Capstone Project

Machine Learning Engineer Nanodegree

Saurabh Ratna
October 9th, 2016

I. Definition

Project Overview

In this project, we will build a model that can decode sequence of digits from natural images and outputs the number it sees in real time. This can be used in real-life situation like recognizing house numbers from Google Street View images and converting them into text addresses. It also can be used converting digits in image to text for further processing of numbers as text. We will be using SVHN Dataset which is s a real-world image dataset for developing machine learning algorithms with minimal requirement on data preprocessing and formatting.

Problem Statement

In this project, we will build a model that can decode sequence of digits from natural images. We will train a machine learning model to identify a number within a natural image. To help develop our model, we will generate a synthetic dataset using MNIST Dataset. Once we have a good architecture, we will train our model on real data. We will then use SVHN Dataset which is a good large scale data set collected from house numbers in Google Street View. Further breaking down, we will take following steps to achieve our solution. (This approach is taken from Udacity Deep Learning Course)

Step 1: Data Exploration

We will first explore the dataset. We will create a validation set from train and test set by randomly selecting data.

Step 2: Data Customisation

We will create a synthetic dataset by concatenating digits in the MNIST dataset and then split into **train, validation and test dataset**.

Step 3: Data Curation

Then we will create the data in such a way so that it is more amenable to machine learning processes. Ideally, we would want our data to be zero mean and small variance. We could use Gaussian Normalization and resize the images to have same size of all the images.

Step 4: Model Development

We will create a data model using convolution neural networks and we will train our model using the training and validation dataset.

Step 5: Test/Measure Performance

We will then measure the performance of our model using the test dataset.

Step 6: Measure Performance on Live Data

Finally, we will measure the performance of our model on SVHN data set. We will iterate the training process on our model until a satisfying performance is achieved.

Metrics

For measuring the performance of the model, we will check how many house numbers were correctly identified. We will count an identification as correct only when all the digits are recognized correctly as a house number is incorrect even if a digit in it is incorrect. So we will use the percentage of correctly identified house number as performance measure metric for the model.

II. Analysis

Data Exploration

We are using the SVHN Dataset. SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as 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.

There are two formats available for download:

Format 1: Full Numbers

Fig. 1: Full numbers in bounded boxes

These are the original, variable-resolution, color house-number images with character level bounding boxes, as shown in the examples images above. (The bounding box information are stored in **digitStruct.mat** instead of drawn directly on the images in the dataset.) Each tar.gz file contains the original images in png format, together with a digitStruct.mat file. The digitStruct.mat file contains a struct called **digitStruct** with the same length as the number of original images. Each element in digitStruct has the following fields: **name** which is a string containing the filename of the corresponding image. **bbox** which is a struct array that contains the position, size and label of each digit bounding box in the image.

Format 2: Cropped Digits

Fig 2: Cropped Digits to 32x32

Character level ground truth in an MNIST-like format. 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.

After we extract the data, we are going to examine the following features of the data: These features roughly represents the location, distribution and alignment of numbers in the image.

labels: An array containing the digits present in the array. No digit is represented by '10'.

images: This is the filename of the image. (A .png file)

tops: An array containing the distance of the digits from top. Contains 0 if no digit is present.

heights: An array containing the height of each digit. It's the height of blue box represented in above image. Contains 0 if no digit is present.

widths: Similar to **heights**, this contains the widths of digits.

lefts: Similar to **tops,** this contains the distance of digits from lefts.

Parameter	Mean	Median	Standard Deviation
tops	12.41	8.0	14.61
heights	34.36	30.0	19.37
widths	18.66	16.0	11.75
lefts	64.84	51.0	43.87

Table 1: Statistics for training dataset

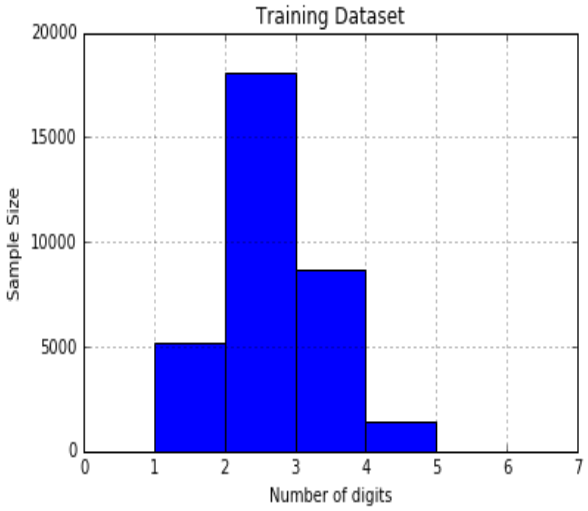
Parameter	Mean	Median	Standard Deviation
tops	64.84	51.0	43.87
heights	23.45	14.0	27.28
widths	27.89	24.0	13.45
lefts	17.25	15.0	8.78

