Capstone Project
Machine learning Engineer Nanodegree

Definition

Domain Background

In the capstone project, I seek to tackle a computer vision domain problem with deep learning. Deep learning has gained a lot of attention in recent years due to relative ease in training neural networks and the community growing enabling lot of interesting applications. In this project, we will train a deep learning model to recognize hadwritten digits within image. Such model canbe used in camera application to recognize digits in real time or by maps application to automatically tag buildings.

Project Overview

We will train a convolutional neural network with pretraining to recognize hadwritten digits within an image. The model is trained on Street View Housing Numbers dataset (SVHN). Most existing models preprocess the image and segment the digits and run the model on each segment but we will be training a model that does not require prior segmentation as proposed by Ian GoodFellow[1] .

Problem Statement/ Solution

We are going to train a Convolutional neural network to decode a sequence of digits within a natural image. CNN is a discriminative model and the problem is finding the most probable digit sequence given an image. To do so, we will pretrain a convolutional autoencoder of 4 layers on te training dataset of images 32x32 in size and then use results while training model on SVHN dataset connected to 6 softmax layers with 10 class labels corresponding to digits and space.

Evaluation Metrics

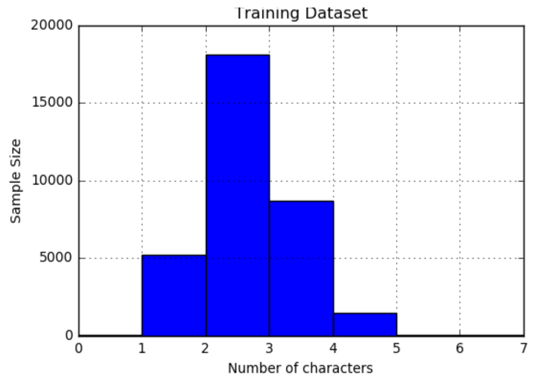
The SVHN dataset comes with testing dataset. The evaluation metric used is accuracy i.e. percentage of labels correctly predicted by the model on the testing data.

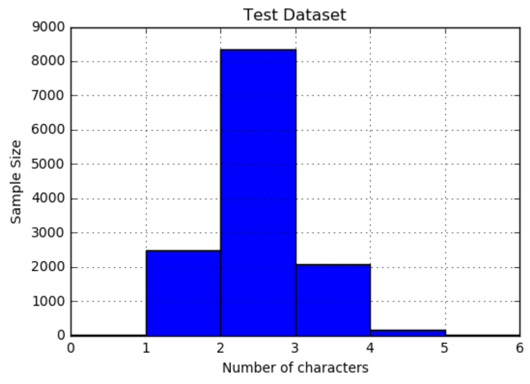
Analysis

The extraction of dataset and analysis of the data is present in the notebook PrepareTraining.ipynb. Download the dataset and extract in the same directory as the file.

SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. SVHN is obtained from house numbers in Google Street View images.

SVHN dataset comes in two fromats, one split into individual digits like MNIST and another with sequence of digits with character level bounding boxes. I will be using the second format. The images are in RGB color space. They are converted to gray scale for this project. The training dataset contains 33,402 images and testing dataset contains 13068 images with following distribution for length of numbers.





SVHN dataset is seperated into three seperate files for train, test and extra in gzip format.

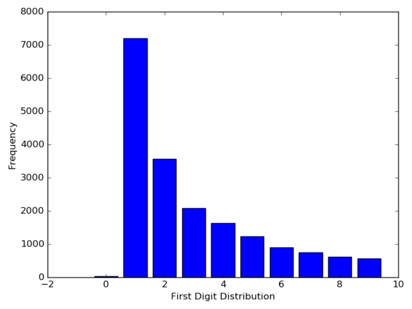
Each folder has digitStruct.mat file which is a dictionary specifying important attributes of the samples like digits in the image, bounding box of each digit and filename of actual image.

