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.

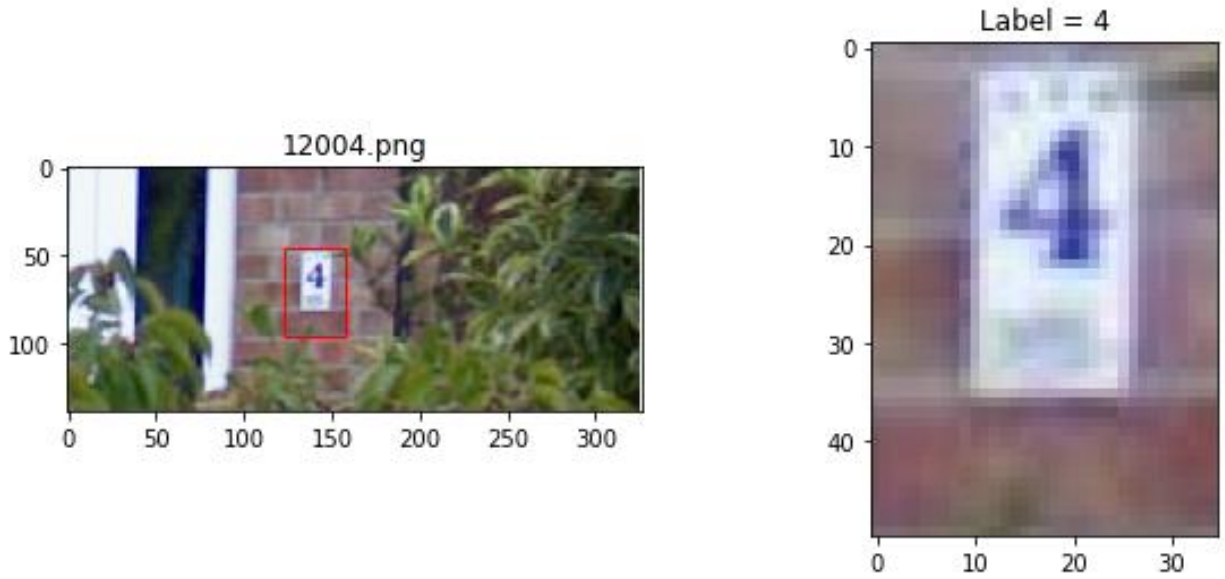


FIGURE 1 EXAMPLE OF PARENT IMAGE FROM THE DATASET AND THE INDIVIDUAL DIGIT EXTRACTED BY OUR PREPROCESSING

(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, which will be discussed in turn.

Caffe

This section includes the description of the model built in caffe to do the classification task. The code can be found the /code/caffe folder in the repository.

Data Preprocessing

Pre-processing the 32-by-32 images from the SVHN dataset centered around a single digit. In this dataset all digits have been resized to a fixed resolution of 32-by-32 pixels. The original character bounding boxes are extended in the appropriate dimension to become square windows, so that resizing them to 32-by-32 pixels does not introduce aspect ratio distortions.

The steps I had to take to get the data ready are described as follows:

1. Caffe takes a Lightning Memory-Mapped Database file as input for training the model. The initial step was to build a lmdb database and put the data into that database. The data was available in a matlab file as a 4D array. The shape of the data was to be changed in the following format:

   (Width, Height, Channel, Number of images) -> (Number of images, Channels, Width, Height)

2. The digit “0” in the dataset was labelled as “10”, hence the labels were fixed before the data was inserted in the database.
3. Normalization: Once I got the data into the database, the next part was to normalize the images. I wrote a script to calculate the mean of the data in the training set. The script generates a binaryproto file that is feed to the model in the input layer i.e., the data layer.

Deep Learning Network and Training Algorithm

The Caffe network is designed using the following strategy: a convolution layer is feeding in to a Relu transfer function which is connected to a pooling layer (average). This setup is further connected to a batch normalization layer before the input goes into the second convolution layer.

The design of the final model was then decided by tweaking the network parameters. I tried various batch sizes, learning rates and optimizers to see the changes in the model accuracy.

Experimental Setup

The code required to run the caffe model is in /code/caffe folder in the repository. The folder contains the code for crating lmdb database, data preprocessing, and model training.

These are the following files required to build a model for prediction:

svhnDatareader.py: This file manages to read the data from matlab file and then create a lmdb database for training and testing in the same working directory. This file also handles the mislabeling of digit “0” before putting it in the database.

preprocessing.py: This file calculates the mean of the images in training dataset. It generates a file named “mean.binaryproto” which is further used in the model architecture for normalization.

svhnet_train_test.prototxt: This model architecture is defined in this file.

svhnet_solver.prototxt: This is a solver file. The model parameters such as batch size, learning rate, optimizer etc. are defined in this file.

train_svhnet.py: This is the training file. It takes in the solver and starts training the model. The model is evaluated in this file.

Model Architecture:

The final model architecture is shown in the following figure:

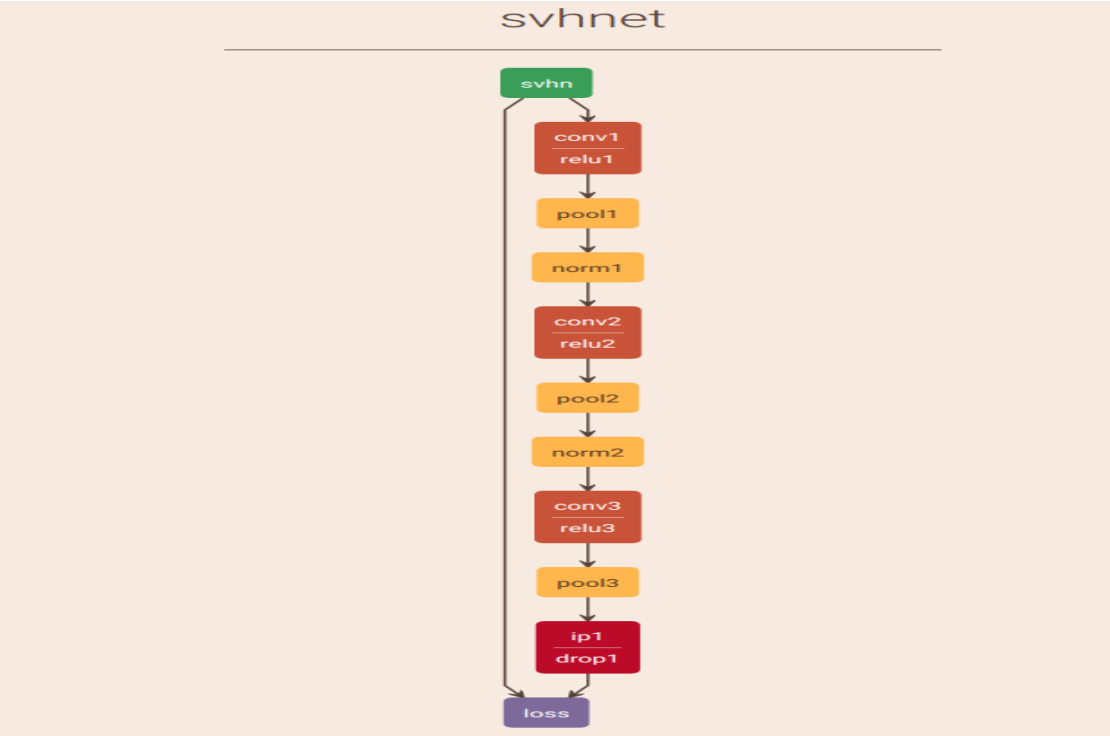


FIGURE1 BEST ARCHITECTURE CAFFE

This network architecture comprises of three convolution-relu-pooling layers. The batch normalization is applied after layer 1 and layer 2. A dropout layer is added to the fully connected layer with the dropout value of 0.5. This model uses Adam Optimizer with the base learning rate of 0.001.

The training loss and the model accuracy graphs are shown below:

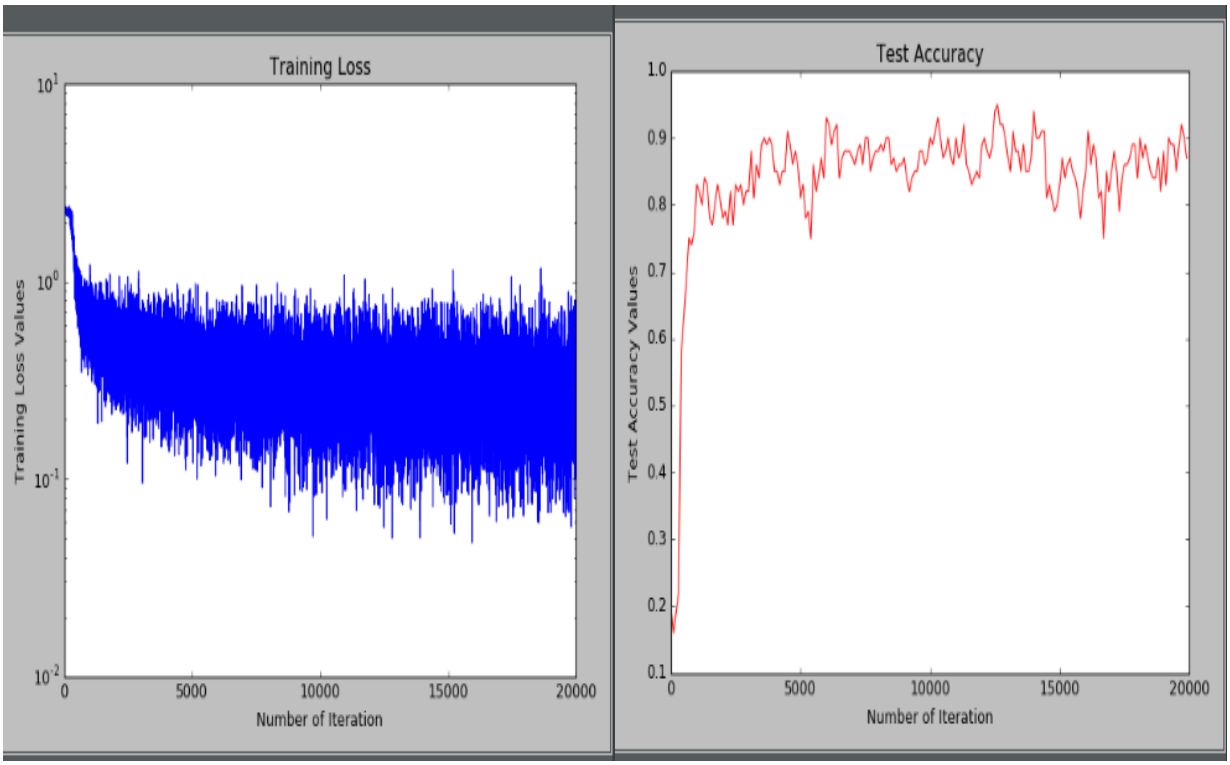


FIGURE 2 Training loss and Test accuracy

The model seems to be performing well. The model has achieved the accuracy of 91%.

TensorFlow

This section includes the description of the model built in TensorFlow to do the classification task. The code can be found the /code/TensorFlow folder in the repository.

INTRODUCTION / OBJECTIVE

The aim of this study is to perform a deep learning analysis on image data, measure, compare results and describe the findings. I am looking to train a model that would Classify as accurately as possible each of the digit images available.

In this project, I am using TensorFlow Framework that I encountered in class to classify the Street View House Numbers (SVHN) dataset using Convolution Neural Network (CNN) architecture. I chose this dataset because it gave me 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.