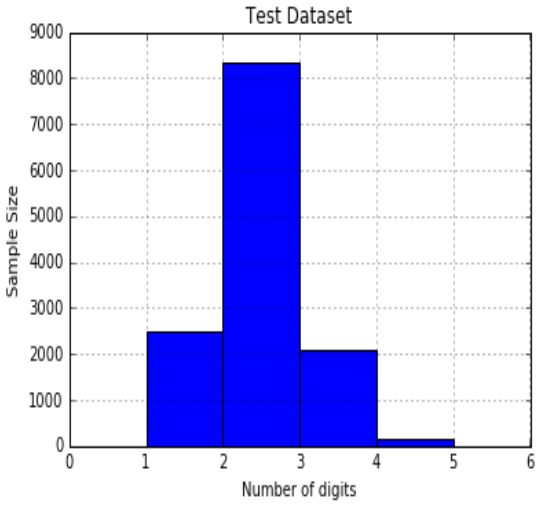
Table 2: Statistics for testing dataset

Comparing the above two tables, we can see that the data in Training dataset is more spread out than in test dataset. As a result training the model on training dataset might be a bit difficult, but it should perform good on testing dataset.

Exploratory Visualization

Now counting the number of digits each dataset has in images.





So it can be visualized that the images have numbers containing upto 5 digits. We have stored the data in arrays that are capable of storing upto 6 digits. So we will have no difficulty storing the numbers as separate digits in our dataset. For each **image**, we have an array of **labels, images, tops, heights, widths and lefts** which contain 6 entries corresponding to each digit in number of the image. Absence of a digit is represented as '10' in **labels** '0' in all other arrays.

For example, here is an image followed by the arrays in above order.

Fig 4: Sample Image

labels: [1. 4. 5. 10. 10. 10.]

images: [30. 30. 26. 0. 0. 0.]

tops: [22. 22. 22. 0. 0. 0.]

heights: [12. 11. 12. 0. 0. 0.]

lefts: [61. 74. 85. 0. 0. 0.]

Algorithms and Techniques

We will be using Convolution Neural Network for building our model. A ConvNet is made up of Layers. Every Layer has a simple API: It transforms an input 3D volume to an output 3D volume with some differentiable function that may or may not have parameters. We will take an approach similar to TensorFlows' CIFAR-10 Classification Problem. **ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture.** ConvNet is suited for our dataset as it can work directly on pixels.

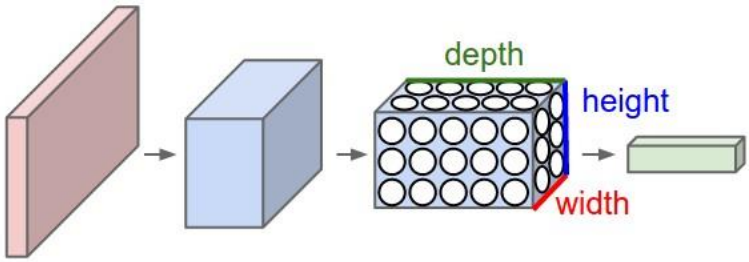


Fig 3: ConvNet

We will be training our model in layers.

Input Layer:

This layer has the image in form of pixels.

ConvNet Layer:

In this layer, each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other.

Pooling Layer:

This is a form of non-linear down-sampling. We will be using the most common **max pooling** technique which partitions the input image into a set of non-overlapping rectangles and, for each subregion, outputs the maximum.

Fully Connected Layer:

Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have full connections to all activations in the previous layer.

Output Layer:

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

Benchmark

Accuracy score is the percentage of correctly predicted data. We will set the benchmark in terms of accuracy of prediction. According to other researches, the highest accuracy that can e achieved on SVHN dataset is 97%. Considering that we'll train our model on limited number of data, we expect an accuracy of **~90%** from our model.

III. Methodology

Data Preprocessing

Performed the following steps to preprocess data:

1. Extract information from 'digitStruct' and save it in python friendly format.
2. Generate new images using the bounding boxes in previous image.
3. Resize the new image to 32x32 pixels.
4. Finally we generate the training, testing and validation data set. The validation data set is generated using the extra dataset.

Implementation

The implementation is broken down into following jupyter notebook files:

1_Extraction_and_Exploration

Here, we download and then extract the SVHN dataset. Then we explore the dataset in terms of its characteristics. We study data statistically and report mean, median and standard deviation of features of training and testing dataset. We also generate the visuals of digits count in a number over training and testing dataset.

2_Data_Preprocessing

We load the images from test, train and extra dataset. Then we remove the bounding boxes and resize the images to 32x32 pixel. We also save our dataset as pickle to use in model implementation phase.

3_Model_Implementation

Here we retrieve our dataset saved in previous phase. Then we define an accuracy function which is used to measure the performance of our model. Then we define our model. We initialize weights and biases as well as the classifiers that would be trained to recognize each digit. Then we train our model on the dataset in batches. Once the training is complete, the model is saved to used to be used for result and performance measurement.

4_Results

Here we refine our previously defined model to improve its performance. We use samples of already labelled images for prediction by our model. The predictions as well as original samples are shown in the Fig 5 and Fig 6.