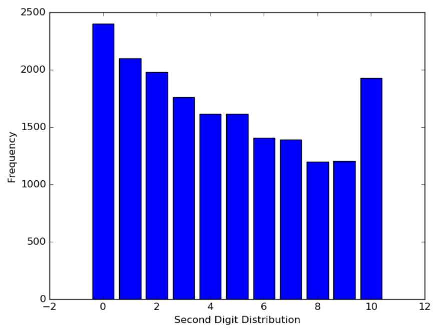
The character height (table 1) is distance between the top and the bottom of the bounding box

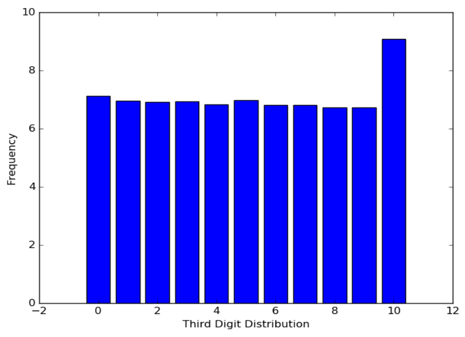
Dataset	Mean	Median	Standard Deviation
Train	34.366	30.0	19.378
Test	27.898	24.0	13.459

Table 1

Following images show number frequency distribution for each digit place.







The number of target labels in the dataset is 6 but we find that almost all of the 5th and 6th digits are space, So I chose to consider only the first four digits in the numbers. Fig. 2 is a sample size plot for 5 digits length per each digit. Plot on figure 3 shows digit wise number of samples for length 4.

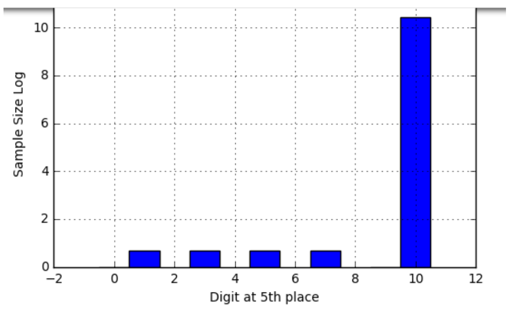


Figure 2

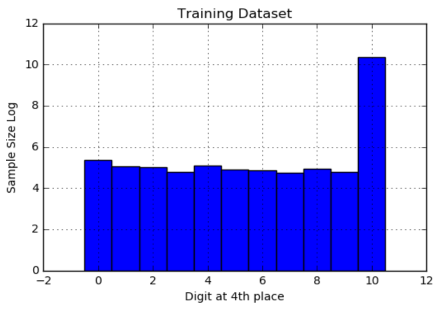
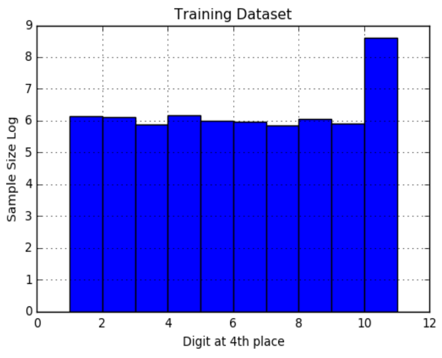


Figure 3

In digit place 4, No digit: 32175, 4: 161, 1: 156, 2: 150, 8: 140, 5: 134, 6: 130, 9: 121, 3: 120, 7: 115

If I directly consider the training data, the classifier will tend to overfit to space from 4th digit due to high imbalance towards space. So I undersampled data with labels containing space in 4th digit onwards by randomly picking 15000 samples and oversampled the data with labels containing digits in 4th place onwards by adding images from extra dataset.

Fig. 4 shows the sample sizes for 4th digit afer resampling for the training data.



No digit: 12026, 0.0: 1098, 5.0: 789, 1.0: 667, 7.0: 606, 6.0: 592, 2.0: 591, 3.0: 583, 4.0: 582, 9.0: 542, 8.0: 521

Figure 4

We also find the image histograms are concentrated in one side of intensity, so we perform local contrast normalization(LCN) to enhance the images (fig. 5).

