Unsupervised Image Object Removal in the Street View House Numbers Dataset

Machine Learning II Final Project

Bill Grieser

The Project

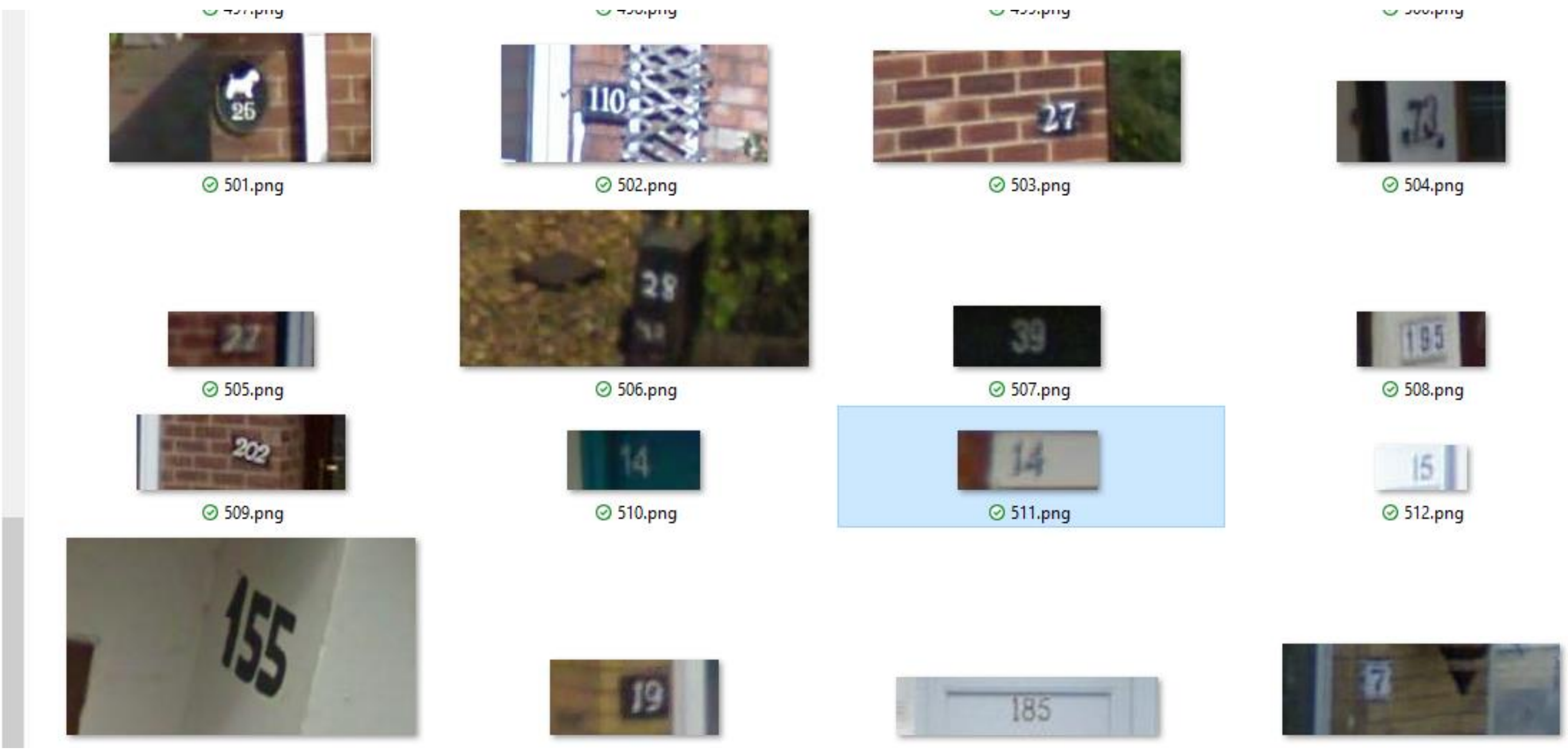
Using the Street View House Numbers dataset,

- Classify the digits in the images
- Use the by-products of the image classification network to remove the digits from the test images

The Data

Images culled from Google Street View that contain digits (with digit locations already identified in metadata):

Data Includes:
- Parent images
- Metadata for bounding box around each digit
- Label for each digit



Reference: 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

Unsupervised Digit Removal from Images

Not supervised: No labeled training data

Approach:

- Train a model in PyTorch to classify digit images
- Use by-products of the model to identify key pixels for an image
  - Key Pixel: represents the background without the digit
- Create a Generalized Regression NN, trained with the key pixels, to generate an image

Goal is when the generated image is returned to the parent, a photorealistic image without the digit results

The PyTorch Model

- Preprocessing:
  - Resize to 40 x 40
  - Grayscale
- Optimizer: SGD
- Batch Size: 32
- Learning rate: 0.005
- Architecture:
  - 3 Convolution-BatchNorm-Relu-Maxpool, layers, Dropout, and Fully Connected Layer
  - Number of kernels: 48 – 64 – 32
  - Kernel size 5x5
  - Dropout p=0.50

```python
class ConvNet48_Dropout5(BaseNet):
    def __init__(self, num_classes, channels, image_size):
        super(ConvNet48_Dropout5, self).__init__(num_classes, channels, image_size)
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 48, kernel_size=5, padding=2),
            nn.BatchNorm2d(48),
            nn.ReLU(),
            nn.MaxPool2d(2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(48, 64, kernel_size=5, padding=2),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2))
        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 32, kernel_size=5, padding=2),
            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
```

Overfitting Countermeasures

- Used dropout at 50%
- Monitored performance metrics over epochs to look for tell-tale loss decreasing while validation accuracy increasing

Batch Size

- Used a small batch size of 32 to get frequent weight updates
- Performed as well as 16
- Larger batch sizes ran faster, did not perform as well

Experimental Approach

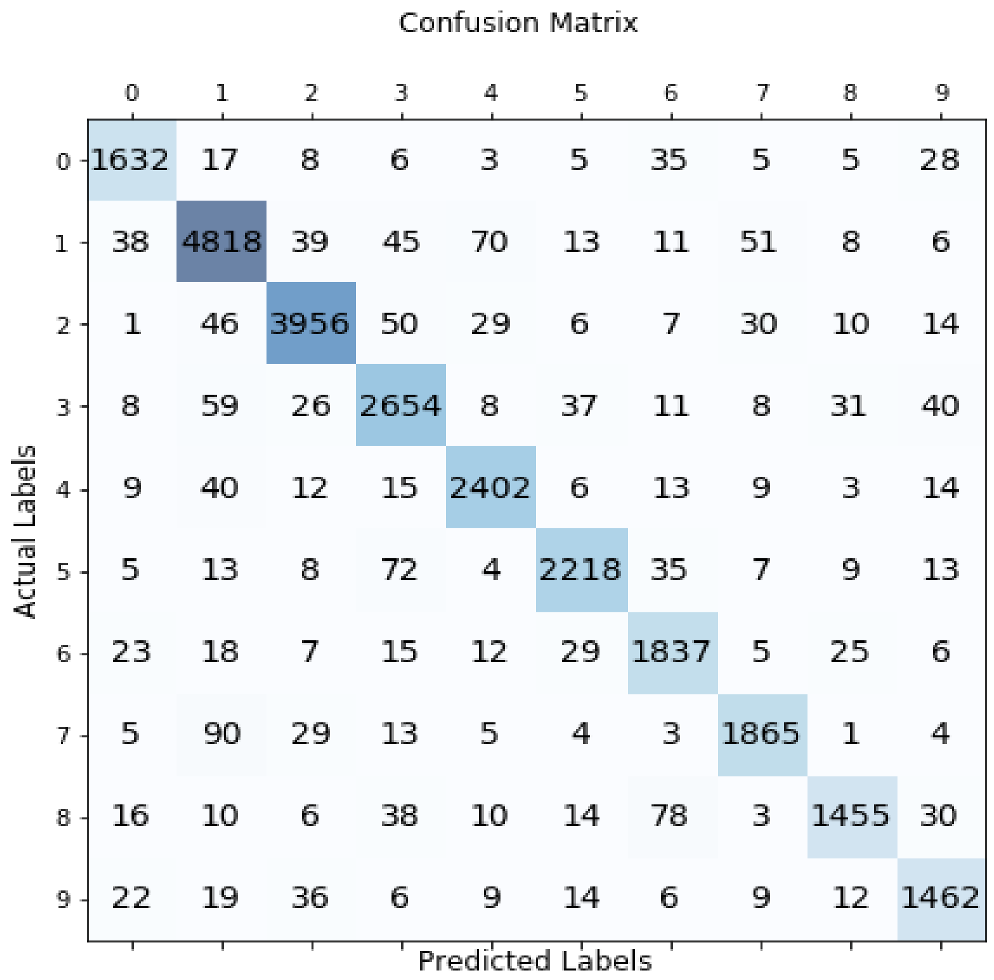
- Main python code accepted command line arguments to select different network architectures and hyper-parameters
- Each run results run written file
- Bash scripts to kick off runs in the background
- Used cloud GPU – way too slow on a CPU-only machine
- Performance measures collected each epoch:
  - Loss
  - Val Accuracy
- Weights saved for later visualization

```
ubuntu@ip-172-31-21-233:~/code/Final-Project-Group-7/code/pytorch$
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$
```

