# Image Classification and Unsupervised Image Object Removal in the Street View House Numbers Dataset

# Group 7 Final Project Report

**Bill Grieser -- Darshan Kasat -- Shivam Thassu**

# Introduction

In this project, we will use three Neural Network Frameworks that we encountered in class to classify the Street View House Numbers (SVHN) dataset. (Netzer, et al., 2011) The three frameworks are:

* TensorFlow
* Caffe
* PyTorch

In addition, we will use the by-products of the PyTorch classification model to attempt unsupervised object removal of digits in the SVHN test data set.

We chose this dataset because it gave us the opportunity to utilize Convolutional Layers and other topics that we covered in class. The SVHN is considerably more complicated than the MNIST dataset, as it contains confounding objects in the images, the digits appear in different angles in the images, and the dataset is in color.

For the Image Object Removal, the goal of this task it to use by-products of the PyTorch classification network to draw inferences about the foreground and background region of a SVHN digit image, and use the inferences to train a Generalized Regression Neural Network using a Radial Basis transfer function to regenerate the image in a photorealistic manner but without the foreground object in it – in this case, the digit.

For the image classification task, we will measure success by the overall accuracy rate in classifying the test data. We have set a target of 90% accuracy. For the Object Removal task, the goal is to explore the concept and determine areas of further study.

# Description of the Data

The SVHN data was collected by Netzer, et al. from the Street View images in Google. In their work, they took a two-step approach: first, identify the digits in an image, and then, classify the digit as 0 through 9. In the data they publish, we are relying on their first step: identifying the bounding boxes in images that contain a digit. We are addressing the second task, which is to recognize the specific digit. (Note: This data was also used in a Kaggle challenge but we have obtained the data from the website of the original project and we have done our own pre-processing).

The data provided by Netzer, et al. are a set of parent images containing a street view image that includes digits. Metadata for each image identifies one or more digits within the parent image.

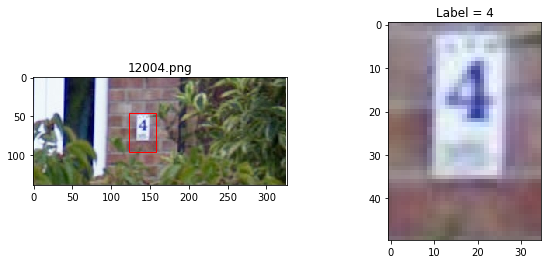


Figure 1 Example of Parent Image from the dataset and the individual digit extracted by our preprocessing

(The bounding box is not in the original but is generated based on the metadata). To see several examples of parent images with their digits extracted based on the metadata, in the code directory, run:

python3 read\_pickles.py

This will display several examples of the data.

The data is provided in two sets: a training set with 33,402 parent images and a test set with 13,068 parent images. These contain 73,257 individual digits in the training set and 26,032 digits in the test set.

# Image Classification Task

For the image classification task, we used three frameworks to train models.

## Caffe

## TensorFlow

## Pytorch

This section describes the approach taken with PyTorch to classify the digits in the SVHN dataset. **Note: to run any of the associated python code, the files expect that they will be run from the current directory, so be sure to cd to the /code/pytorch folder from our repo before trying to run them.**

### Deep Learning Network and Training Algorithm

The strategy used for the PyTorch network is to use convolutional layers connected to Batch Normalization and feeding a Relu transfer function and then a MaxPool layer. These layers feed a fully-connected layer which uses a linear transfer function across 10 output classes.

To improve the design of the ultimate model, several candidate networks were created and compared to each other. The models varied the size of the convolutional kernels in the layers, the number of kernels per layer, the number of fully-connected layers, and the use and placement of dropout layers. Several runs were made with the networks, and with different batch sizes, learning rates and optimizers, in order to find the most effective combination.

### Experimental Setup

The code specific to the PyTorch model is in the /code/pytorch folder in the repository. The pytorch folder contains the code for both the digit training and the image infill tasks.

**For the prediction task:**

* **train\_predictor.py –** Manages the training of a prediction network; accepts command line arguments to select a network architecture, the number of epochs, the batch size, the optimizer, and the learning rate. This does not produce graphical output and so it may run in the background.
* **predictor\_nets.py –** Defines several PyTorch networks as subclasses of Module with different architectures that are used by train\_predictor.py.
* **see\_pytorch\_cm.py –** This displays the results for a model trained by train\_predictor using matplotlib.
* **make\_run\* --** bash script files used to run train\_predictor.py with different combinations of inputs.

(The other files will be discussed in the Image Infill section).

The help section for train\_predictor.py lists the options for the command line arguments:

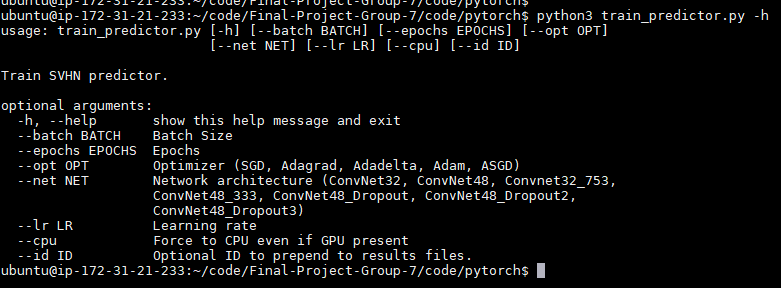


Figure 2Command-line arguments to train\_predictor.py

The train\_predictor.py file does not produce graphical or interactive output. This allows it to be run without a terminal. The approach taken was to create a bash script containing several invocations of train\_predictor.py with different input parameters. The bash script would be started with the nohup command and pushed to the background, allowing it to run without a terminal. One of several different combinations could take up to 4 hours to run. These scripts were run on a AWS instance we set up at the beginning of the class. The PyTorch tensors were targeted to the GPU if available and would fall back to a CPU if none were present (or if forced by the –cpu flag). The use of the GPU was extremely important – this was much quicker than using the CPU only.

