Machine Learning Engineer Nanodegree

Capstone Project

Ozan Pekmezci May 18th, 2018

I. Definition

Project Overview

The aim of this project at hand is to build a software to detect house numbers on streets. The project was featured in the Deep Learning course of Udacity.

The domain is number recognition from images. The app recognizes the numbers on the image and shows it to the user. This project used Google's paper as a reference point. The paper explains Google's way to recognize multi-digit numbers from static Street View images using Deep Convolutional Neural Networks. This project also does the same using a different architecture. The best part of this project is the combination of Machine Learning with Software Engineering which are the field of interests of the author.

The project was split into three steps. The first being the digit recognition using synthetic dataset. Second one was using doing the same with real street number data and third the Android app implementation, which was optional on Udacity Deep Learning course. For the first step, MNIST dataset was used. MNIST is database that contains handwritten digits. Therefore, they are actually not the best source to detect multi-digit street numbers. That's why the digits from MNIST were concatenated to simulate house numbers on streets. The second step uses SVHN dataset, which contains house numbers images acquired from Google Street View. Lastly, third step also was supposed to SVHN dataset, but it didn't came to life during the scope of this capstone due to the reason that Tensorflow Apps never ran on the phone of the author. Therefore, the first step became just detecting one digit, second became multi digit detection with synthetic MNIST data and third multi digit detection with SVHN.

Problem Statement

The problem is the fact that house numbers have different formats. The numbers can appear with non-standard baseline, broken outlines, non-standard fonts or bad localization. The goal was recognizing all of those cases.

The strategy to solve this problem is using Convolutional Neural Networks with Tensorflow framework. The end solution would run on Android operating system to increase portability. MNIST and SVHN datasets are used to train and test data. The algorithm receives images as an input and extracts digits from them if there are any.

Metrics

The metrics are coverage, overall accuracy and per character accuracy. In the first phase of the project, we achieved 91.77% overall accuracy and 98.24% per character accuracy. For coverage, we define a confidance threshold and discard the predictions that are less likelier than the threshold. Coverage is the proportion of non-discarded values to all values. In the end, overall accuracy of 87.2%, per character accuracy of 96.8% and 96.5% coverage is reached.

II. Analysis

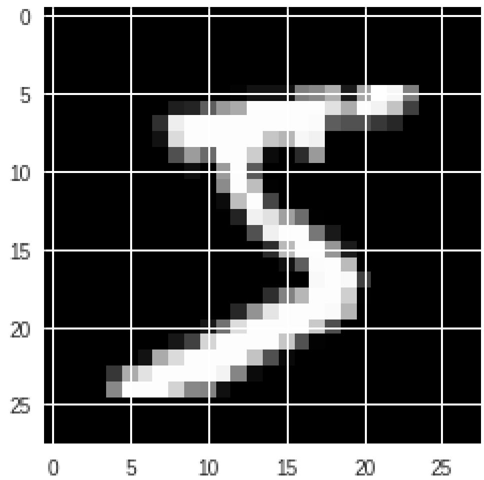
Data Exploration

The main dataset to be used for this type of a problem is the SVHN dataset, which contains Google Street View House Numbers data, however the author chose MNIST database for the beginning. The reason is simple, MNIST database provides handwritten numbers and SVHN contains sequence of digits. That's why the initial idea was concatenating MNIST characters to form an artificial dataset so that we can avoid problematic situtations that occur on house numbers like digits being upside-down, containing some lines inbetween or written in another artistic way.

According to its official website, MNIST dataset contains 60,000 training and 10,000 testing examples. All digits are normalized, centered in a fixed-size image, which makes it a good choice for machine learning since it handles pre-processing steps for already. Another reason to use MNIST initially is the fact that, it is easy to import via Keras. Keras is a machine learning frontend that serves as an abstraction layer to run different machine learning backends like Tensorflow. It also is really easy to import MNIST with the line:

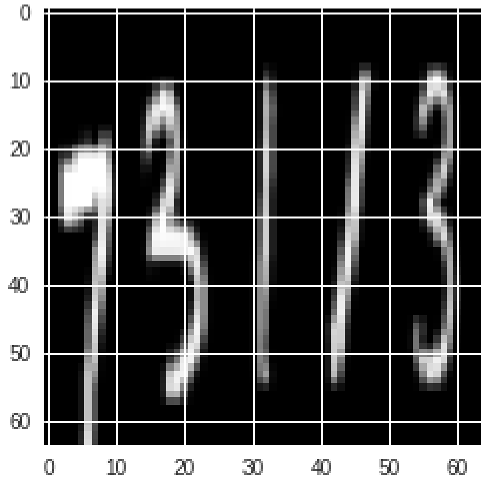
```
from keras.datasets import mnist
```

.After that, The dataset is minimal and has the size of 11 Megabytes. Each of the images are basically 28 by 28 pixels. Although they preserve color values, the software at hand transforms the images to black and white and uses them like that. This creates no problems, since different colors don't change the ability to recognize different digits on the images.