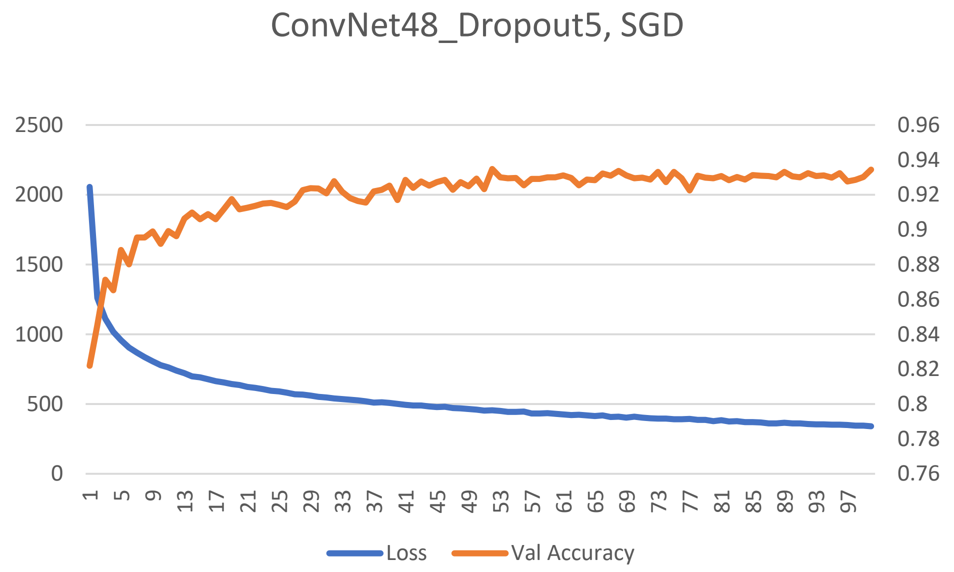
Results

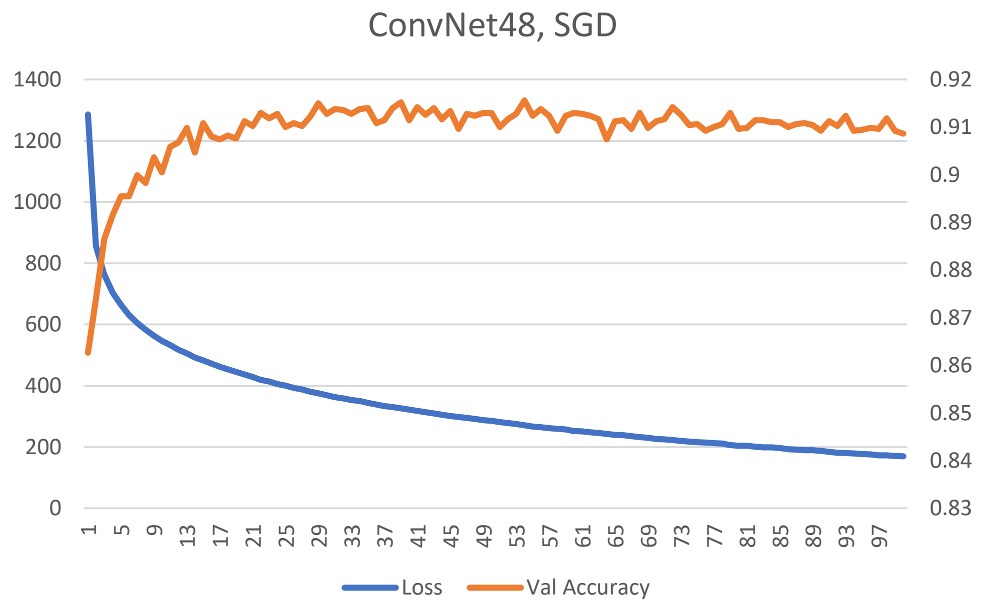
- Accuracy is the performance metric
- Achieved 91.6% on the test data

Digit	Precision	Recall	F1	# Samples
0	0.93	0.94	0.93	1744
1	0.94	0.94	0.94	5099
2	0.96	0.95	0.96	4149
3	0.91	0.92	0.92	2882
4	0.94	0.95	0.95	2523
5	0.95	0.93	0.94	2384
6	0.90	0.93	0.92	1977
7	0.94	0.92	0.93	2019
8	0.93	0.88	0.90	1660
9	0.90	0.92	0.91	1595
Avg / total	0.93	0.93	0.93	26032