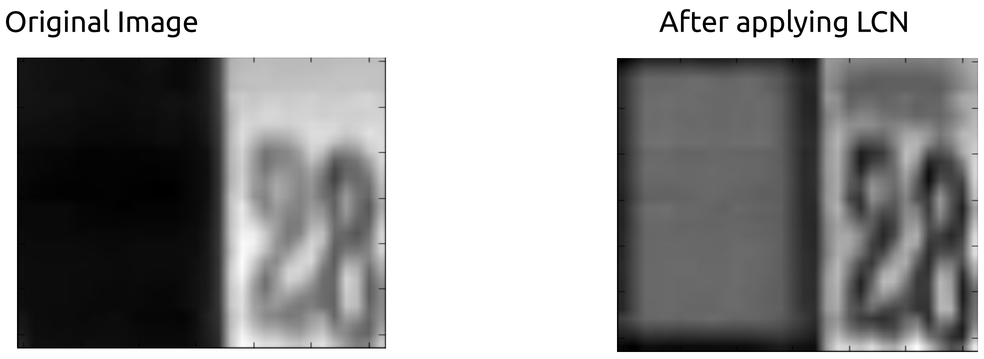


Figure 5

I will be using Convolutional neural network connected to a fully connected layer to recognize the sequence of digits. This is largely due to to the network being able to identify images at different scales. Another advantage here is that a convnet is able to directly work on raw pixels.

Pretraining

Unsupervised learning has been a powerful idea in rich feature extraction. It can be applied to our problem to arrive at a feature representation which can help us reach optimal result quicker.

The corresponding code is present in Pretrain.py. In this phase, I train a convolutional auto encoder (CAE) on the training dataset. The autoencoder has 4 convolutional and corresponding deconvolutional layers. Many data scientists have tried using pooling and unpooling layers but there is growing evidence that pooling layer is not really that beneficial as shown by Springenberg[2] and team from Freiburg university who observed convolutional layer with increased stride can replace pooling layer without any loss in accuracy, so I discarded the idea of having pooling layers.

Convolutional Autoencoder

Autoencoders are models to learn representation(encodings) for a data in unsupervised manner. It has a series of layers encoding by reducing number of dimensions connected to a series of layers decoding the representation trying to reconstruct the image back from representation. Particularly in a CAE, encoding layers are convolutional layers and decoding are deconvolutional layers. These learnt representations can further be trained for a particular classification problem. In our pretrainig, we will have 4 encoding and 4 deconding layers.

- conv0: convolutional layer, batch_size x 32 x 32 x 1, convolution size: 5 x 5 x 1 x 16 with stride 2 in every dimension
- conv1: convolutional layer, batch_size x 16 x 16 x 16, convolution size: 5 x 5 x 16 x 32
- conv2: convolutional layer, batch_size x 8 x 8 x 32, convolution size: 5 x 5 x 32 x 64
- conv3: convolutional layer, batch_size x 4 x 4 x 64, convolution size 4 x 4 x 64 x 128
- deconv0: deconvolutional layer, batch_size x 2 x 2 x 128, output shape batch_size x 4 x 4 x 64
- deconv1: deconvolutional layer, batch_size x 4 x 4 x 64, output shape batch_size x 8 x 8 x 32
- deconv2: deconvolutional layer, batch_size x 8 x 8 x 32, output shape batch_size x 16 x 16 x 16
- deconv3: deconvolutional layer, batch_size x 16 x 16 x 16, output shape batch_size x 32 x 32 x 1

After training CAE for 10000 iterations, the model had 0.03 loss and below is a reconstructed image by CAE of original from training dataset (fig. 6)