DATA SET

Google Street View House Number(SVHN) Dataset

Souce: http://ufldl.stanford.edu/housenumbers/

SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It is similar in flavor to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000-digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images.

- Total 10 Classes, 1 for each digits i.e Label '9' for digit 9 and '10' for digit 0.
- 73,257 digits for training, 26,032 digits for testing
- Available in two different formats:
  Format 1: Original images with bounding box available for each character (may contain multiple characters in same images).
  Format 2: MNIST like 32x32 cropped images having single character in each image.

I worked on Format 2 dataset in this project using CNN Architecture on TensorFlow framework.

```
('Training Set', (73257, 32, 32, 3))
('Test Set', (26032, 32, 32, 3))

('Total Number of Images', 99289)
```

WHY TENSORFLOW

TensorFlow is an open source software library for numerical computation using dataflow graphs. Nodes in the graph represents mathematical operations, while graph edges represent multi-dimensional data arrays (aka tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

TensorFlow as nothing but numpy with a twist. A major difference between numpy and TensorFlow is that TensorFlow follows a lazy programming paradigm. It first builds a graph of all the operation to be done, and then when a "session" is called, it "runs" the graph. It's built to be scalable, by changing internal data representation to tensors (aka multi-dimensional arrays). Building a computational graph can be considered as the main ingredient of TensorFlow.

The advantages of using TensorFlow are:

- It has an intuitive construct, because as the name suggests it has "flow of tensors". You can easily visualize each part of the graph.
- Easily train on CPU/GPU for distributed computing
- Platform flexibility. You can run the models wherever you want, whether it is on mobile, server or PC.

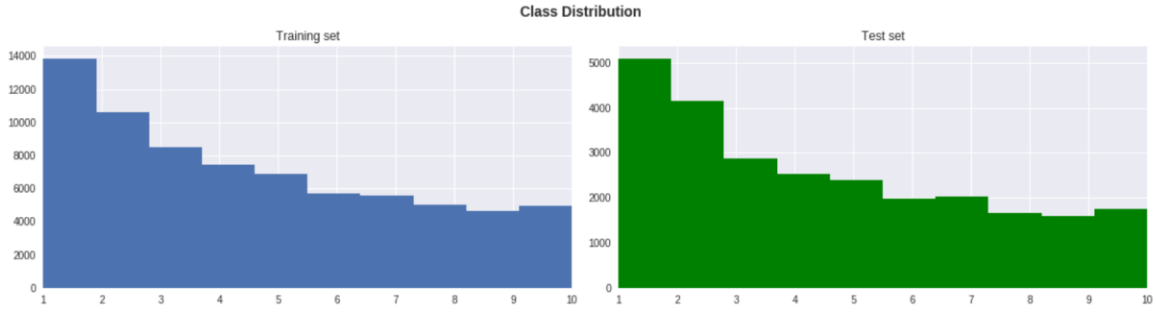
PREPROCESSING

Pre-processing the 32-by-32 images from the SVHN dataset centered around a single digit. In this dataset all digits have been resized to a fixed resolution of 32-by-32 pixels. The original character bounding boxes are extended in the appropriate dimension to become square windows, so that resizing them to 32-by-32 pixels does not introduce aspect ratio distortions.

1. Created a Balanced (Startified) 13% of data in Validation Set.
   Splitting to 13% in Val Set as it gives around 9500 data having minimum of 800 instances of each class.

```
('Training Set', (63733, 32, 32, 1))
('Validation Set', (9524, 32, 32, 1))
('Test Set', (26032, 32, 32, 1))
```

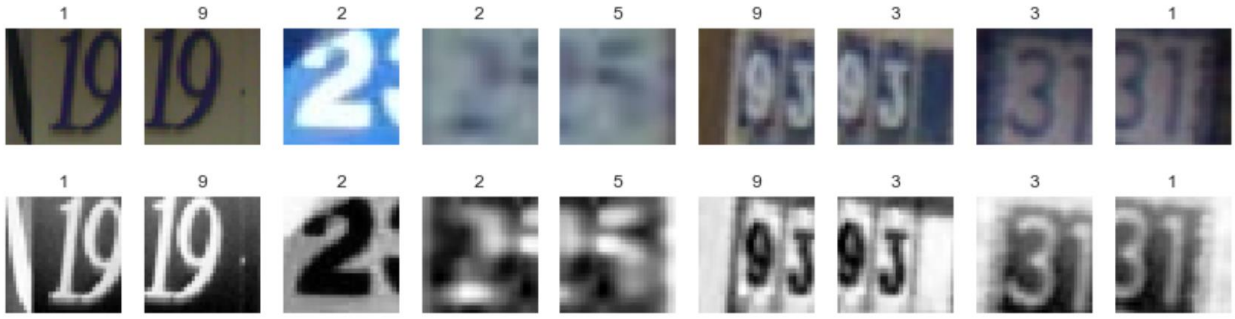
2. Initial Exploratory Analysis



All distributions have a positive skew, meaning that we have an underweight of higher digit values.

3. Converting the Label 10's to 0's
   So, we got a target label of [0 1 2 3 4 5 6 7 8 9]

4. RGB to Grayscale
   To speed up my experiments I convert the images from RGB to Grayscale, which reduced the amount of data to process. Since we are not concerned about the color of the image, just its shape (i.e. the number shown).
   Y = 0.2990R + 0.5870G + 0.1140B



Sample Images from my training set. RGB to Grayscale.

5. Normalization
   Normalization refers to normalizing the data dimensions so that they are of approximately the same scale. Divide each dimension by its standard deviation, once it has been zero-centered. Calculated the mean and standard deviation here.

6. One Hot Encoding
   Apply One Hot Encoding to make label suitable for CNN Classification.

```
('Training set', (63733, 10))
('Validation set', (9524, 10))
('Test set', (26032, 10))
```

7. Storing to Disk
   Used h5py package to store the numerical data, so that it can be easily manipulated using NumPy.

CODE WORKFLOW

The Work flow which I followed is broken down into 4 major phases

1. Making TensorFlow computational graphs (CNN Architecture)
2. Designing the TensorFlow Model
3. Execution of TensorFlow Model
4. Model Evaluation

Making TensorFlow computational graphs

I wrote functions for creating new TensorFlow Variables in the given shape and initializing them with random values. Followed by functions for stacking CONV-RELU layers and POOLING layers as well.

A convolutional layer produces an output tensor with 4 dimensions. We will add fully-connected layers after the convolution layers, so we need to reduce the 4-dim tensor to 2-dim which can be used as input to the fully-connected layer. So, I defined a flatten layer as well followed by the function to stack FC-RELU layer. Below are the functions that I created

```
def conv_weight_variable(layer_name, shape):
def fc_weight_variable(layer_name, shape):
def bias_variable(shape):
def conv_layer
def flatten_layer(layer):
def fc_layer
```

Designing the TensorFlow Model

First task was to define Placeholder Variables. They serve as the input to the graph that we may change each time we execute the graph. This allows us to change the images that are input to the TensorFlow graph.

```
x = tf.placeholder(tf.float32, shape=(None, img_size, img_size,
num_channels), name='x')
```

I implemented the following ConvNet architecture based on the layer patterns proposed in the CS231n notes:

```
INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL] -> DROPOUT ->(Flatten) [FC ->
RELU] -> FC -> Softmax
```

Applied Optimization methods, computed cross-entropy as well

Execution of TensorFlow Model

Once the TensorFlow graph has been created, I created a TensorFlow session which is used to execute the graph.

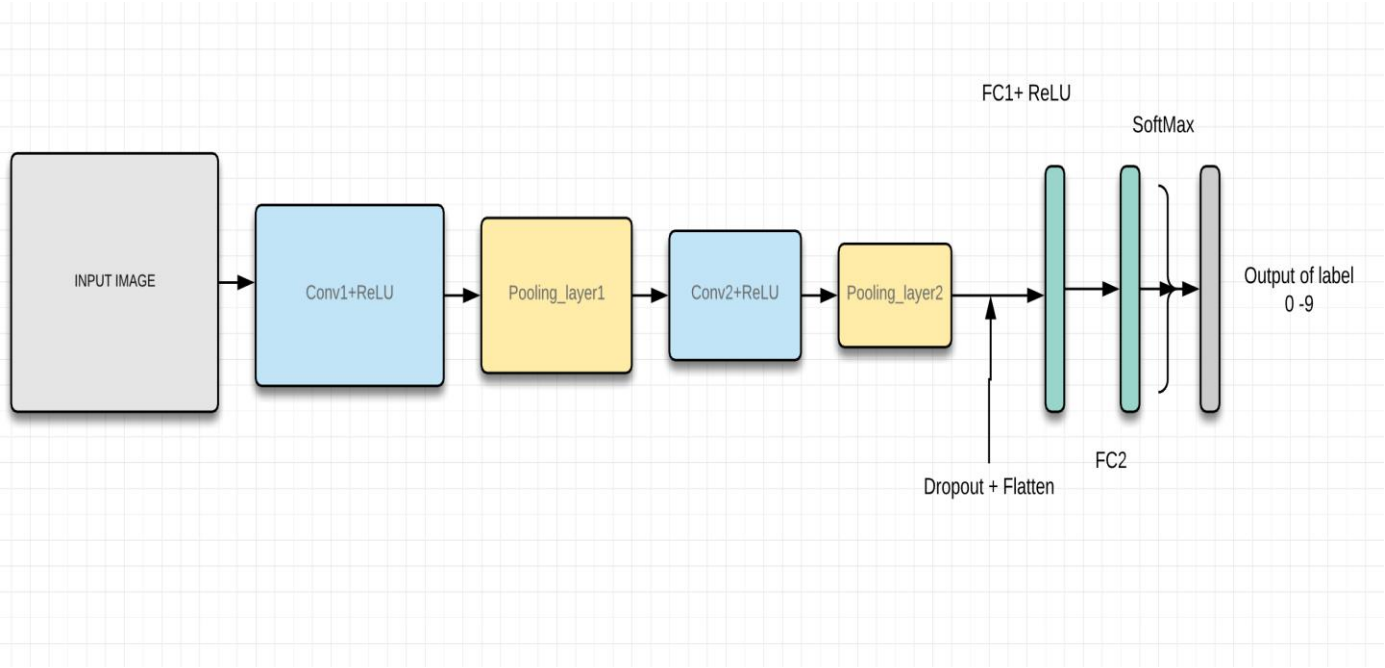
In order to save the variables of the neural network, I created a so-called Saver-object which is used for storing and retrieving all the variables of the TensorFlow graph. The saved files are often called checkpoints because they may be written at regular intervals during optimization. This is the directory used for saving and retrieving the data. After that, the execution starts on the training, validation set and testing set and we get the accuracy percentage for our model.

```
optimize(num_iterations=50000, display_step=1000)
```

Model Evaluation

Here I found and plotted some of the mis-classified examples in my testset and a confusion matrix showing how well our model is able to predict the different digits.

MODEL ARCHITECTURE



Convolutional Layer 1 & Pooling Layer 1

Create the first convolutional layer. It takes x as input and creates num_filters1 different filters, each having width and height equal to filter_size1.

Convolutional Layer 2 & Pooling Layer 2

Create the second convolutional layer, which takes as input the output from the first convolutional layer. The number of input channels corresponds to the number of filters in the first convolutional layer. Finally, we wish to down-sample the image so it is half the size by using 2x2 max-pooling.

Dropout Layer

To reduce overfitting, we will apply dropout after the pooling layer.

Flatten Layer

The convolutional layers output 4-dim tensors. I wanted to use these as input in a fully-connected network, which requires for the tensors to be reshaped or flattened to 2-dim tensors.