Training Metrics

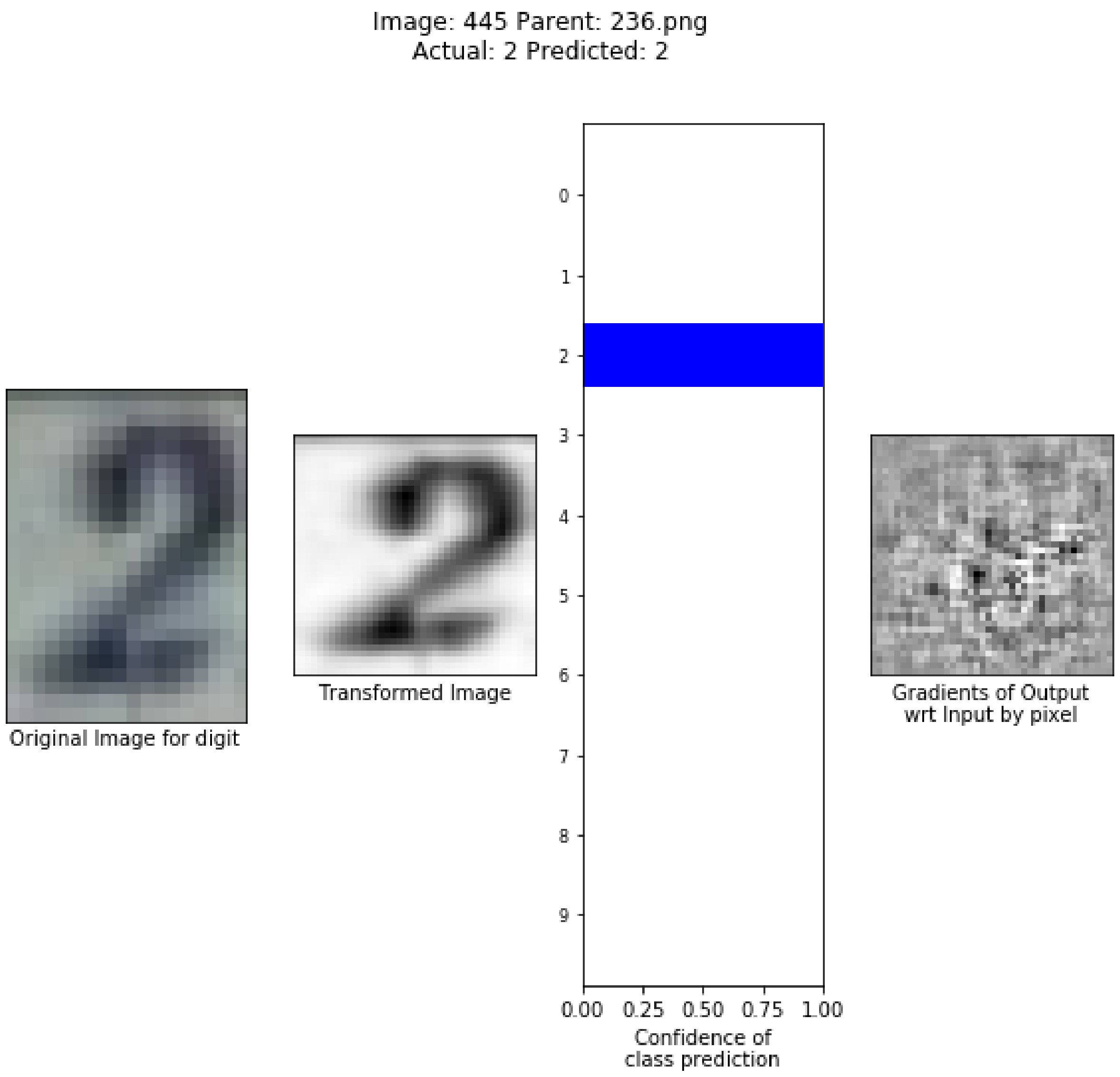




"Winning" network did not show overfitting signs but networks without dropout did

Identifying Key Pixels

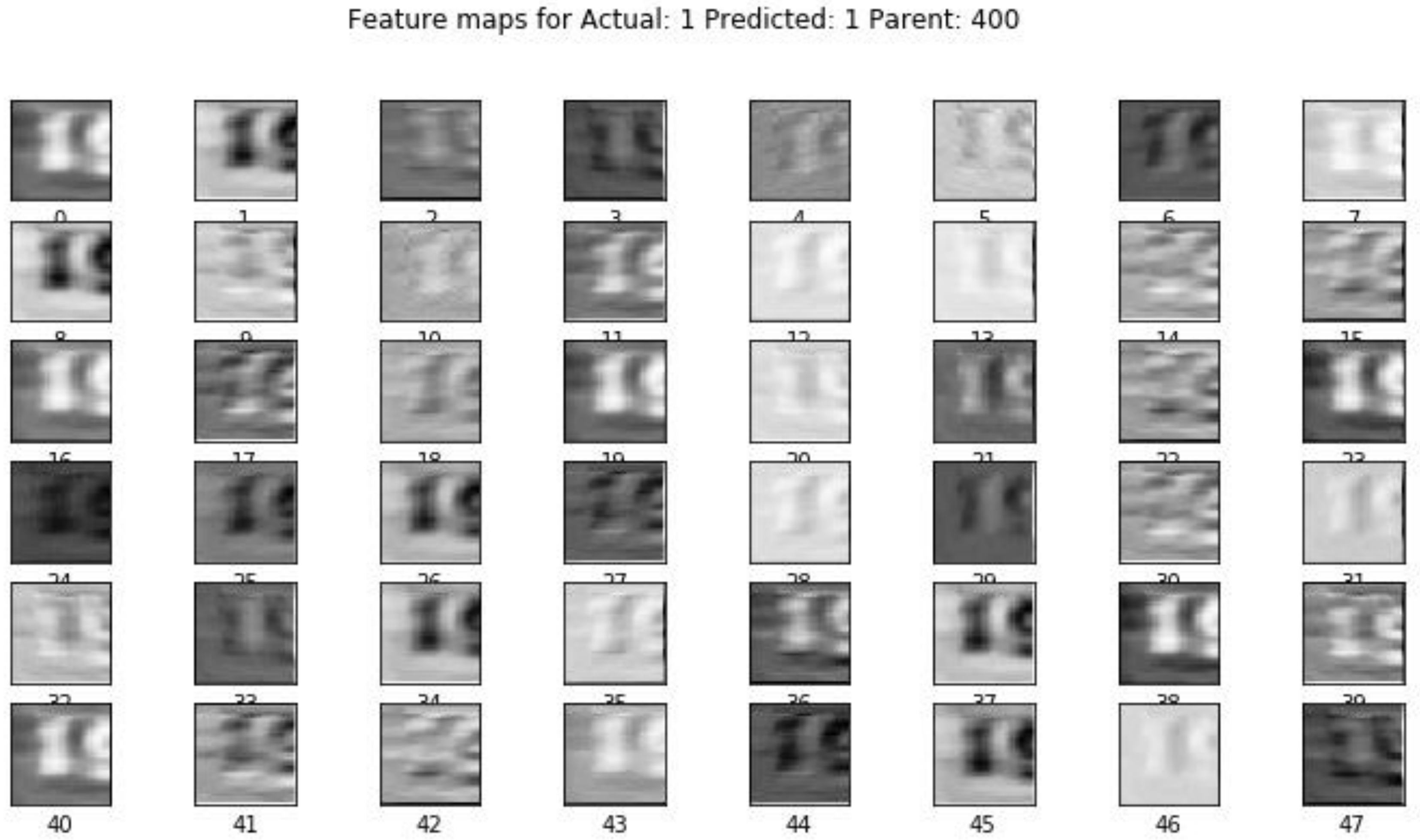
- Goal: Find Pixels likely to be in the background
- First thought, gradients of predicted output class with respect to input for each pixel
- Very noisy
- Promising, perhaps better suited to a supervised approach