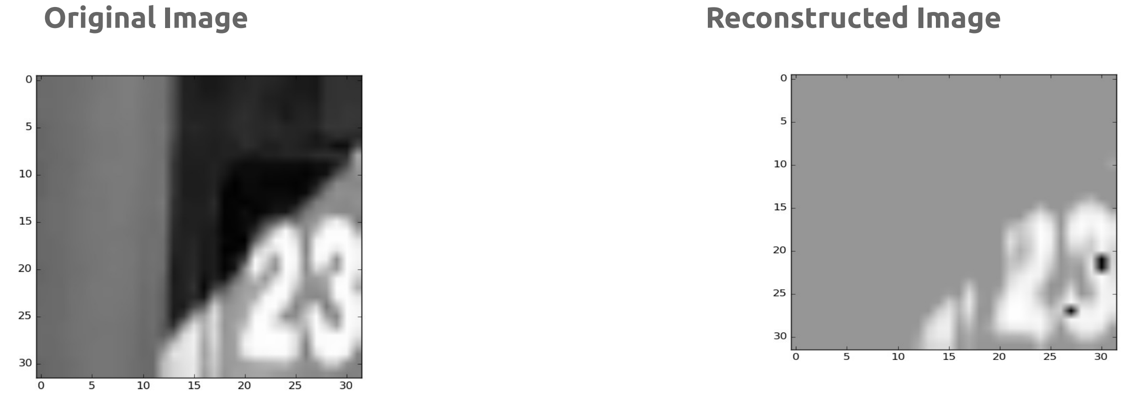


Figure 6

Classification model

Our classification model as explained earlier is a convolutional neural network 4 layers in depth connected to six softmax layers where each softmax has 10 classes.

Convolutional Neural Networks (CNN)

In a convolutional model, we're explicitly modeling what happens around a pixel and we're using the exact same convolutions no matter where in the image we are unlike in multi layer perceptron where we have set of weights for each pixel. Convolution is a process of multiplying an image with a kernel/filter (matrix of values) across the image to identify various features in an image like Gaussian for smoothing and Canny for edge detection. CNN are neural networks that can learn these kernels by themselves based on problem at hand.

Thus CNN extract features based on the learnt kernels. These features are then used in a fully connected network for classification. This whole architecture is then trained in a supervised manner with back propagation.

Following are some of the layers used in a CNN.

Images / Input Layer

This layer holds the the input which is equivalent to a store of the the pixels of all images.

Local Contrast Normalization

Local contrast normalization (LCN) is a method used to normalize the contrast of an image in a non-linear way. Instead of performing a global normalization based on the range of values of the entire image, LCN operates on local patches of the image on a per pixel basis.

Convolution Layer

Convolution layer share their parameters across space. It works by sliding over the vector with depth, height and width and producing another matrix(called a convolution) that has a different weight, depth height. Essentially, rather than having multiple matrix multipliers, we have a set of convolutions.

Local Response Normalization

Local Response Normalization(LRN) type of layer turns out to be useful when using neurons with unbounded activations (e.g. rectified linear neurons), because it permits the detection of high-frequency features with a big neuron response, while damping responses that are uniformly large in a local neighborhood. It is a type of regularizer that encourages "competition" for big activities among nearby groups of neurons.

ReLu:

The rectifier function is an activation function that can be computed efficiently compared to more conventional activation functions like the sigmoid and hyperbolic tangent, without making a significant difference to generalisation accuracy. The rectifier activation function is used instead of a linear activation function to add non linearity to the network, otherwise the network would only ever be able to compute a linear function.

Dropout is a technique where we ignore half of the nodes randomly when training the weights and halve the weight when predicting. This makes the training faster and also is practically the best way to prevent overfitting. We choose maximum of all the responses in a given neighborhood.

Fully Connected Layer

This layer connects every neuron from the previous convolutions to every neuron that it has. It converts a spatial like network to a 1d network, so we can then use this network to produce our output

The Output layer

The output from this layer are logits which represents matrix showing the probabilities that of having a character in a particular position

Benchmark Model

Benchmark model for this problem is linear Support Vector Machines (SVMs) trained on Histogram of Oriented Gradient (HOG) features. I have trained this model on our training data and achieved accuracy of 70 %.

