulation and working with lists, dictionaries, and numpy arrays.

Future work:

Since, MNIST and SVHN both are datasets about Digit Classification, I would really like to try working on using weights from MNIST and use those on the SVHN data. I would also like to explore more architectures that could help to classify these images.

References:

https://discuss.pytorch.org/t/using-your-own-data-for-pytorch/20193/3
https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/
http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf
https://discuss.pytorch.org/t/convolution-input-and-output-channels/10205
http://cs231n.github.io/convolutional-networks/
https://www.analyticsvidhya.com/blog/2018/02/pytorch-tutorial/
https://codetolight.wordpress.com/2017/11/29/getting-started-with-pytorch-for-deep-learning-part-3-neural-network-basics/
https://stackoverflow.com/questions/38135950/meaning-of-weight-gradient-in-cnn
https://pytorch.org/docs/master/torchvision/datasets.html#imagefolder
https://github.com/xpzouying/animals-classification/blob/master/animals-classification.py
http://adventuresinmachinelearning.com/convolutional-neural-networks-tutorial-in-pytorch/