Fully-Connected Layer 1

Add a fully-connected layer to the network. The input is the flattened layer from the previous convolution. The number of neurons or nodes in the fully-connected layer is fc_size. ReLU is used so we can learn non-linear relations.

Fully-Connected Layer 2

Add another fully-connected layer that outputs vectors of length 10 for determining which of the 10 classes the input image belongs to. Note that ReLU is not used in this layer.

Softmax

The second fully-connected layer estimates how likely it is that the input image belongs to each of the 10 classes. However, these estimates are a bit rough and difficult to interpret because the numbers may be very small or large, so we want to normalize them so that each element is limited between zero and one and the 10 elements sum to one. This is calculated using the so-called softmax function and the result is stored in y_pred.

Calculate Cross-entropy

To make the model better at classifying the input images, we must somehow change the variables for all the network layers. To do this we first need to know how well the model currently performs by comparing the predicted output of the model y_pred to the desired output y_true.

Optimization Method: Adam

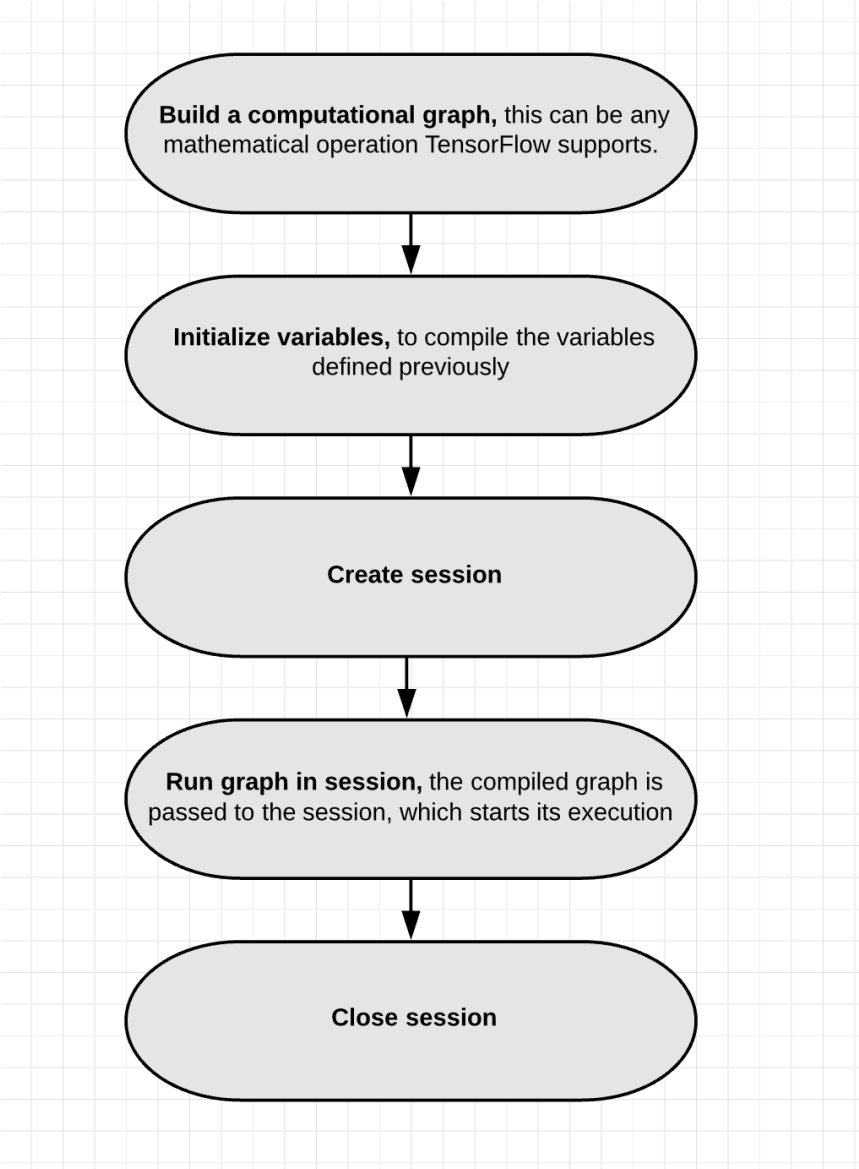
Now that we have a cost measure that must be minimized, we can then create an optimizer. In training deep networks, it is usually helpful to anneal the learning rate over time. There are three common types of implementing the learning rate decay. Here I used exponential decay and Adagrad optimizer.

The best architecture that gave me the best results:

Layer	**Description**
Input Layer	We input a batch of 64 images (64, 32, 32 ,1)
Conv1+ReLU1	32 Filters, 5x5 Kernal size, Stride 1, Zero Padding
Pooling1	2x2, Stride 2
Conv2+ReLU2	64 Filters, 5x5 Kernal size, Stride 1, Zero Padding
Pooling2	2x2, Stride 2
Dropout	0.5
FC	256 nodes

FRAMEWORK WORKFLOW

The following is the workflow which I followed while creating the CNN model on TenserFlow framework

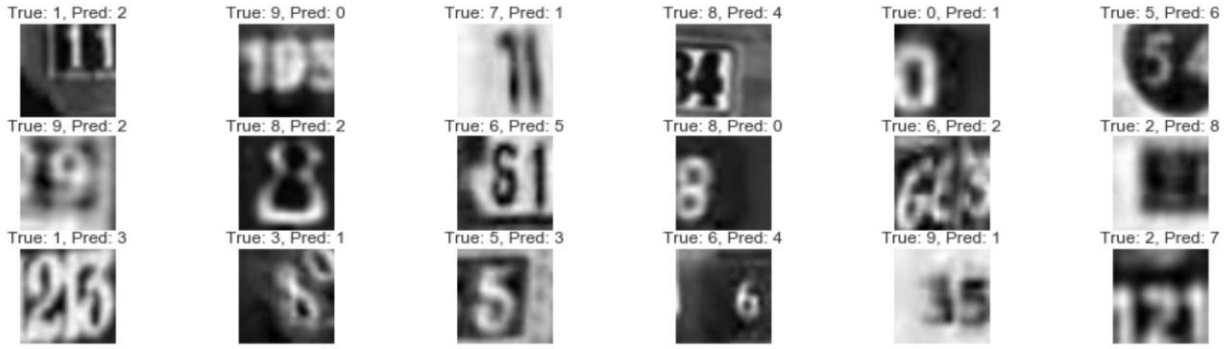


RESULTS & CONCLUSIONS

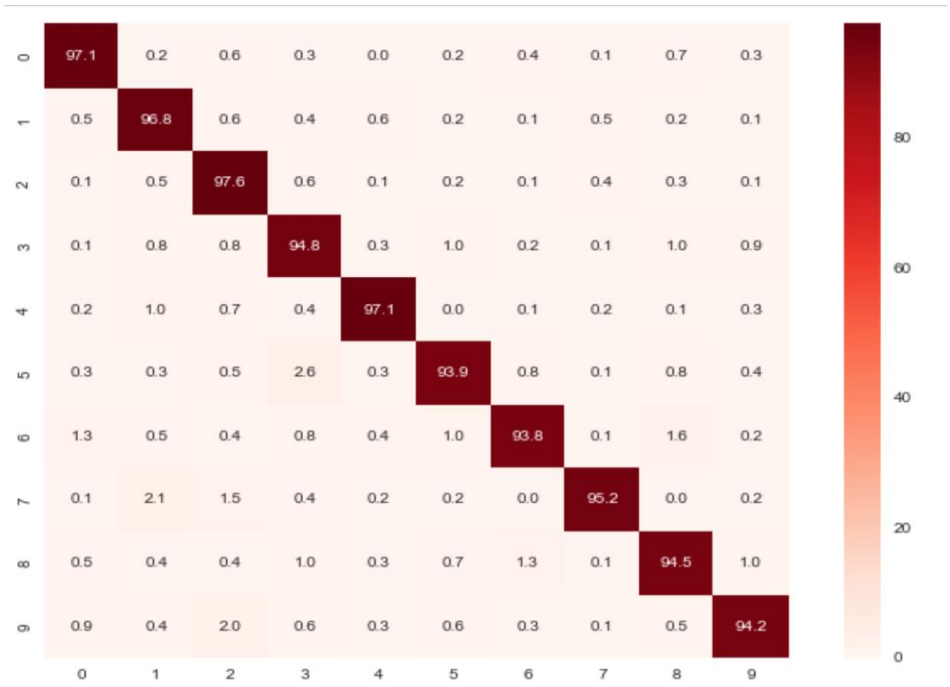
With the mentioned Architecture I got good Test accuracy of **92.96 %** at 50000 iterations which is good enough I think.

```
Minibatch accuracy at step 49000: 1.0000
Validation accuracy: 0.9022
Test accuracy: 0.9296
Time usage: 0:06:50
```

I had some of the misclassified images which I found so I dig in deeper to see how good the model works



So, I calculated the confusion matrix to show how well our model is able to predict the different class values. The Accuracy % of all digits seem high.



INTERNET CODE

I followed the cs231n notes Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition for the architecture to get the broad idea and suggestions. For TensorFlow Framework setup I referenced Stack overflow and the official website of TensorFlow

Percentage of code found/copied (by the defined formula): 30%

REFERENCES

[1] A.Karpathy, http://cs231n.github.io/convolutional-networks.

[2] Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks (https://arxiv.org/abs/1312.6082)

[3] https://www.tensorflow.org/

[3] Stack Overflow

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:

```
ubuntu@ip-172-31-21-233:~/code/Final-Project-Group-7/code/pytorch$ python3 train_predictor.py -h
usage: train_predictor.py [-h] [--batch BATCH] [--epochs EPOCHS] [--opt OPT]
                          [--net NET] [--lr LR] [--cpu] [--id ID]

Train SVHN predictor.

optional arguments:
  -h, --help       show this help message and exit
  --batch BATCH    Batch Size
  --epochs EPOCHS  Epochs
  --opt OPT        Optimizer (SGD, Adagrad, Adadelta, Adam, ASGD)
  --net NET        Network architecture (ConvNet32, ConvNet48, Convnet32_753,
                   ConvNet48_333, ConvNet48_Dropout, ConvNet48_Dropout2,
                   ConvNet48_Dropout3)
  --lr LR          Learning rate
  --cpu            Force to CPU even if GPU present
  --id ID          Optional ID to prepend to results files.
ubuntu@ip-172-31-21-233:~/code/Final-Project-Group-7/code/pytorch$
```

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:

```
class ConvNet48_Dropout3(BaseNet):
    def __init__(self, num_classes, channels, image_size):
        super(ConvNet48_Dropout3, self).__init__(num_classes, channels, image_size)
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 48, kernel_size=3, padding=1),
            nn.BatchNorm2d(48),
            nn.ReLU(),
            nn.MaxPool2d(2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(48, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2))
        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2))

        self.calculate_conv_layer_output_size((self.layer1, self.layer2, self.layer3))
        self.drop1 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(self.num_conv_outputs, self.num_classes)

    def forward(self, x):
        in_size = x.size(0)
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = out.view(in_size, -1)
        out = self.drop1(out)
        out = self.fc1(out)

        return out
```

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 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.