Methodology

Data Preprocessing

We performed the following steps to preprocess data:

1. Extract information from the digit Struct and save it in a python friendly format.
2. Generate new images using the bounding boxes in the previous image
3. Resize our new images to 32x32 pixels because CNN expects a fixed size images. We train the model on the boxed regions only. This model can be enhanced further by either selective searching or sliding windows approach to extract regions
4. Generate final training, testing and validation dataset. The validation set is selected by sampling the extra dataset
5. During exploration step, we find histograms are concentrated on one side, so we perform local contrast normalization
6. We pretrain the weights of the CNN model with a convolutional autoencoder on training, extra and testing dataset

Implementation

The training of the CNN is present in the file Training.py. We train the model in batches of training dataset. Once that is done, we save the trained model to be used in Prediction.ipynb.

Model

- Local Contrast Normalization is performed on the image with gaussian kernel
- C0: convolutional layer, batch_size x 32 x 32 x 1, convolution size: 5 x 5 x 1 x 16 with stride 2 in every dimension
- C1: convolutional layer, batch_size x 16 x 16 x 16, convolution size: 5 x 5 x 16 x 32
- C2: convolutional layer, batch_size x 8 x 8 x 32, convolution size: 5 x 5 x 32 x 64
- C3: convolutional layer, batch_size x 4 x 4 x 64, convolution size 4 x 4 x 64 x 128
  - After Dropout, fullyconnected layer, weight size: 512 x 128
- fullyconnected layer, weight size: 128 x 32
- Output layer, weightsize: 32 x 11

To train, we read the already preprocessed data into the model and train it in batches. During the training, we try to minimize loss and log the accuracy we are achieving so we can keep track of how well our model is improving. Once the model is trained, we evaluate it using our test step. One step further would be to save the model and load the model so it can be used in other applications such as an android app.

The final model is made up of the following:

- Weights are pretrained with convolutional auto encoder
- We implemented convolutions for depths at 16, 32, 64 and 128
- We created 4 classifiers / logits
- We implemented learning rate decay at 0.05
- We used AdagradOptimizer as our optimizer

- We introduced dropout just before the fully connected layer with 0.8 keep probability
- We used accuracy as our metric

Optimization

With CNN hyper parameters used, I got accuracy of 80% over 10000 iterations on testing dataset. To fine tune the CNN hyper parameters, I have performed Bayesian optimization loop for 30 iterations. For this task, I used Sigopt library to give hyper parameter suggestions. Figure 7 gives the improvement achieved with 1000 epochs for CNN and 500 iterations for pretraining.

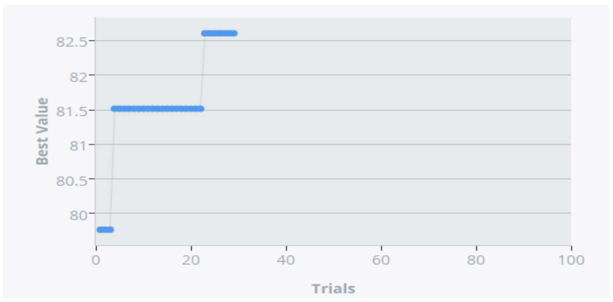


Figure 7

The best parameters after Bayesian optimization are:

Parameter	Optimal Value
Layer 1 depth	20
Layer 1 Kernel size	5
Layer 2 depth	40
Layer 2 Kernel size	5
Layer 3 depth	60
Layer 3 Kernel size	4
Layer 4 depth	140
Layer 4 Kernel size	3

The above parameters gave 83% accuracy over 10000 iterations on testing dataset.

I found the validation accuracy not improving while training accuracy is reaching 100 % clearly suggesting overfitting to the data. To solve the problem

1. I introduced L2 regularization on weights in hidden layer of fully connected network
2. Drop out between the fully connected networks

This improved the validation accuracy to 87.8 % and testing accuracy to 88 %.

Results

Our final CAE has learnt representations with just 0.03 loss and the classification model had accuracy of 88 % with the test dataset.