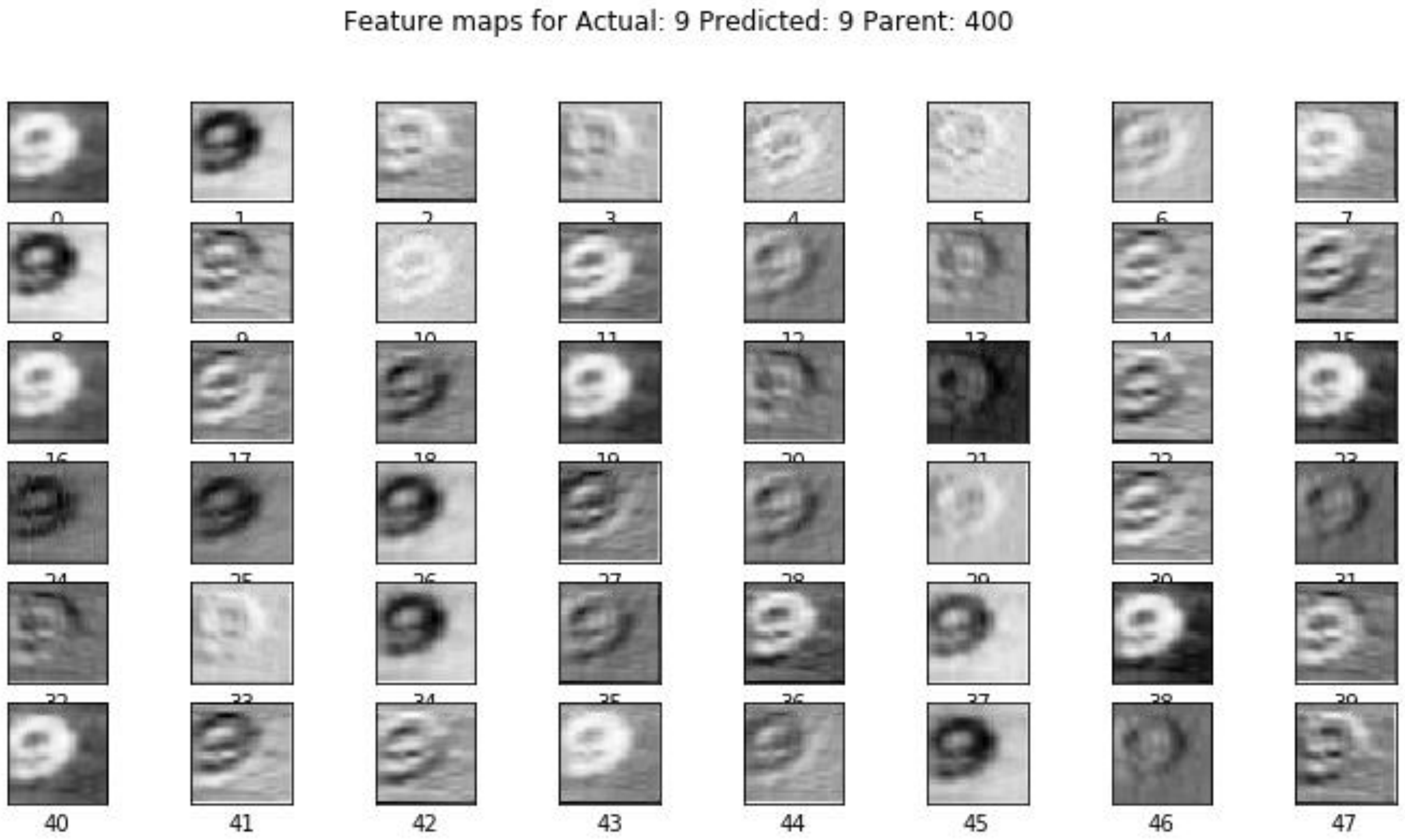
Next Try: Feature Maps from first Conv Layer

- First layer set up with stride, padding to have same Feature Map size as the image
- 48 Feature Maps from the first layer, reduced using mean() and std() across the pixel dimension – results in a tensor dimensioned the same as the image

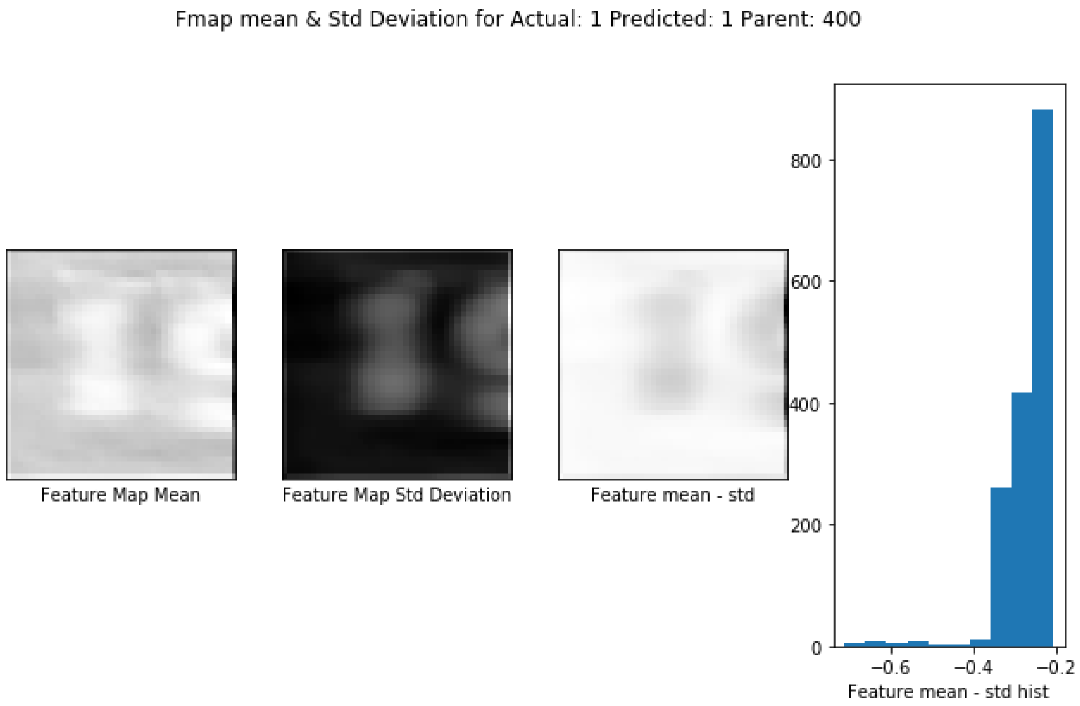
Feature Map for the first digit . . .