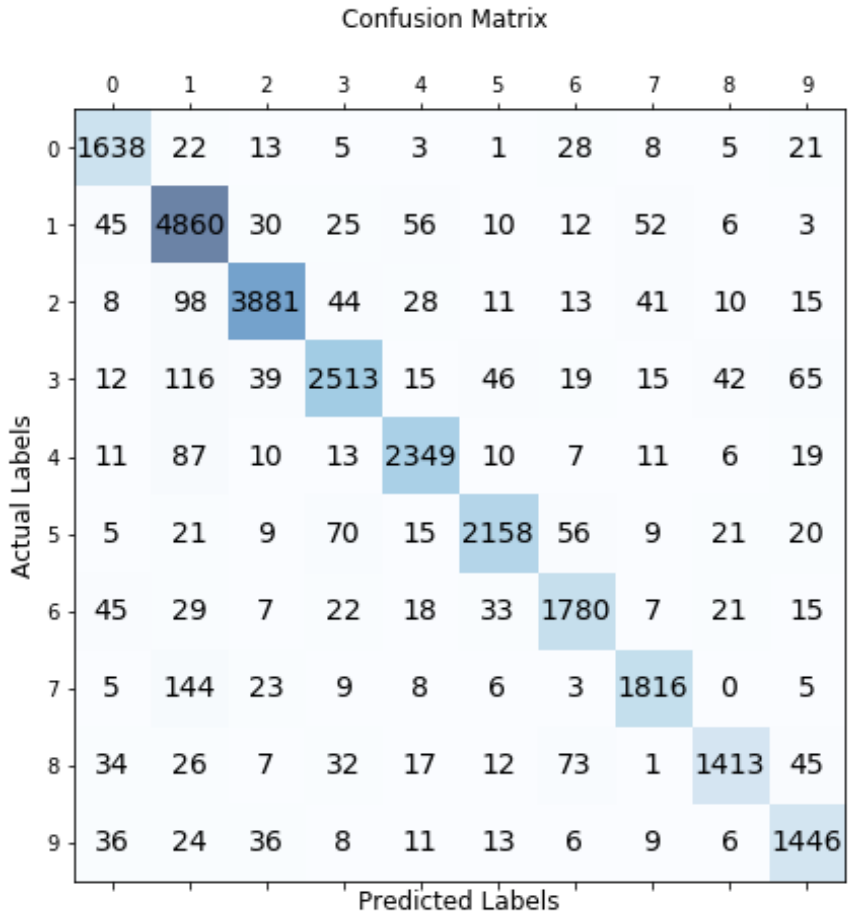
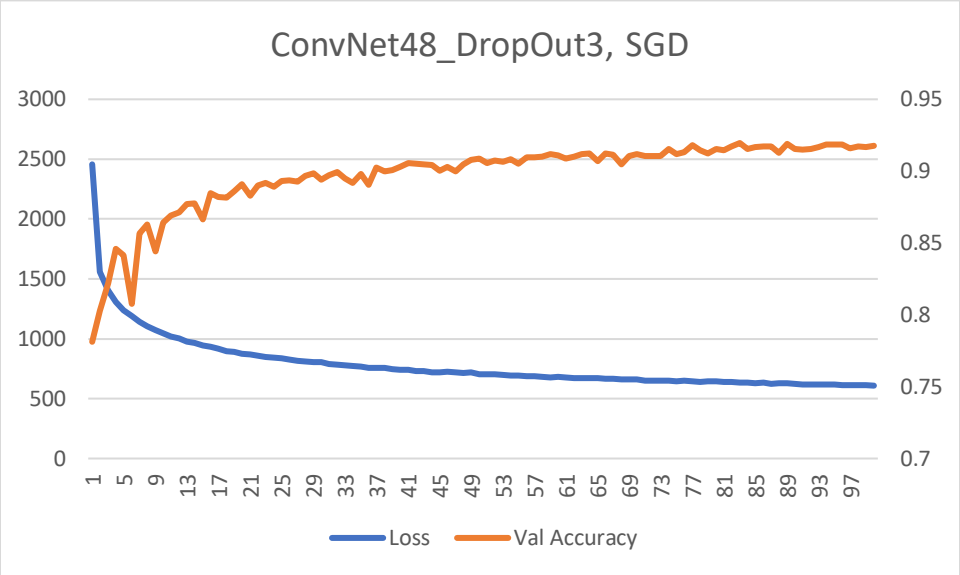


FIGURE 5 PYTORCH BEST MODEL CONFUSION MATRIX

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,

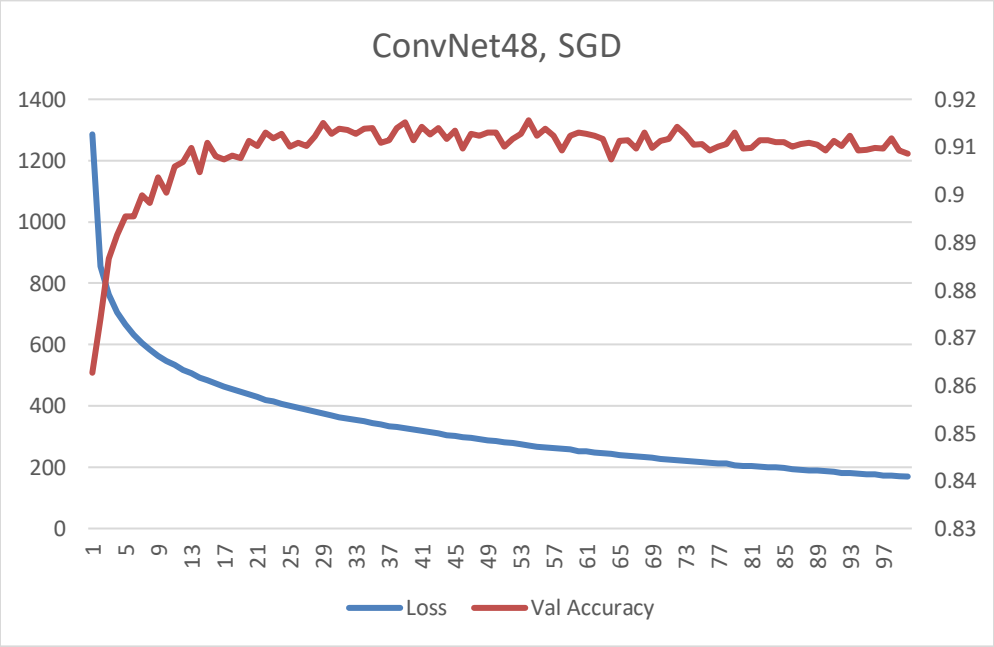
however, the confusion matrix shows strong results, with the diagonal column of prediction-matching-actual being the dominant result.

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.

Sample Training Run Results

From the "winning" network with dropout:

```
ConvNet48_Dropout3(
  (layer1): Sequential(
    (0): Conv2d(1, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(48, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer3): Sequential(
    (0): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (drop1): Dropout(p=0.5)
  (fc1): Linear(in_features=800, out_features=10, bias=True)
)

Image size: (40, 40)
Total network weights + biases: 55245
Epochs: 100
Learning rate: 0.005
Batch Size: 32
Final loss: 0.1202
Run device: cuda
Num Conv outputs: 800
Loss function: CrossEntropyLoss()
Optimizer:
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.005
    momentum: 0
    nesterov: False
    weight_decay: 0
)
Elapsed time: 2085

Accuracy of the network on the 26032 test images: 91.6%

Accuracy of 0   : 93.9%
Accuracy of 1   : 95.3%
Accuracy of 2   : 93.6%
Accuracy of 3   : 87.2%
Accuracy of 4   : 93.1%
Accuracy of 5   : 90.5%
Accuracy of 6   : 90.0%
Accuracy of 7   : 89.9%
Accuracy of 8   : 85.1%
Accuracy of 9   : 90.7%
```

From the similar network without dropout:

```
ConvNet48(
  (layer1): Sequential(
    (0): Conv2d(1, 48, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(48, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer3): Sequential(
    (0): Conv2d(64, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc): Linear(in_features=800, out_features=10, bias=True)
)

Image size: (40, 40)
Total network weights + biases: 137933
Epochs: 100
Learning rate: 0.005
Batch Size: 32
Final loss: 0.0179
Run device: cuda
Num Conv outputs: 800
Loss function: CrossEntropyLoss()
Optimizer:
Adagrad (
Parameter Group 0
    initial_accumulator_value: 0
    lr: 0.005
    lr_decay: 0
    weight_decay: 0
)
Elapsed time: 2667

Accuracy of the network on the 26032 test images: 91.0%

Accuracy of 0   : 92.0%
Accuracy of 1   : 93.9%
Accuracy of 2   : 92.6%
Accuracy of 3   : 87.1%
Accuracy of 4   : 92.0%
Accuracy of 5   : 90.3%
Accuracy of 6   : 89.4%
Accuracy of 7   : 89.4%
Accuracy of 8   : 86.4%
Accuracy of 9   : 91.5%
```

The network with dropout ran faster and delivered a higher accuracy, all other things being equal.

Digit Removal from Images

We were inspired by the topics in class related to deriving information from the by-products of a network, such as the gradients and the feature maps. This task was to experiment with the by products to see if it was possible to use them to alter a test set parent image to remove the digit, ideally in a photo-realistic manner.

Shperber (Shperber, 2017) discusses using supervised deep learning techniques to determine for an image pixel-by-pixel whether the pixel is in the image foreground or background. This approach requires labeled training data which was not available to us with the SVHN data. We want to know if an unsupervised approach could give acceptable results in the limited case of images of digits. In our case, instead of removing the background as Shperber proposes, we would remove the foreground object (the digit) and attempt to synthesize an acceptable background.

Approach

- Train a supervised neural network to classify digits in SVHN data.
- Use the by-products of the network to identify key pixels in a digit image that are not part of the digit itself.
- Use a Generalized Neural Network employing a radial basis function to take the key background pixels and synthesize a new image, omitting the digit, and patch this image into the parent image, in place of the original

The GRNN was based on ideas from Alilou and Yaghmaee (Alilou & Yaghmaee, 2015), who kindly provided their paper. In their work, they removed text and other obscuring items that had been overlaid on a detailed image. They knew the precise pixel locations of the areas they were infilling. In our case, we only had key pixels from the original that we believed were part of the background. Consequently, we did not follow their procedure exactly, but they did greatly influence the resulting approach.