One run of train\_predictor.py results in three files being written to the results folder in the /code/pytorch folder. In this way a record of the run could be kept for later evaluation. The files all start with a stem based on the python file that generated them and a timestamp for the run, and have different file name endings. The files are:

* **<stem>\_results.txt --** The model and parameters used and the overall accuracy achieved.
* **<stem>\_measures.csv –** Data collected at each epoch including the total loss during the epoch, the validation accuracy against 2000 samples from the test data, and the elapsed time duing the training run.
* **<stem>.pkl –** The model parameters saved after training, for use by the see\_pytorch\_cm.py script and also by the image infill task.

**Overfitting**

Two main over-fitting countermeasures were used. First, a dropout layer was employed and this ended up being the difference between the second- and first-place models. The dropout layer prevents the model from becoming over-reliant on any particular input. As we saw through the year and again on the final, using a dropout layer can improve the performance against the unseen test data. The other counter measure was to monitor the validation accuracy of the model being trained on a subset of the training data. If during a run the validation accuracy gets higher but then starts to go down as the epochs continue, this is evidence of overfitting. If we had seen this in the data, we would have implemented an early-stopping regime. It was seen at a mild level but not enough to go to implement early stopping.

**Minibatches**

During training, several mini-batch sizes were tried: 16, 32, 128, and 256. The models with 256 did run faster but they did not perform as well as the smaller batches. Having more frequent weight updates improved the overall performance of the model. There was not a difference between the sizes of 16 and 32, so after this experimentation, we used a batch size of 32 for later runs.

**Learning Rate**

We experimented with three learning rates: 0.001, 0.01, and 0.005, The smallest rate took too long to show progress, while the larger 0.01 rate had too many abrupt shifts in validation accuracy, and we felt that 0.005 performed the best.

### Results

The best performing run had these characteristics:

**Architecture:**

****

Figure 3Best-Performing PyTorch Network

The network featured three convolutional-rulu-MaxPool sequences with kernel sizes of 48, 64, and 32, followed by a Dropout layer at p=.50 and a fully connected layer. (In second place was the same network but without the dropout). This used an SGD optimization function and a learning rate of 0.005 over 100 epochs.

The network achieved a 91.6% accuracy rate on the test data.

The best performing digit was “2” and the worst was “8”.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Label (The Digit) | Precision | Recall | F1 | # Samples |
| 0 | 0.89 | 0.94 | 0.91 | 1744 |
| 1 | 0.90 | 0.95 | 0.92 | 5099 |
| 2 | 0.96 | 0.94 | 0.95 | 4149 |
| 3 | 0.92 | 0.87 | 0.89 | 2882 |
| 4 | 0.93 | 0.93 | 0.93 | 2523 |
| 5 | 0.94 | 0.91 | 0.92 | 2384 |
| 6 | 0.89 | 0.90 | 0.90 | 1977 |
| 7 | 0.92 | 0.90 | 0.91 | 2019 |
| 8 | 0.92 | 0.85 | 0.89 | 1660 |
| 9 | 0.87 | 0.91 | 0.89 | 1595 |
| avg/total | 0.92 | 0.92 | 0.92 | 26032 |

Figure 4 PyTorch Best Classififaction Summary

The model overpredicted 1 and 6 at the highest rates. Interestingly there were many more “1”’s than other digits. Perhaps the model needs mor balanced training data.

# 

Figure 4 PyTorch Best Model Confusion Matrix

The confusion matrix shows that the highest number of classification errors were classifying a “7” as a “1”. Humans may very well also have difficulty with this on some of the images where parts of the 7 were cut off. In general, however, the confusion matrix shows strong results, with the diagonal column of prediction-matching-actual being the dominant result.

**Model Performance during the training run**

This chart shows for the model that had the best accuracy how the validation accuracy and model loss performed during the training run. The loss and accuracy appear in general inversely related, which is desirable. The validation accuracy appears to be leveling off without retreating, a sign that overfitting is not occurring. This chart helped convince us to not implement early stopping while also not increasing the number of epochs past 100.

This is for a training a network very similar to the winning network but without dropout:

There is evidence of overfitting here – notice how the validation accuracy is tailing down as the loss continues to decline. The model is lowering loss but what it is learning is not generalizable. If adding dropout had not reversed the overfitting, this would have been a candidate to implement early stopping.

# Digit Removal from Images

# Summary and Conclusions

# References

Alilou, V., & Yaghmaee, F. (2015). Application of GRNN neural network in non-texture image inpainting and restoration. *Pattern Recognition Letters*, 24-31.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., & Ng, A. Y. (2011). Reading Digits in Natural Images with Unsupervised Feature Learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning.* Retrieved November 18, 2018, from http://ufldl.stanford.edu/housenumbers