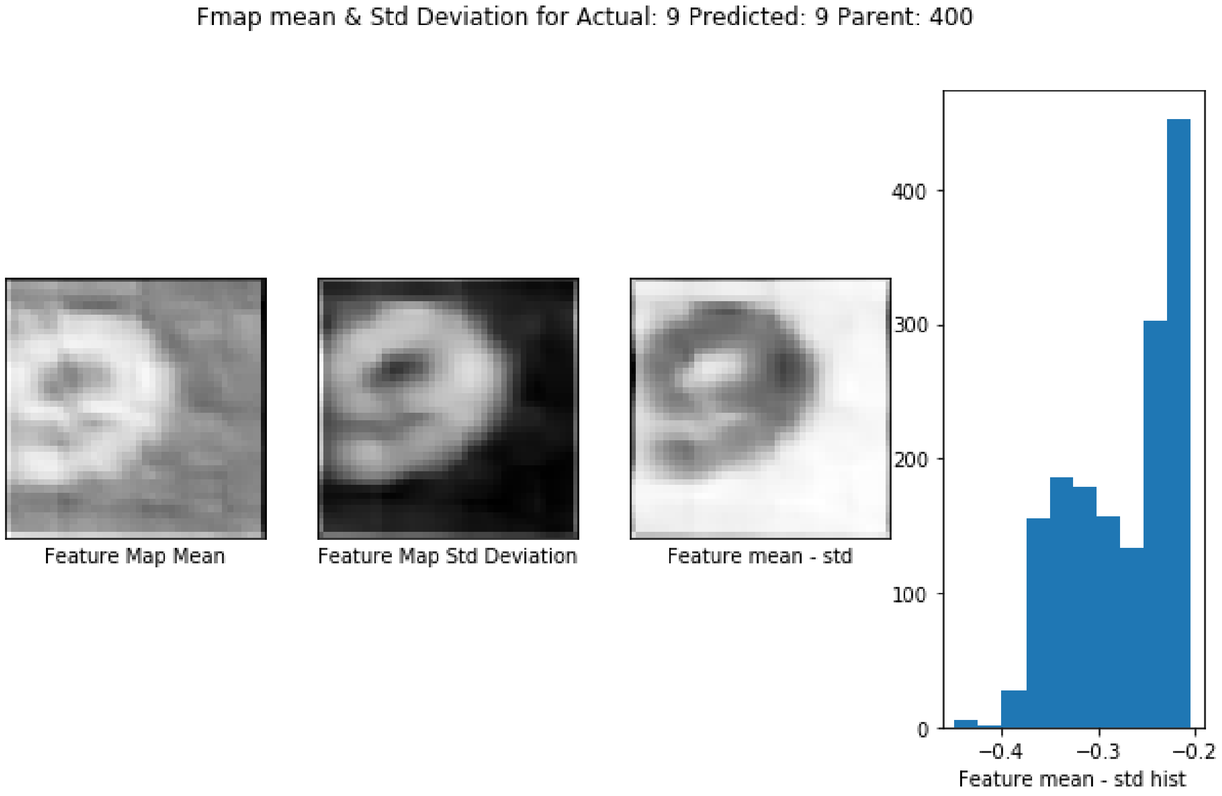


. . .And the second



Feature Map Reductions





Select pixels from largest histogram bucket of Feature Maps mean – standard deviation

Key Pixel Identification

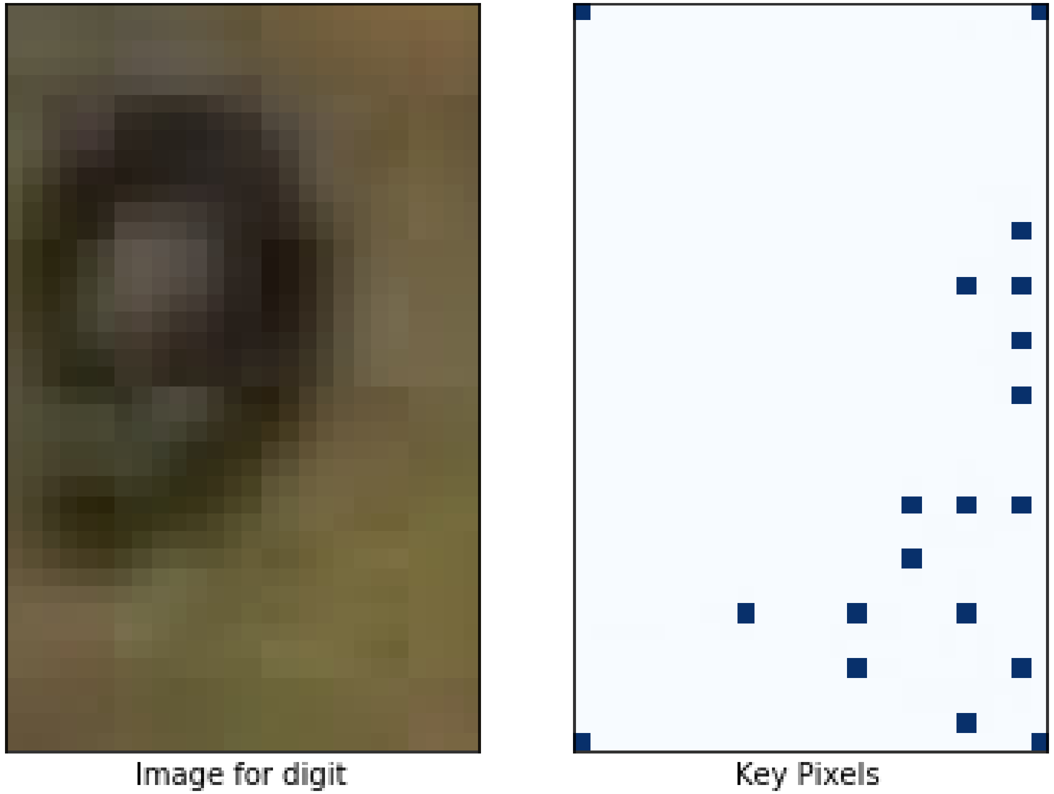


Image Generation

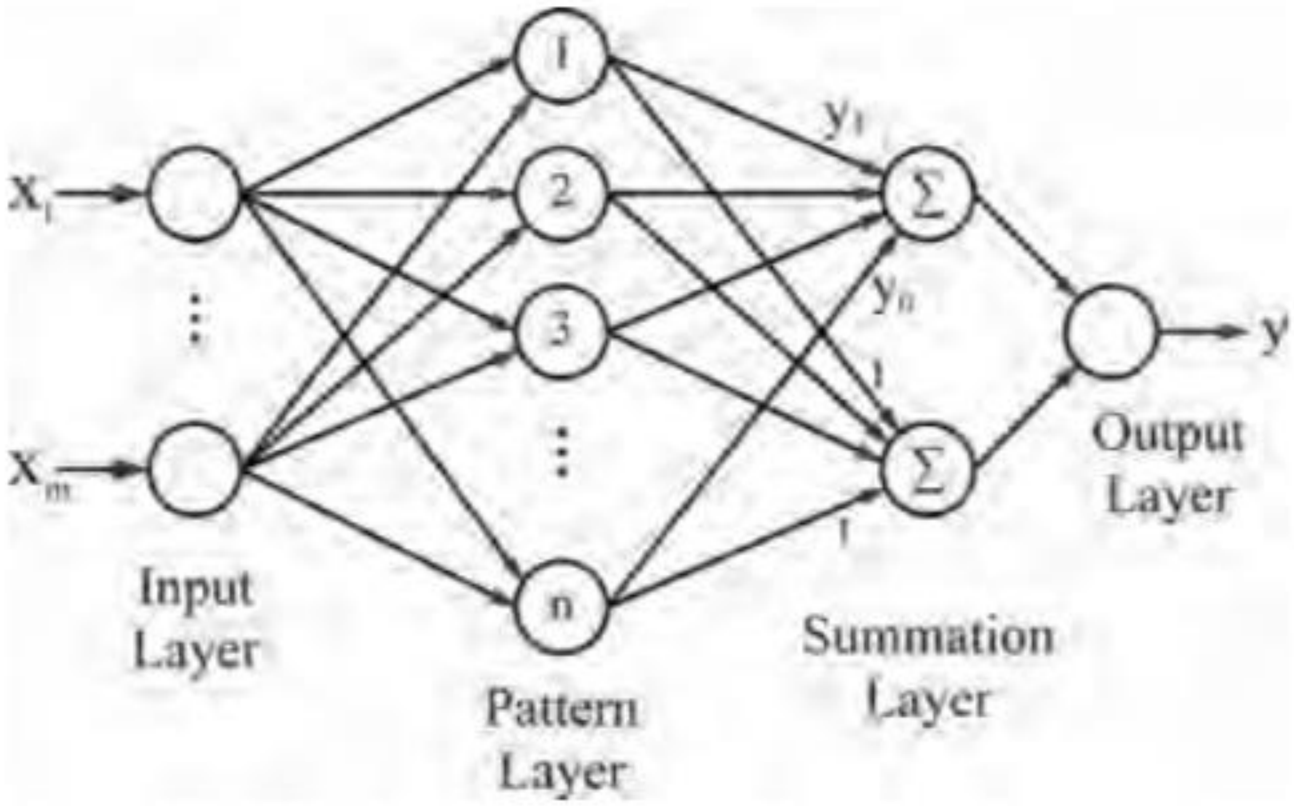
- With key pixels identified, use these to train a new Neural Network to generate a new image based on the background alone
- One Neural Network trained on each image
- The networks accept a pixel co-ordinate as input and output RGB pixel values