Identifying the Key Pixels

Gradients

The initial idea was find pixels in a digit image likely not to be in the digit by feeding the image through a trained prediction network and using the gradients of the input with respect to the output associated with the predicted output class. We hoped that the pixels that were not key to the decision would be background pixels. Unfortunately, this was not fruitful. There are clear examples (see figure) where this appears to be promising, but the gradients were still very noisy

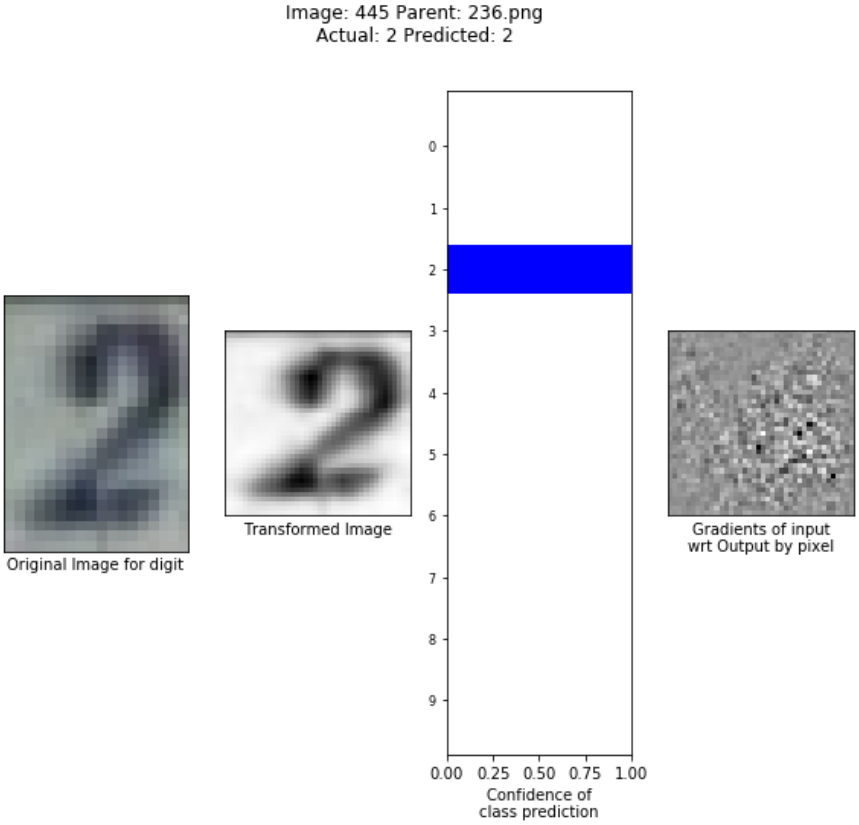


Image: 445 Parent: 236.png
Actual: 2 Predicted: 2
Original Image for digit
Transformed Image
Gradients of input
wrt Output by pixel
0
1
2
3
4
5
6
7
8
9
0.00 0.25 0.50 0.75 1.00
Confidence of
class prediction

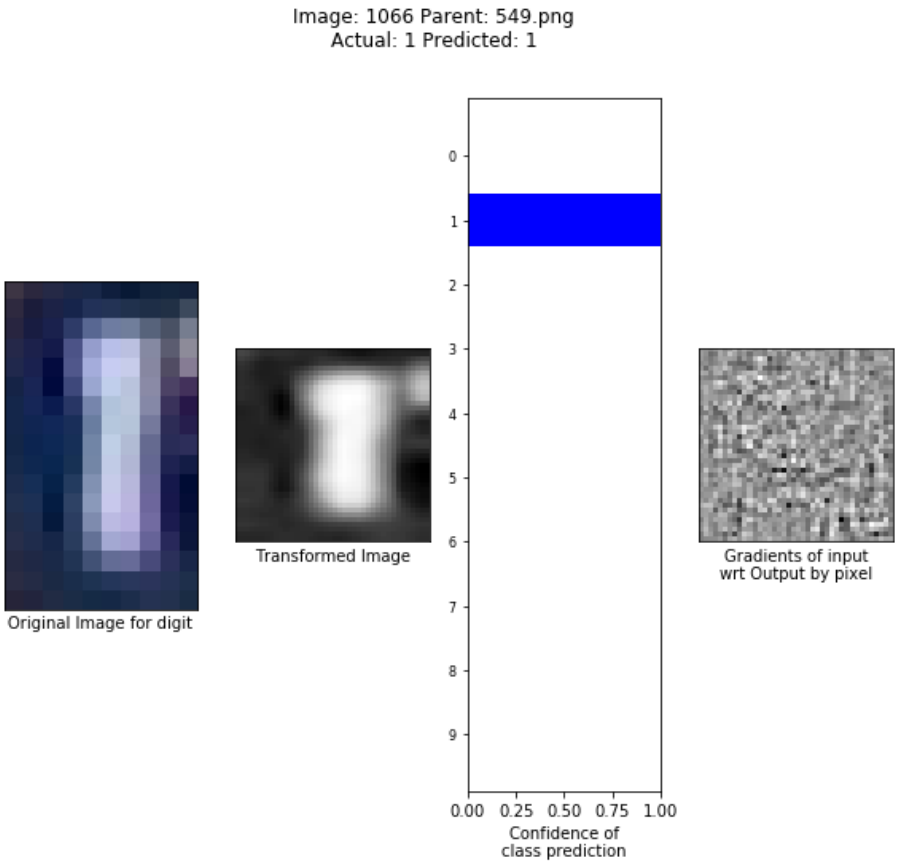


FIGURE 6 EXAMPLE DIGIT PREDICTION, WITH GRADIENTS OF INPUT WRT OUTPUT

In the first example, the gradient plot (dark points are smaller vales, light is larger) shows a pattern matching where the digit is in the image. Visually there are still highs and lows in the background region but more points in the middle (gray). The region corresponding to the digit has more pronounced swings but still has a variety of values. A supervised technique could likely learn to tell the two regions apart but with no training data to support this, we looked for another approach.

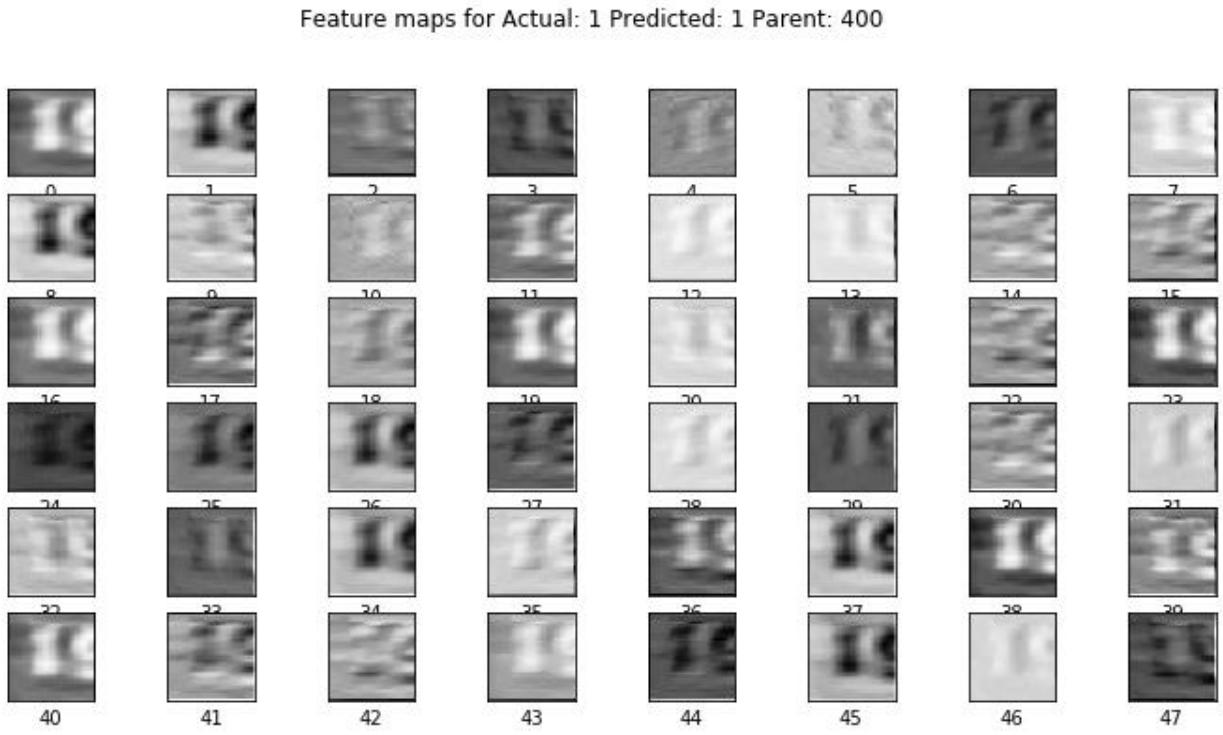
Feature Maps

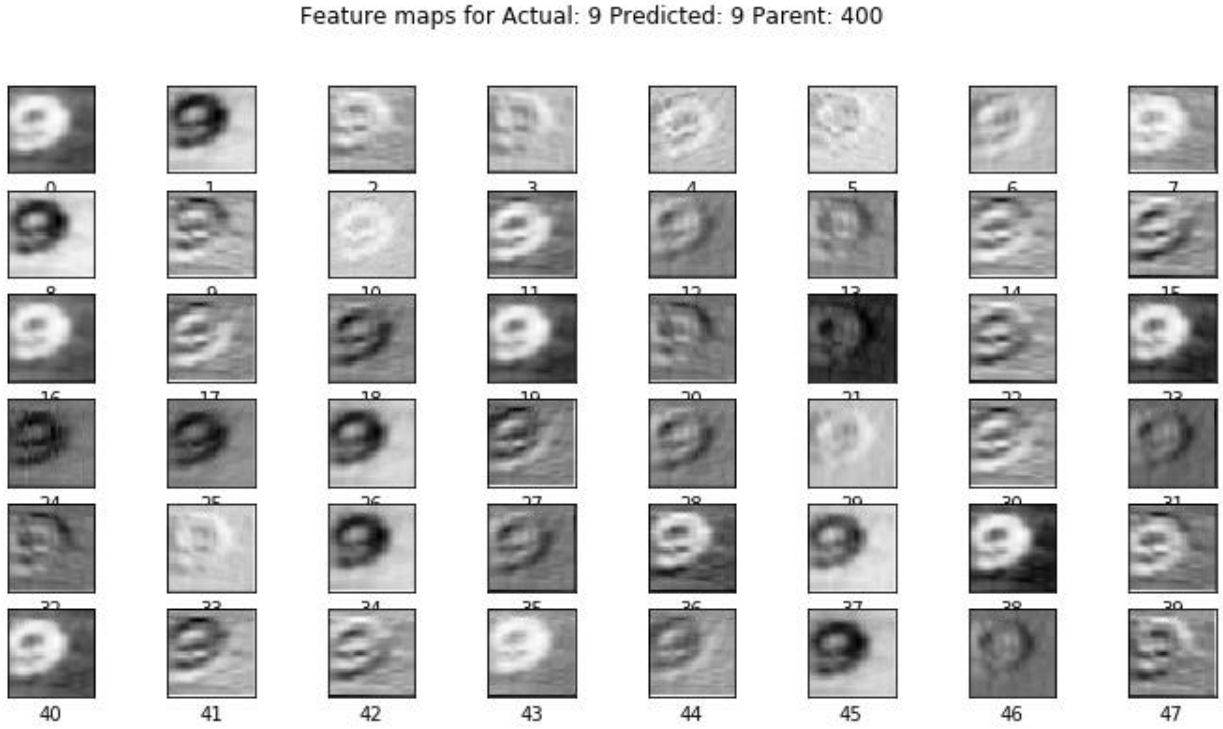
The other by-product we discussed in class was feature maps. We found that running an image through a prediction network and analyzing the feature maps from the first convolutional layer, we could be more successful in identifying background pixels.