Training Accuracy: 97.3 %   Validation Accuracy: 87.8 %   Testing Accuracy: 88 %

Left plot on fig. 8 show accuracy over iterations and right plot on figure 8 shows loss over iterations

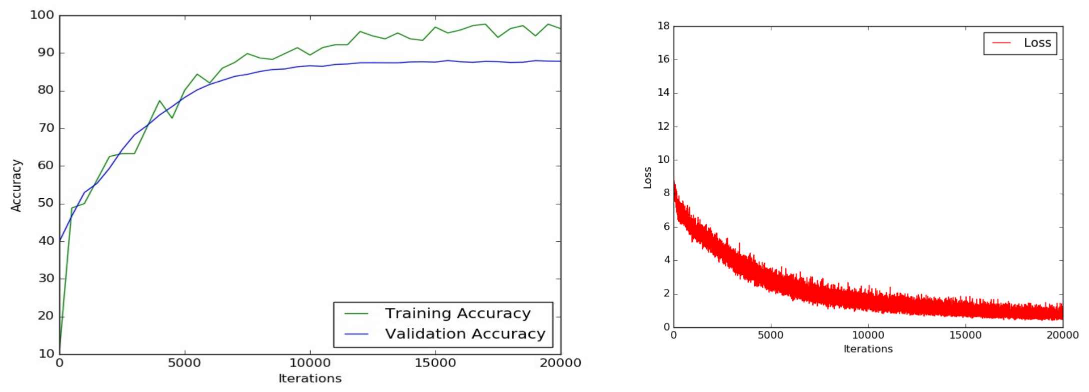
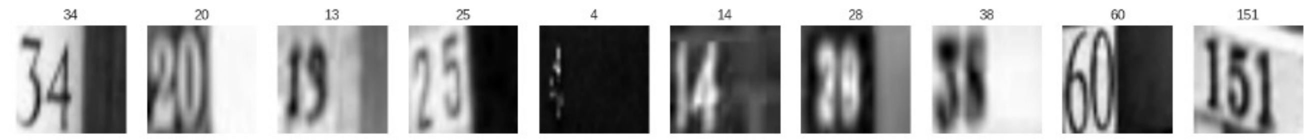


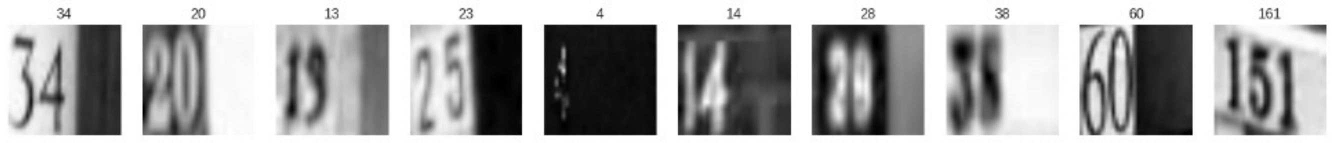
Figure 8

The above plot of accuracies show that our model is robust enough as the accuracies of training and validation dataset are close. It shows that our model can perform well on unseen data.

Following image (fig. 9) shows predictions for 10 input images from testing dataset.



Correct labels



Predicted Labels

Figure 9

We see that most of the digits are recognized correctly. With 151, it is possible to confuse 5 for 6 as it looks very close i.e. almost connected whereas with 25, there is some similarity between 5 and 3 in the sense that digit is tilted making the upper line similar to that of 3. These kind of corner cases can be avoided with more data i.e. with extra dataset or oversampling with rotations in both directions.

Confusion matrix is a standard visualizing tool used in classification problems involving multi classes. It depicts which class labels are confused with other class variables. Following image (figure 10) shows confusion matrix of the final model.