- Based on a paper about image infill after removing overlays:

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

Architecture

- Input: x,y co-ordinate
- Output: Pixel value (range[0..1])
- Pattern layer outputs distance from input X to every node to itself
- Weights between pattern layer nodes and top summation layer are learned
- Weights to bottom note in summation layer always 1
- Output layer transfer function is a radial basis function



Radial Basis Function for summation and output layers

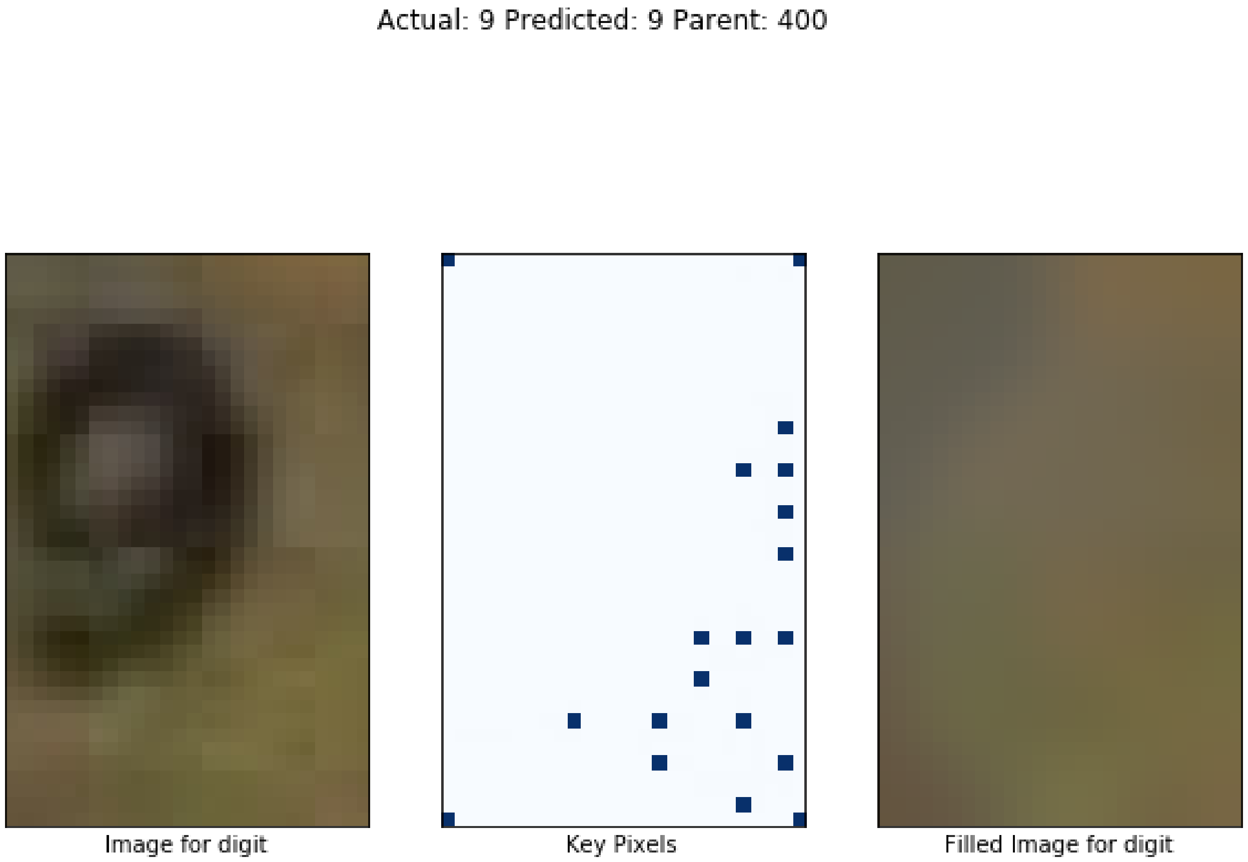
$$y = \hat{f}(X) = \frac{\sum_{i=1}^{n} y_i \cdot \exp\left(\frac{-D_i^2}{2\sigma^2}\right)}{\sum_{i=1}^{n} \exp\left(\frac{-D_i^2}{2\sigma^2}\right)}$$

- Gaussain: Nearby pattern nodes have greater influence than distant ones when predicting a pixel value
- Sigma – “spread” or “bias: determines how quickly the influence of a pattern node diminishes with distance

PyTorch Module (excerpt)

```python
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)
```

Results digit-by-digit



Final Results

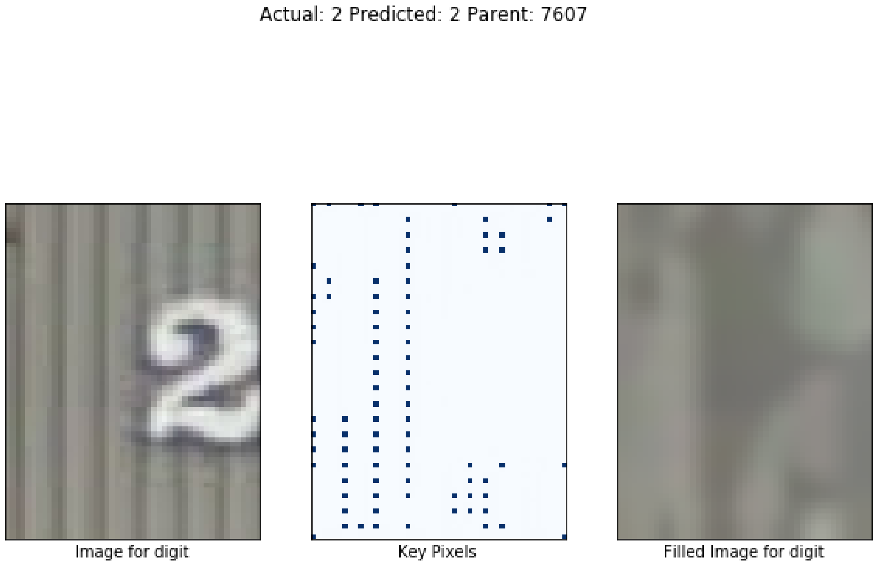
After pasting the generated images back into the parent, replacing the digit

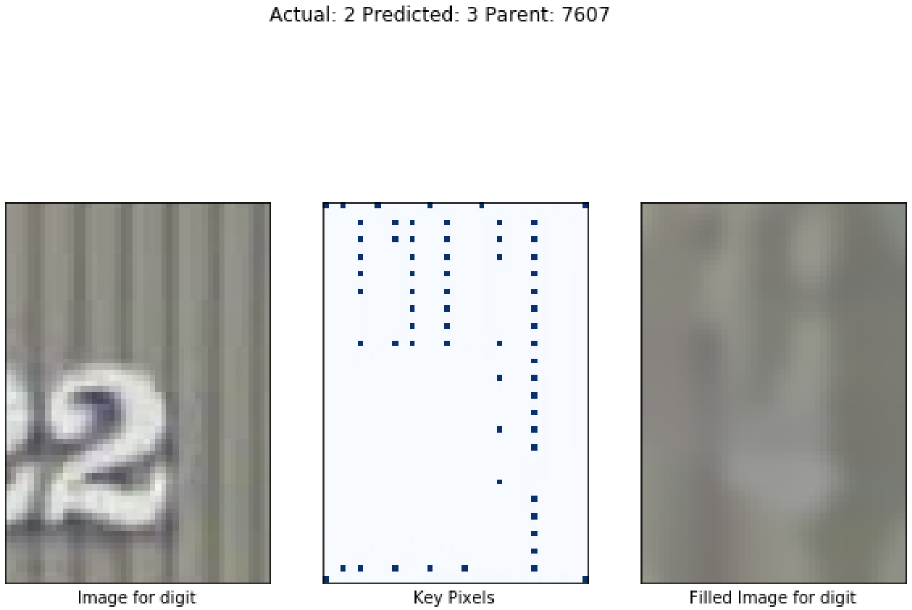
Parent Images before and digit replacement (Image 400)