Consider parent image 400 (401.png):

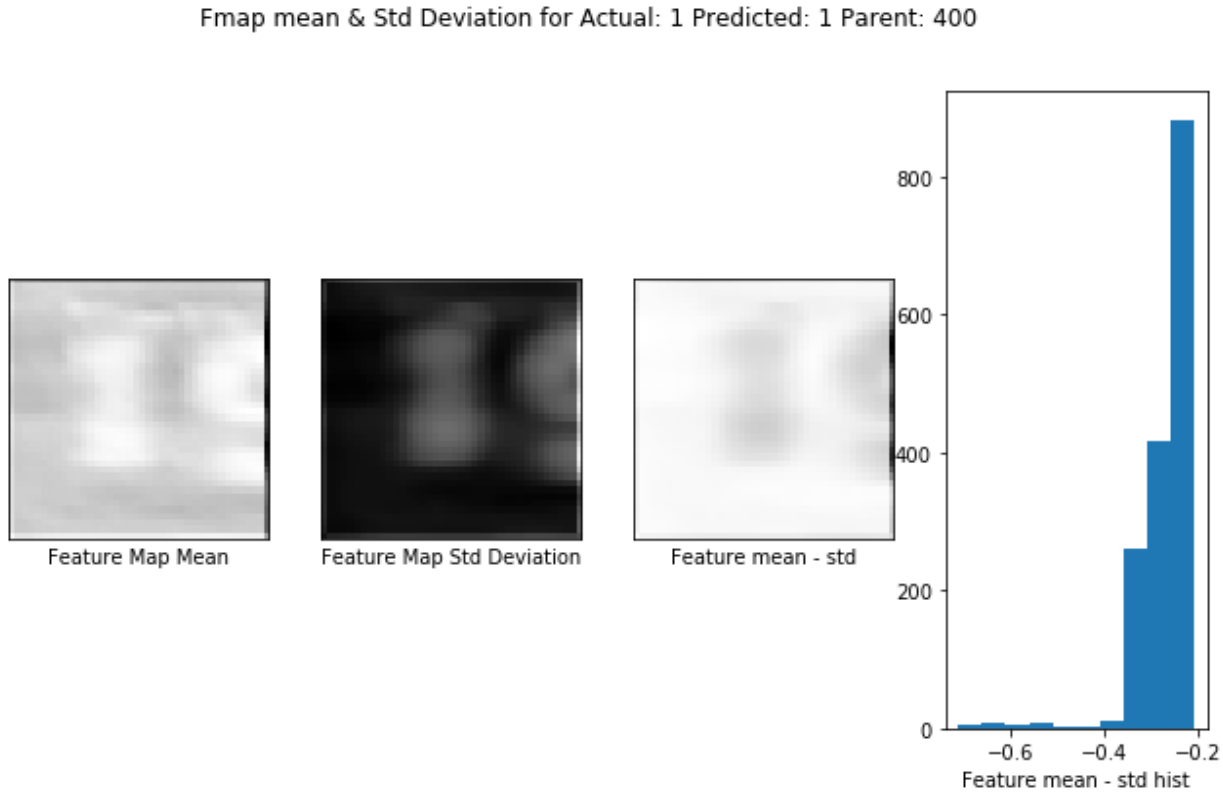
FIGURE 7 PARENT IMAGE 400

This is the parent for two digit images. Extracting the feature maps:





And then trying different reductions of the feature maps to one set of data with the same dimensions as the image:



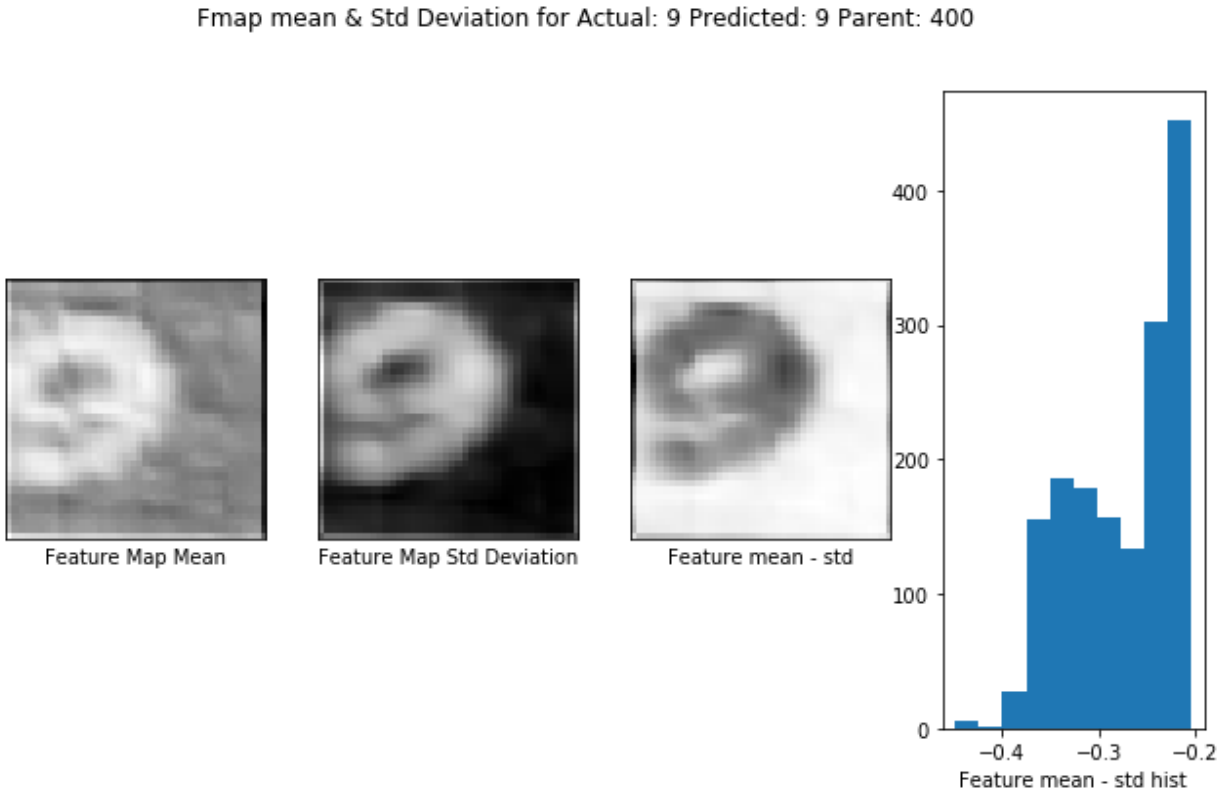


FIGURE 8 REDUCED FEATURE MAPS

The last two figures show the 48 feature maps for each digit reduced using various methods: mean, standard deviation, and mean – standard deviation. The histogram shows the distribution of values for the mean – standard deviation. By subtracting these two, the result tended to be smoother than either one individually. The technique to pick the key pixels likely to be in the background was to select pixels from the histogram bucket that had the most members. (A tunable parameter is how many buckets to create; we are using 35 right now but this needs further study). For selecting key pixels we use a dilation of 3 so that only every third pixel in each direction in the image is eligible to be used to generate a new image. Corner pixels are automatically included in order to help fit to the existing parent image.

This results in key pixels being selected as so:

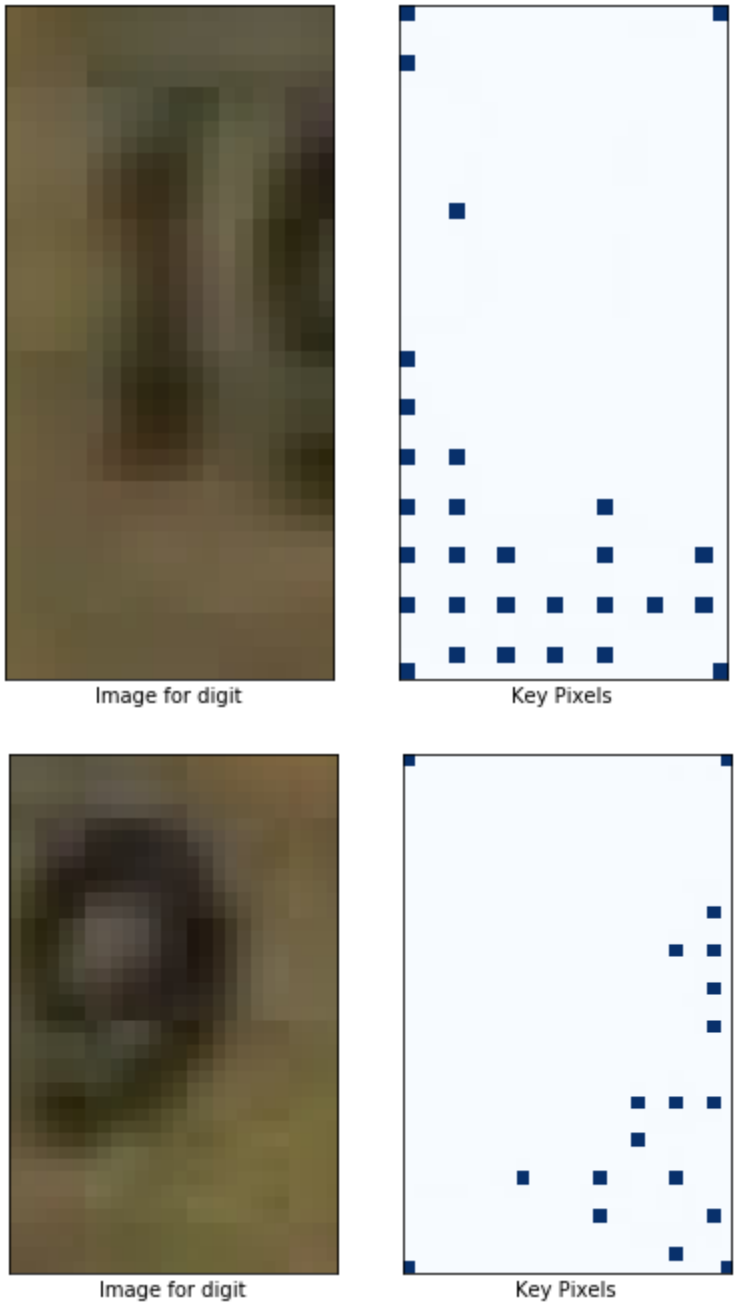


FIGURE 9 KEY PIXEL SELECTION

Generating a new image

This is where the work of Alilou and Yaghmaee was relied on most heavily. A network using a Radial basis Function is created and trained for each digit image. The training data for the network are the key pixels identified above. The network then predicts the value for each pixel in the image, resulting in an output image. Because the network never learned about the pixels that make up a digit, the digit is not present in the generated image.

They used a network with this architecture (excerpted from their paper):

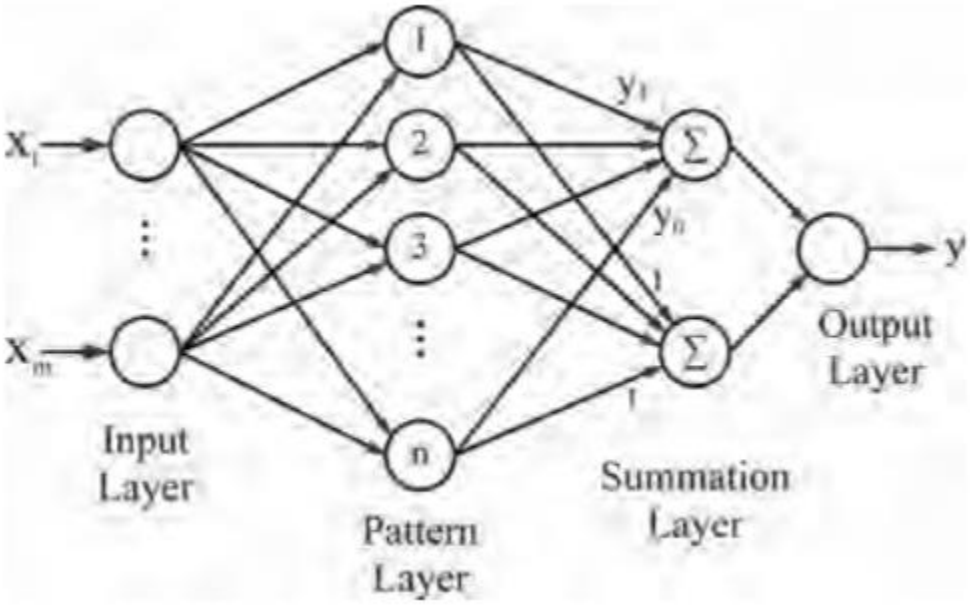


FIGURE 10 REFERENCE GRNN ARCHITECTURE

In this network, the input are pixel co-ordinates and the output is the pixel value for those co-ordinates. During training, the target values are the actual pixel values for the key pixels. (For color images such as the ones we are using this requires three separate networks). There is one node in the pattern layer for each training pixel.