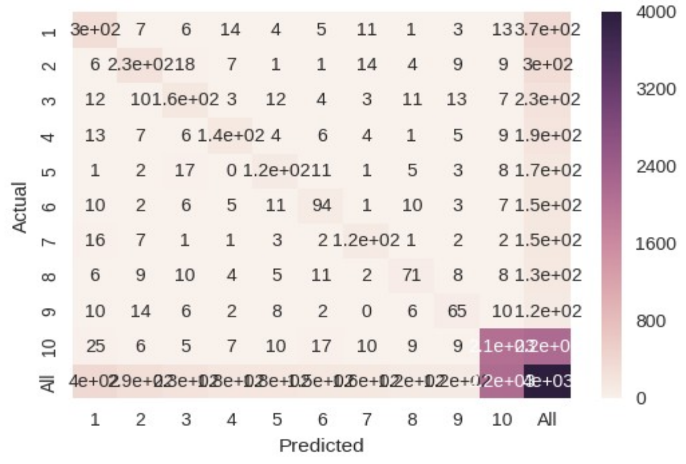


Figure 10

Justification

Our benchmark model has accuracy of 70 %. Our model was able to achieve accuracy of 88 % whe trained for 20000 iterations. Since we have over 200,000 more images to train on, we expect the accuracy to continue to improve moving closer to human recognition. One can achieve better accuracy with more number of iterations and a deeper model. Also since we resized to pretty lowe resolution of 32 lot of information is lost, if we had used bigger image size, the accuracy could have been better.

According to the paper written on this model, one can achieve over 97% accuracy with 11 layers and 4096 features model.

Conclusion

To summarize the project,

- I have trained a deep learning model to recognize sequence of digits in a natural image.
- For this task, I have used publicly available SVHN dataset.
- After resizing the images to 32 x 32 fixed size, I have pretrained weights with a Convolutional auto encoder upon the dataset.
- I found the dataset to be unbalanced so oversampled and undersampled the classes to prepare training data and chose accuracy on testing dataset as the evaluation metric.
- I have implemented a 4 layer deep CNN connected to four 3 layer deep fully connected networks to classify the digits.
- While training I find the validation accuracy not improving with traiining accuracy, so introduced L2 regularization term, drop out and exponential decaying learning rate.
- To fine tune the hyper parameters, I have used Bayesian optimization upon CNN parameters, learning rate, drop out probability.

The first challenge I had with implementing the project has to do with my system. It takes a lot of time to train a neural network on a single CPU. So while performing Bayesian optimization, I have used AWS server along with my system both running optimization loops.

Another challenge I faced is with accuracy not improving on validation dataset while training accuracy is reaching 100 % clearly suggesting overfitting and can be solved with more data, regularizaton and drop out. So I added some images from extra dataset to oversample instead simple schema like introducing random rotations. I added L2 regularization on hidden layer of fully connected network and drop out between the fully connected layers with exponential decay learing rate.

Improvement

There are several ways this implementation can be improved. One of which is to use Recurrent neural networks. Using that implementation, we would be able to predict beyond 5 digits at once. In the OCR problem, this approach had recently gained attention due to Connection Temporal Classification loss. We can either pretrain convolutional neural network using unsupervised learning or train entire model together.

References

1. Goodfellow, Ian J., et al. "Multi-digit number recognition from street view imagery using deep convolutional neural networks." arXiv preprint arXiv:1312.6082 (2013)

2. Josh Tobias Springenberg., et.al. “Striving For Simplicity: The All Convolutional Net“

Results

Training CAE

```
Cost after 0 steps is 0.0789925381541
Cost after 500 steps is 0.0369241349399
Cost after 1000 steps is 0.0324857160449
Cost after 1500 steps is 0.0404012463987
Cost after 2000 steps is 0.0412800423801
Cost after 2500 steps is 0.034858006984
Cost after 3000 steps is 0.0402882583439
Cost after 3500 steps is 0.0316104143858
Cost after 4000 steps is 0.0316808894277
Cost after 4500 steps is 0.0361944623291
Cost after 5000 steps is 0.0310679078102
Cost after 5500 steps is 0.0335410125554
Cost after 6000 steps is 0.0435601323843
Cost after 6500 steps is 0.0373108573258
Cost after 7000 steps is 0.0424386225641
Cost after 7500 steps is 0.0276077352464
Cost after 8000 steps is 0.0331194214523
Cost after 8500 steps is 0.0347782224417
Cost after 9000 steps is 0.0359886661172
Cost after 9500 steps is 0.0313676968217
```