Original　　Altered

Hits and Misses

Hit (7607)


Actual: 2 Predicted: 2 Parent: 7607
Image for digit
Key Pixels
Filled Image for digit


Actual: 2 Predicted: 3 Parent: 7607
Image for digit
Key Pixels
Filled Image for digit


Parent Images before and digit replacement (Image 7607)
22
Original
Altered

Hit (1201)

Actual: 5 Predicted: 5 Parent: 1201

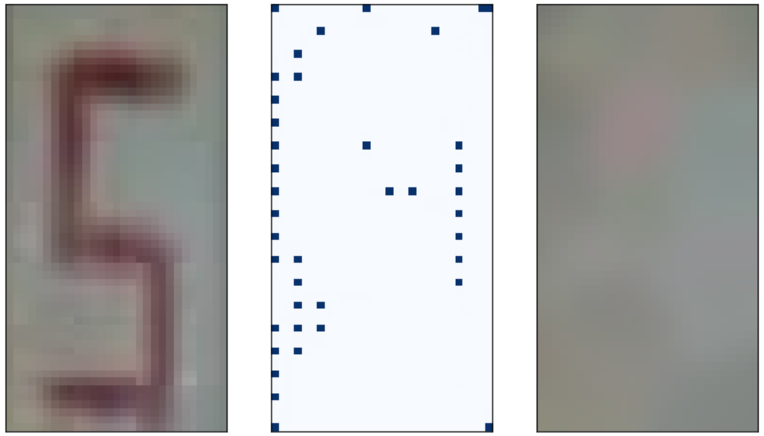


Image for digit | Key Pixels | Filled Image for digit

Actual: 4 Predicted: 4 Parent: 1201

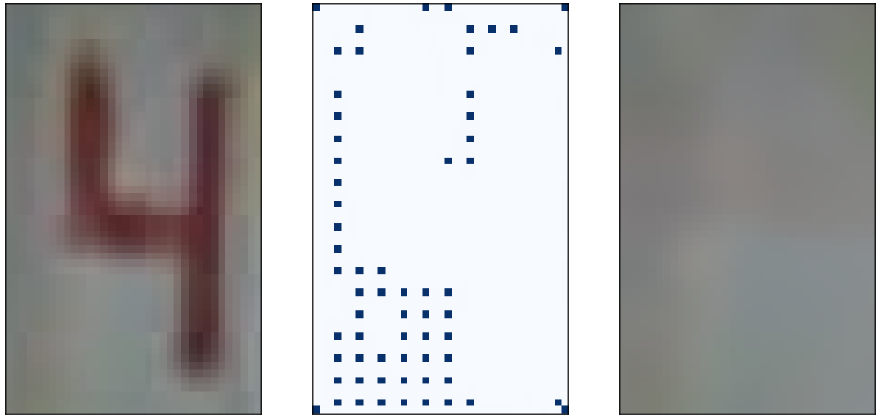


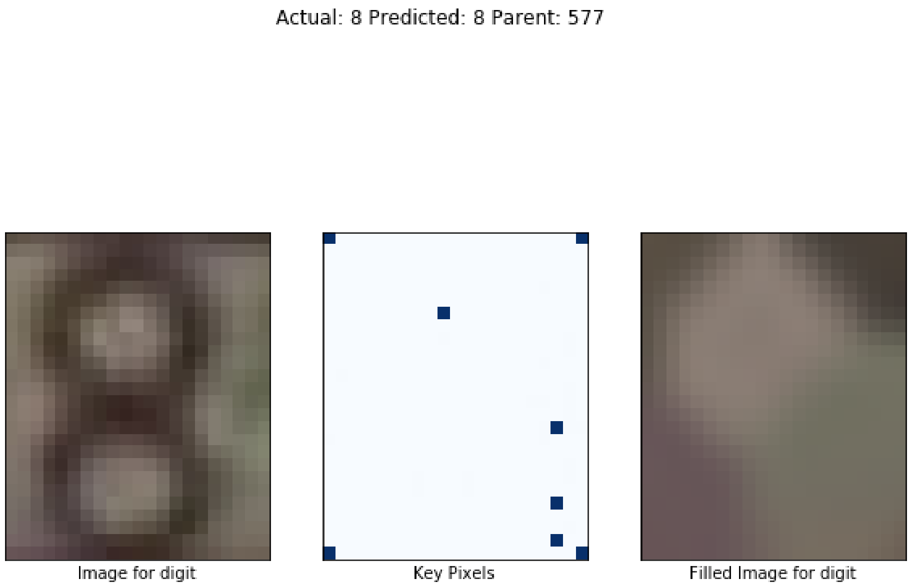
Image for digit | Key Pixels | Filled Image for digit

Parent Images before and digit replacement (Image 1201)

Original | Altered

Near Hit (577)


Actual: 1 Predicted: 1 Parent: 577
Image for digit
Key Pixels
Filled Image for digit


Actual: 8 Predicted: 8 Parent: 577
Image for digit
Key Pixels
Filled Image for digit


Parent Images before and digit replacement (Image 577)
Original
Altered

Miss (1360)

Actual: 3 Predicted: 3 Parent: 1360

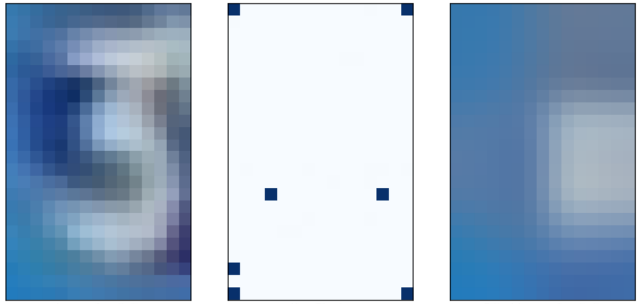


Image for digit Key Pixels Filled Image for digit

Actual: 4 Predicted: 4 Parent: 1360

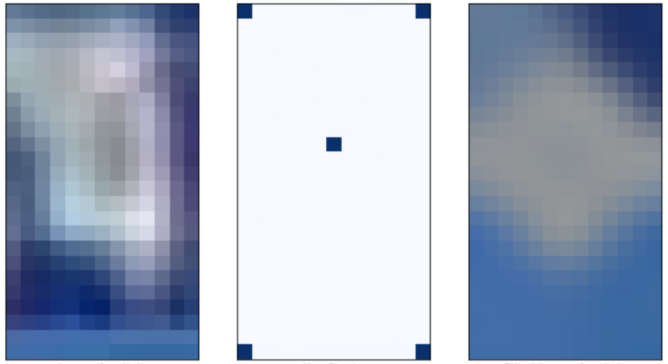


Image for digit Key Pixels Filled Image for digit

Actual: 2 Predicted: 2 Parent: 1360

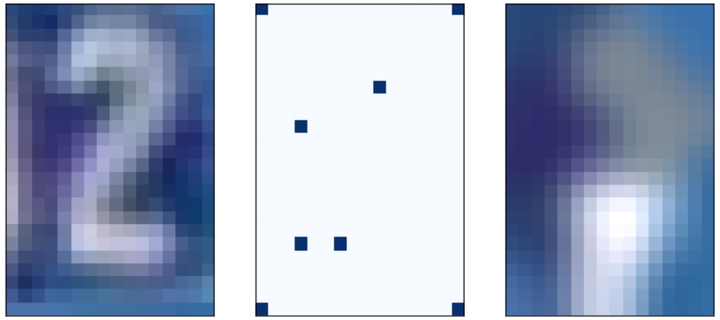
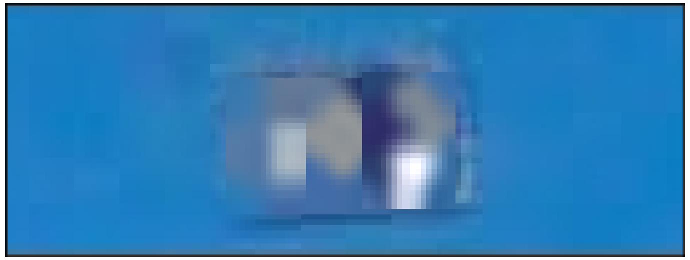