To be more consistent with our notation, we would modify this diagram to show one input (instead of 2) as a length 2 vector for the pixel co-ordinates.

Each node in the pattern layer calculates the squared Euclidean distance between their training pixel and the input co-ordinate pair. This distance is passed to the summation layer where each Euclidean distance squared is run through an RBF. The top summation node includes a learnable weight parameter and the bottom node does not. Then in the output layer, the output from the top node is divided by the bottom, resulting in the implementation of the following equation:

$$y = \hat{f}(X) = \frac{\sum_{i=1}^{n} y_i \cdot \exp\left(\frac{-D_i^2}{2\sigma^2}\right)}{\sum_{i=1}^{n} \exp\left(\frac{-D_i^2}{2\sigma^2}\right)}$$

FIGURE 11 RBF INPUT AND WEIGHT FUNCTION

(again from (Alilou & Yaghmaee, 2015)). Note the presence of the sigma term, which in RBF is sometimes called spread. (Our book calls it bias). (Hagan, Demuth, Beale, & De Jesus)This equation results in a predicted value for the pixel at the input co-ordinates. This model assumes that pixel values are on a scale of 0 to 1.

The nature of radial basis function networks is that training samples for nodes close to the input have a greater effect on the output than more distant nodes. For image infill, this makes sense – a pixel is more likely to be similar to nearby pixels than distant ones. In our nomenclature, calculating the distances from each node to the input would be a non-standard input function, and doing the calculation would be a weight function.

Another borrow from Alilou & Yaghmaee is that they calculate the smallest missing part of their images first and then fold the predicted points into the training data and retrain the model before predicting the next region. Because we do not have defined regions of present and missing pixels in the image, we modify this to an iterative approach where first the candidate pixels in the dilation nearset to a training pixel are predicted, and then these are folded into the training data and the model retrained, and so forth until all pixels in the dilation have values, either from the training data or from predictions. Then the entire image is predicted.

The sigma parameter determines the extent to which the influence of a pixel diminishes as the distance from it increases. In the case we fix it to a smallish value. The minimum is 1 and we set it to a value near1.

Some of PyTorch implementation:

```
def rbf(self, W2, Dsquared):
    return W2 * (torch.exp(-1.0 * Dsquared / (2.0 * self.sigmaSq)))

def forward(self, X):
    self.X = X
    out = torch.zeros(len(X), dtype=torch.float, device=self.device)

    self.Dsquared = torch.zeros((len(X), len(self.pattern_layer)),
dtype=torch.float).to(device=self.device)

    for idx in range(len(X)):

        # Get sqaured distances to all pattern layer points from this X by
        # getting them from the precomputed values
        self.Dsquared[idx] = self.Dsquares[tuple(X[idx].cpu().numpy())]

    self.channel_out = []

    for cidx in range(self.channels):
        self.channel_out.append(self.rbf(self.W2[cidx], self.Dsquared).sum(dim=1) /
self.rbf(1, self.Dsquared).sum(dim=1))

    # Repackage the channels so that all channels for a pixel are on the same row
    out = torch.cat(self.channel_out, dim=0).view(self.channels, -1).transpose(0,1)

    return out
```

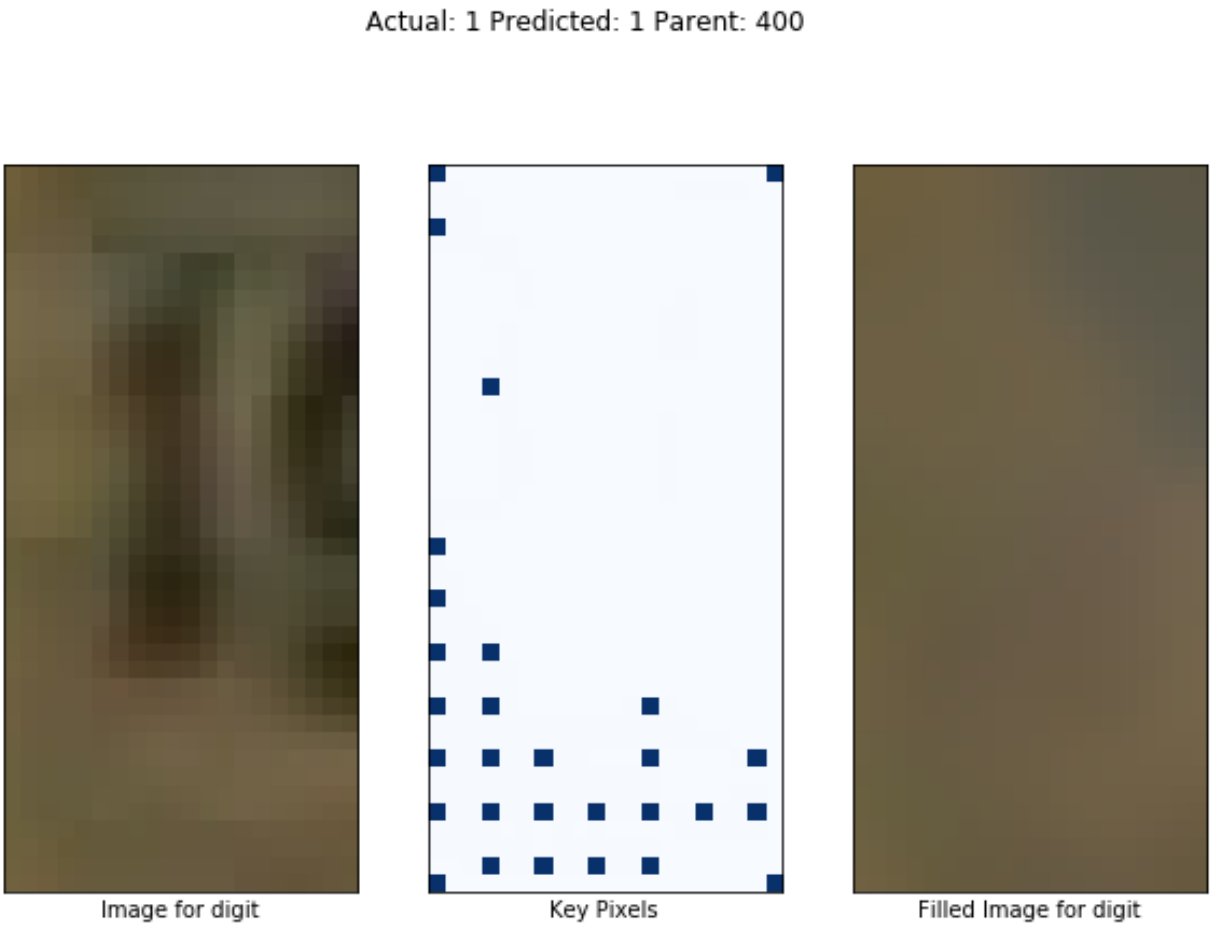
FIGURE 12 FORWARD FUNCTION OF IMAGE INFILL NETWORK

This show the forward() function for the infill network. The input X is one or more length 2 vectors with pixel co-ordinates. The function first constructs the squared distances between each X and the pattern layer coordinates, and then sends the distances to the rbf – one time with the weights for the numerator and one time with constant weight 1 for the denominator of the output calculation. The weights are PyTorch parameters and are trained in the usual fashion, using the MSE loss function and the Adagrad optimizer. Because the number of training points

matches the number of weights being calculated, there should be an exact solution. We found that Adagrad got to this exact solution much quicker than SGD.

Results

The result of generating images for the examples show above:



Actual: 9 Predicted: 9 Parent: 400

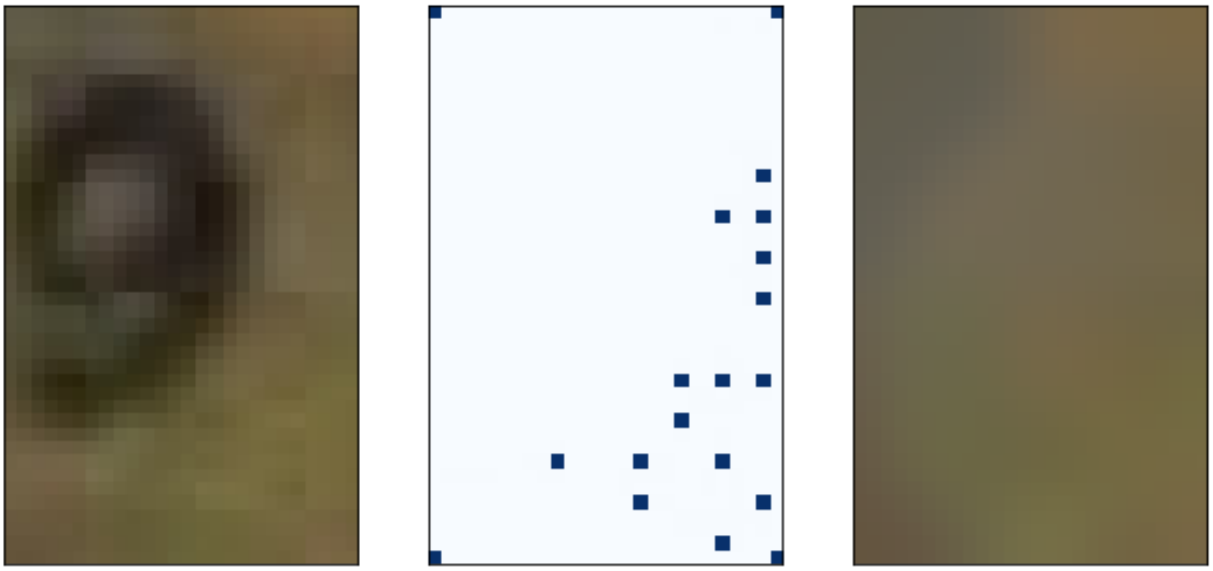


Image for digit | Key Pixels | Filled Image for digit

And then replacing the digit with the new image in the parent:

Parent Images before and digit replacement (Image 400)

Original | Altered

Some hits and misses:

Image 7607:

Actual: 2 Predicted: 2 Parent: 7607

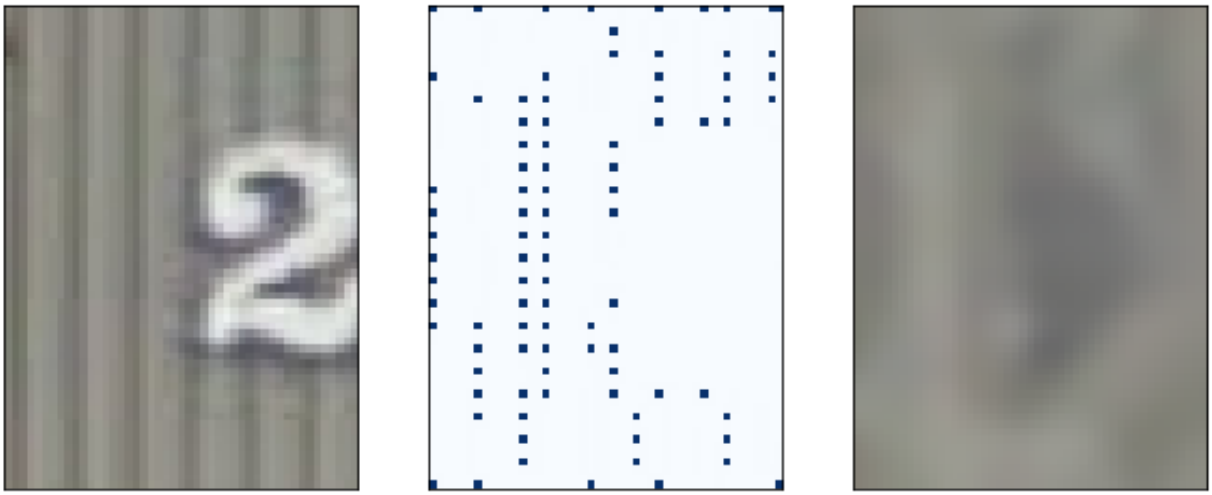


Image for digit | Key Pixels | Filled Image for digit

Actual: 2 Predicted: 9 Parent: 7607

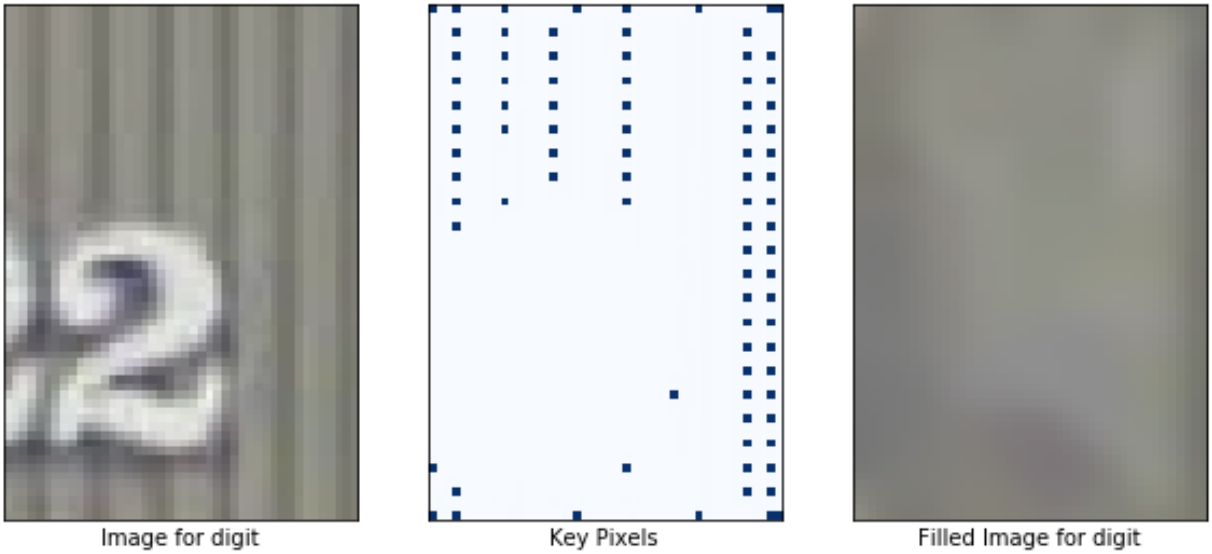


Image for digit | Key Pixels | Filled Image for digit

Parent Images before and digit replacement (Image 7607)

Original Altered

Image 1201:

Actual: 5 Predicted: 5 Parent: 1201

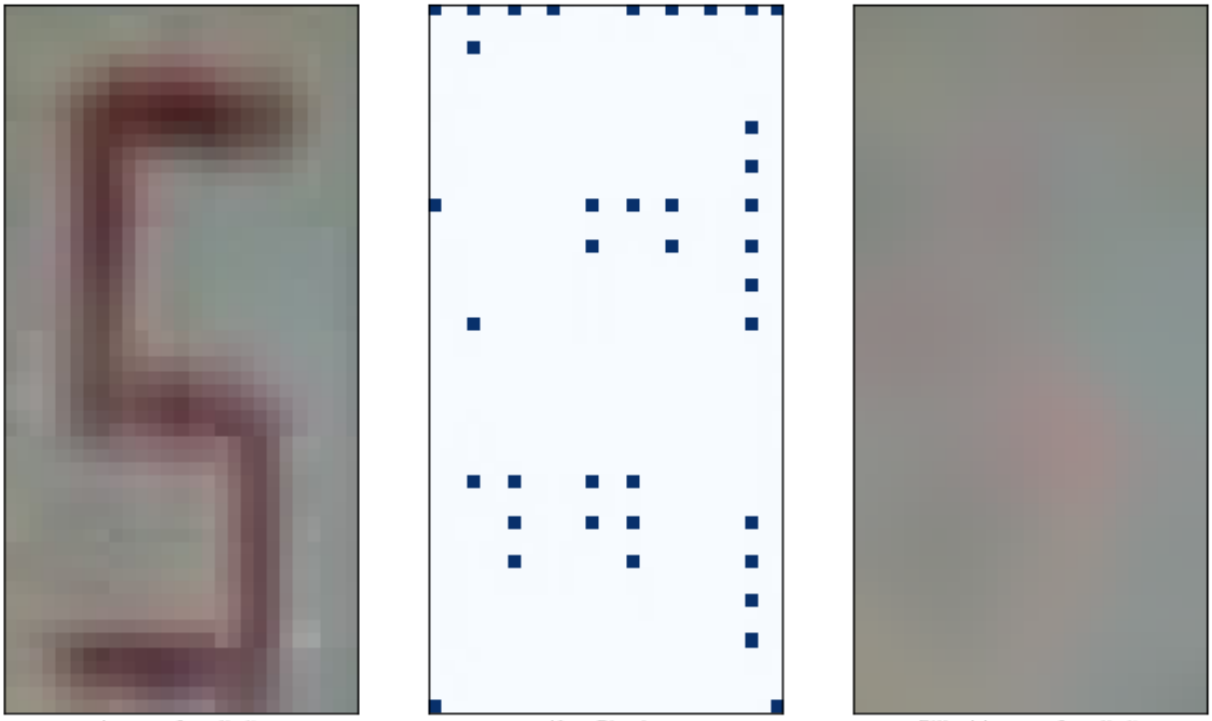


Image for digit Key Pixels Filled Image for digit

Actual: 4 Predicted: 4 Parent: 1201

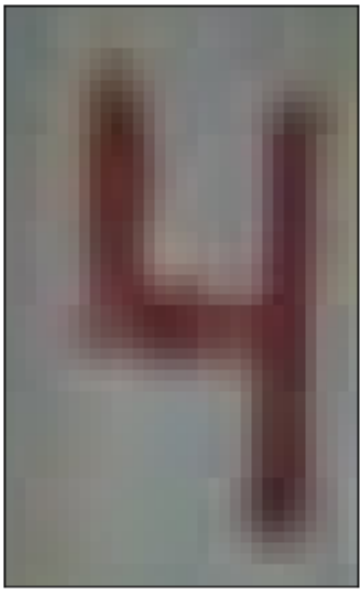
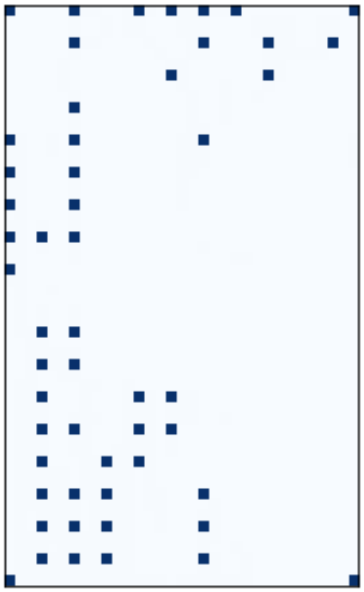
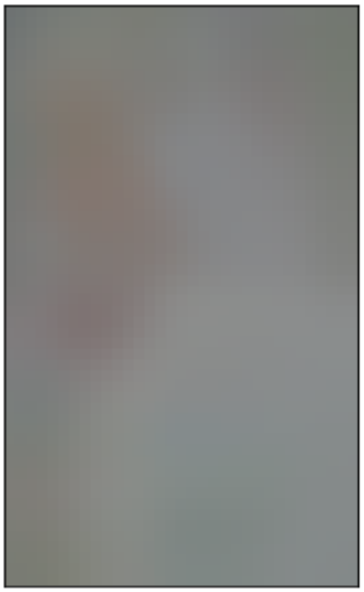


Image for digit



Key Pixels



Filled Image for digit

Parent Images before and digit replacement (Image 1201)

Original

Altered

Others:

Parent Images before and digit replacement (Image 12)

Original

Altered

Parent Images before and digit replacement (Image 29)

Original

Altered

Parent Images before and digit replacement (Image 8775)

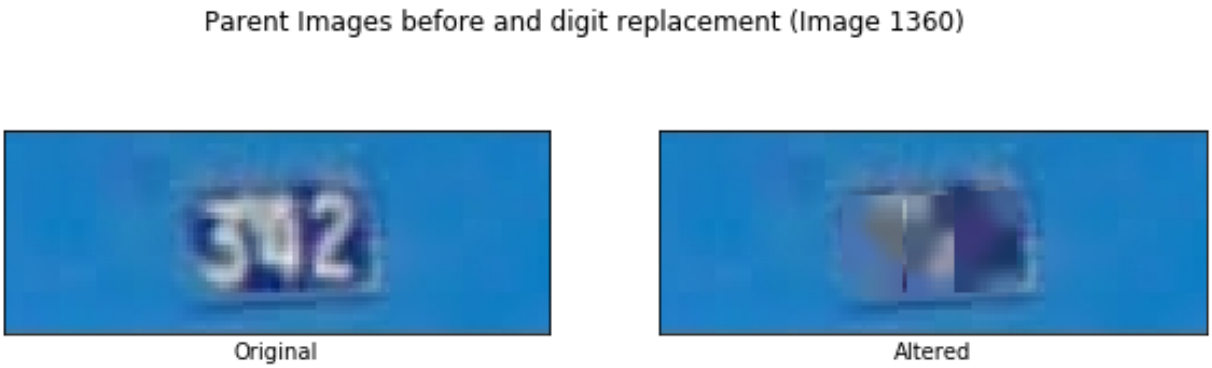
Original

Altered

Parent Images before and digit replacement (Image 555)

Original

Altered



Implementation Notes

The code for this is also in the /code/pytorch folder in the project repo.

Source for the infill task:

- **infiller.py --** User interface to the infill task. When run, it prompts the user for a parent image number, runs the digit removal process, and displays the results using matplotlib.
- **fillnet.py –**PyTorch module implementation of the GRNN network that predicts the image with the digit removed. (Not runnable on its own)
- **fillnet_trainer.py –** Helper class used to train the fill network.(Not runnable on its own)
- **key_pixels.py –** Helper class used to identify key pixels in the image (Not runnable on its own).

Conclusion

- There are some successes and some misses too. The algorithm does best when there is a border of background around the digit. It also does best with larger digit images. These both affect the key pixel identification.
- The image generation algorithm, once key pixels are identified, works well.
- There may be some loss of pixel fidelity as the images are being scaled and rescaled. By reducing the need to resize the images, this may result in a better outcome.

Summary and Conclusions

To summarize our findings, we can say that the model in Pytorch gave us the best accuracy which is 91.6 % and the caffe was the fastest in terms of performance. The accuracy of TensorFlow was the second best.

References

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

Hagan, M. T., Demuth, H. B., Beale, M. H., & De Jesus, O. (n.d.). *Neural Network Design, 2nd Edition.*

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

Shperber, G. (2017, August 27). *Background removal with deep learning*. Retrieved December 1, 2018, from Towards Data Science: https://towardsdatascience.com/background-removal-with-deep-learning-c4f2104b3157