Training the Model

The following steps are taken to train the model:

1. Load the saved dataset.
2. Define the Convolution Neural Network. Our convolution network looks like this:
   [Input] --> [ConvNet] --> [Pooling] --> [ConvNet] --> [Pooling] --> [ConvNet] --> [Fully Connected] --> [Output]
3. Define the weights and biases of our logits. The weights are initialized using Xavier Initialization - a Tensor Flow function that ensures that weights are balanced randomly based on number of neurons.
4. Define the accuracy and loss function.
5. Train the model along with printing accuracy and loss.
6. Save the trained model to be used again.

The Convolution Model

We have 7 layers:

- C1: Convolution layer, batch_size x 28 x 28 x 16, convolution size: 5 x 5 x 1 x 16
- S2: Sub-sampling layer, batch_size x 14 x 14 x 16
- C3: Convolution layer, batch_size x 10 x 10 x 32, convolution size: 5 x 5 x 16 x 32
- S4: Sub-sampling layer, batch_size x 5 x 5 x 32
- C5: Convolution layer, batch_size x 1 x 1 x 64, convolution size: 5 x 5 x 32 x 64
- Dropout F6: Fully connected layer, weight size 64 x 16
- Output Layer: Weight size 16 x 10

To train the model, we load the data and train in batches. While training, we try to minimize the loss and hence print the accuracy to keep a track of performance of the model. Once the training is complete, we evaluate it using test dataset.

Refinement

The initial model produced an accuracy of 81.3% with 20,000 steps. The following changes were made to improve the model's performance:

1. Added a dropout layer to the network, just before fully connected layer. This was to prevent overfitting. It works by randomly dropping weights from the model. The probability of this is set to 0.9375.
2. Also changed the learning rate to exponential decay instead of keeping constant so that it learns fast initially and then slowly over time. (Idea suggested by Udacity's Reviewer in last project)

After implementing these changes, the model predicts with 86.0% accuracy after 20,000 training steps. It should perform better with more training examples.

IV. Results

Model Evaluation and Validation

At the end of training, following results were achieved:

- Minibatch loss at step 19500: 1.344741
- Minibatch accuracy: 90.6%
- Validation accuracy: 74.6%
- Test accuracy: 86.0%

This result is close to what was expected, although it can improve.

These are the final characteristics of the model:

- Created 5 classifiers / logits.
- Implemented convolutions for depth at 16, 32, 64.
- Implemented Max Pooling to hidden layers.
- Weights are initialized using Xavier initializer.
- Implemented a learning rate decay at 0.05.
- Used Adagrad Optimizer as our optimizer.
- Introduced dropout just before the fully connected layer with 0.9375 keep probability.
- Used accuracy as our benchmark.

Justification

The highest accuracy that can be achieved is ~97% and we expected an accuracy of ~90% from our model. We achieved 86.0% accuracy just by training on 20,000 datasets. Hence we can expect the accuracy to continue to improve as we train our model on more datasets. We can expect the accuracy to reach ~95% if we train our model on whole dataset. Hence this is satisfactory result and it can be considered to have achieved our expectation.

V. Conclusion

Free-Form Visualization

In the last file "Results.ipynb", we create random images and use it for testing the performance of our model. The actual images along with labels:

Fig 5: Sample Image for Prediction

And these are the predicted results by our model:

Fig 6: Prediction

It can be visually observed that the model is satisfactorily predicting the results. It should be noted that the model is trained only on 20,000 data while the actual dataset is very large. So if we continue to train the model on the whole dataset, it will continue to improve and will reach an accuracy score of ~95%. Hence the model seems to be satisfactory.

Reflection

I selected the problem of recognizing digits from Street View House Number dataset provided by Google. But as the dataset is very large and I don't have GPU on my system, I trained on smaller portion of dataset and in batches. But I hope that if the same model is trained on more data, performance will improve. The other problem is the large variation in the height and size of images in the dataset. So I had to resize the images to 32 x 32 px to handle them.

Finally I implemented a ConvNet models as ConvNet architecture make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture.

The performance of the network is as expected.

Improvement

1. The **model can be trained on GPU** and hence it can be trained on large number of data improving the accuracy and performance of the model. Also, once we train on the whole data, we will be actually able to observe its performance and make some improvements if required.
2. The **hyperparameters can be further tuned** after training on more data to improve the performance.

References

- SVHN Dataset : http://ufldl.stanford.edu/housenumbers/
- MNIST Dataset: http://yann.lecun.com/exdb/mnist/
- Udacity Deep Learning Course: https://classroom.udacity.com/courses/ud730/
- TensorFlow CIFAR-10 Classification Problem: https://www.tensorflow.org/versions/r0.11/tutorials/deep_cnn/index.html
- Xavier Initialisation : https://www.tensorflow.org/versions/r0.8/api_docs/python/contrib.layers.html#xavier_initializer
- Adagrad Optimizer: https://www.tensorflow.org/versions/r0.11/api_docs/python/train.html#AdagradOptimizer
- Wikipedia: https://en.wikipedia.org/wiki/Convolutional_neural_network
- The Udacity Forum