Image for digit Key Pixels Filled Image for digit

Parent Images before and digit replacement (Image 1360)

Original



Altered

Miss (10766)

Actual: 6 Predicted: 6 Parent: 10766

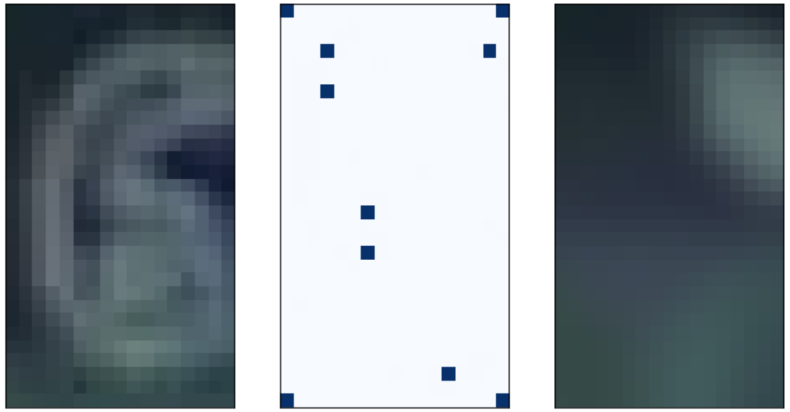


Image for digit Key Pixels Filled Image for digit

Actual: 2 Predicted: 2 Parent: 10766

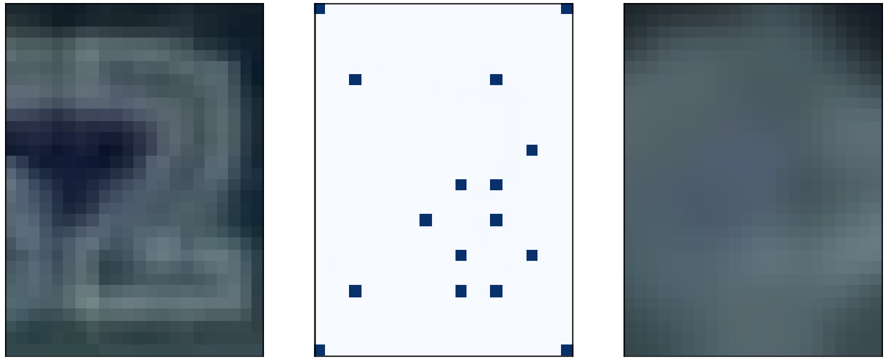
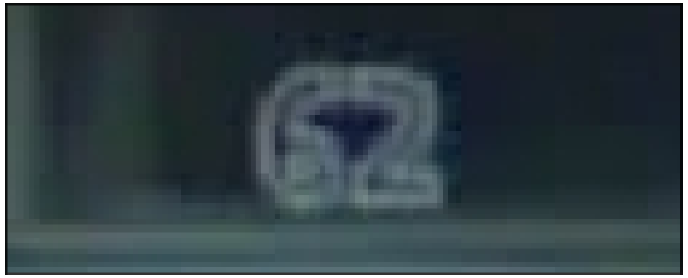
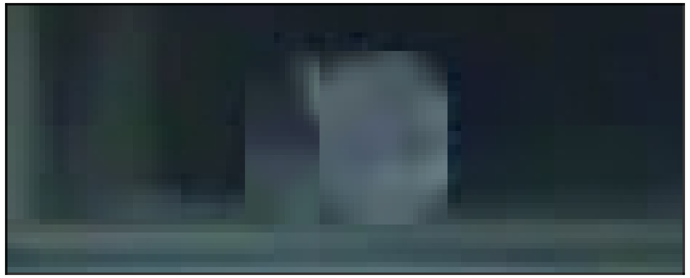


Image for digit Key Pixels Filled Image for digit

Parent Images before and digit replacement (Image 10766)



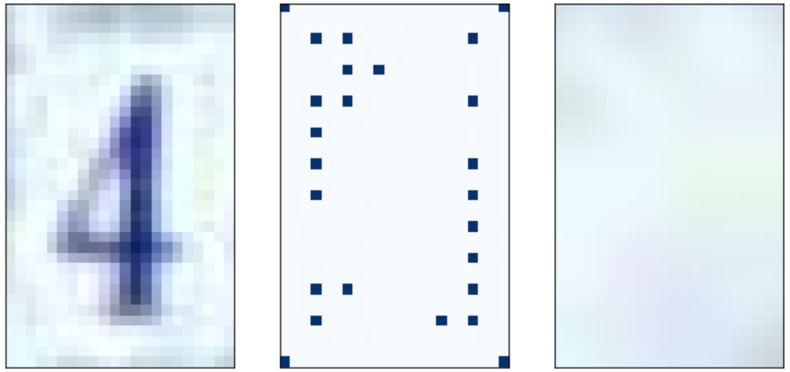
Original



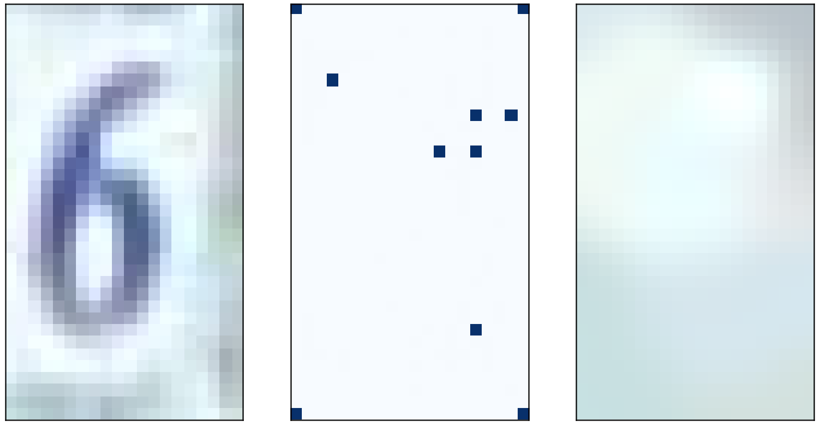
Altered

Hit (7686)

Actual: 4 Predicted: 4 Parent: 7686



Actual: 6 Predicted: 6 Parent: 7686



Parent Images before and digit replacement (Image 7686)

Conclusion

- The approach has promise
- The image infill algorithm is very powerful
  - If a few stray pixels are selected inside the digit (instead of the background) the algorithm does a remarkable job of recreating the digit (unwanted in this case)
- Identifying key background pixels to use as seed for the generated image is key
  - Fair number of misses; need to refine the key pixel selection
- Perhaps this method could be used to bootstrap training data for a supervised approach
- Combined with using gradients to identify key pixels, that could give could results