Training CNN classification model

```
Batch accuracy at step 0: 10.5%
Validation accuracy at step 0:39.9%
Batch accuracy at step 500: 48.8%
Validation accuracy at step 500: 46.7%
Batch accuracy at step 1000: 50.0%
Validation accuracy at step 1000: 52.9%
Batch accuracy at step 1500: 56.2%
Validation accuracy at step 1500: 55.3%
Batch accuracy at step 2000: 62.5%
Validation accuracy at step 2000: 59.4%
Batch accuracy at step 2500: 63.3%
Validation accuracy at step 2500: 64.2%
Batch accuracy at step 3000: 63.3%
Validation accuracy at step 3000: 68.3%
Batch accuracy at step 3500: 70.3%
Validation accuracy at step 3500: 70.7%
Batch accuracy at step 4000: 77.3%
Validation accuracy at step 4000: 73.5%
Batch accuracy at step 4500: 72.7%
Validation accuracy at step 4500: 75.8%
```

```
Batch accuracy at step 5000: 80.1%
Validation accuracy at step 5000: 78.2%
Batch accuracy at step 5500: 84.4%
Validation accuracy at step 5500: 80.2%
Batch accuracy at step 6000: 82.0%
Validation accuracy at step 6000: 81.6%
Batch accuracy at step 6500: 85.9%
Validation accuracy at step 6500: 82.7%
Batch accuracy at step 7000: 87.5%
Validation accuracy at step 7000: 83.8%
Batch accuracy at step 7500: 89.8%
Validation accuracy at step 7500: 84.3%
Batch accuracy at step 8000: 88.7%
Validation accuracy at step 8000: 85.1%
Batch accuracy at step 8500: 88.3%
Validation accuracy at step 8500: 85.6%
Batch accuracy at step 9000: 89.8%
Validation accuracy at step 9000: 85.8%
Batch accuracy at step 9500: 91.4%
Validation accuracy at step 9500: 86.3%
Batch accuracy at step 10000: 89.5%
Validation accuracy at step 10000: 86.6%
Batch accuracy at step 10500: 91.4%
Validation accuracy at step 10500: 86.4%
Batch accuracy at step 11000: 92.2%
Validation accuracy at step 11000: 86.9%
Batch accuracy at step 11500: 92.2%
Validation accuracy at step 11500: 87.1%
Batch accuracy at step 12000: 95.7%
Validation accuracy at step 12000: 87.4%
Batch accuracy at step 12500: 94.5%
Validation accuracy at step 12500: 87.4%
Batch accuracy at step 13000: 93.8%
Validation accuracy at step 13000: 87.4%
Batch accuracy at step 13500: 95.3%
Validation accuracy at step 13500: 87.4%
Batch accuracy at step 14000: 93.8%
Validation accuracy at step 14000: 87.6%
```

```
Batch accuracy at step 14500: 93.4%
Validation accuracy at step 14500: 87.7%
Batch accuracy at step 15000: 96.9%
Validation accuracy at step 15000: 87.6%
Batch accuracy at step 15500: 95.3%
Validation accuracy at step 15500: 88.0%
Batch accuracy at step 16000: 96.1%
Validation accuracy at step 16000: 87.7%
Batch accuracy at step 16500: 97.3%
Validation accuracy at step 16500: 87.5%
Batch accuracy at step 17000: 97.7%
Validation accuracy at step 17000: 87.8%
Batch accuracy at step 17500: 94.1%
Validation accuracy at step 17500: 87.7%
Batch accuracy at step 18000: 96.5%
Validation accuracy at step 18000: 87.5%
Batch accuracy at step 18500: 97.3%
Validation accuracy at step 18500: 87.6%
Batch accuracy at step 19000: 94.5%
Validation accuracy at step 19000: 88.0%
Batch accuracy at step 19500: 97.7%
Validation accuracy at step 19500: 87.8%
Batch accuracy at step 20000: 96.5%
Validation accuracy at step 20000: 87.8%
Test accuracy: 88.4%
```