Example data point from MNIST

For the second stage of the capstone, a synthetic MNIST dataset is generated. Since 99.99% of the SVHN dataset contains house number length less than 5, the maximum length of the synthetic dataset set to be 5. This means that MNIST data points are stitched together to become data points with the length between 1 and 5. To do that, the blank character is utilized with the label 10. For example, this example below has the label (7, 3, 1, 1, 3).



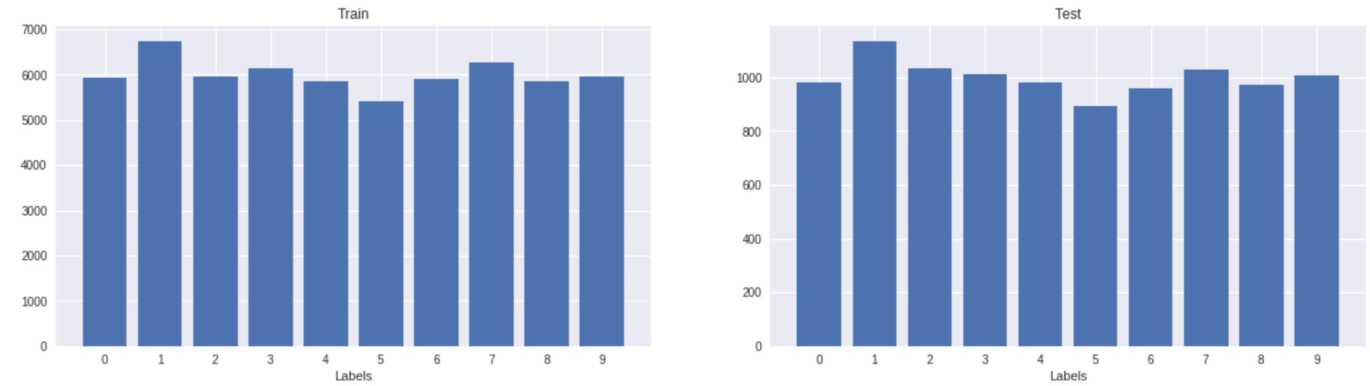
Example from multi-digit MNIST

The SVHN dataset on the other hand, is much bigger, has the size more than 200 megabytes. It contains 73257 training and 26032 testing examples. Those examples are directly extracted from Google Maps Street View, that's why all data are found in their real environment. By default, there are 10 classes, 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10. However, in our case, digit '0' has the label 0 and label 10 corresponds to the blank character. That's why a preprocessing is required. SVHN dataset comes in two different formats; format 1 contains original images with bounding boxes around characters. Format 2 has MNIST-like 32-by-32 images centered around a single character, which we used so that we can use similar model architectures for different versions. 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. However, getting and importing is as not easy as importing MNIST data since the dataset is provided in `.mat` format. The dataset should be downloaded and the data should be extracted from the dataset programatically.

Example data from SVHN

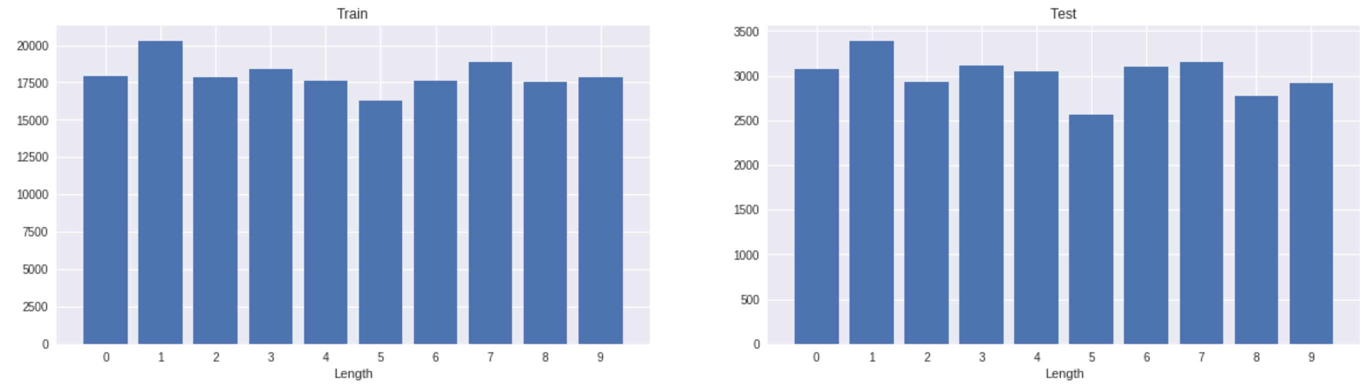
Exploratory Visualization

Amount of labels on MNIST dataset



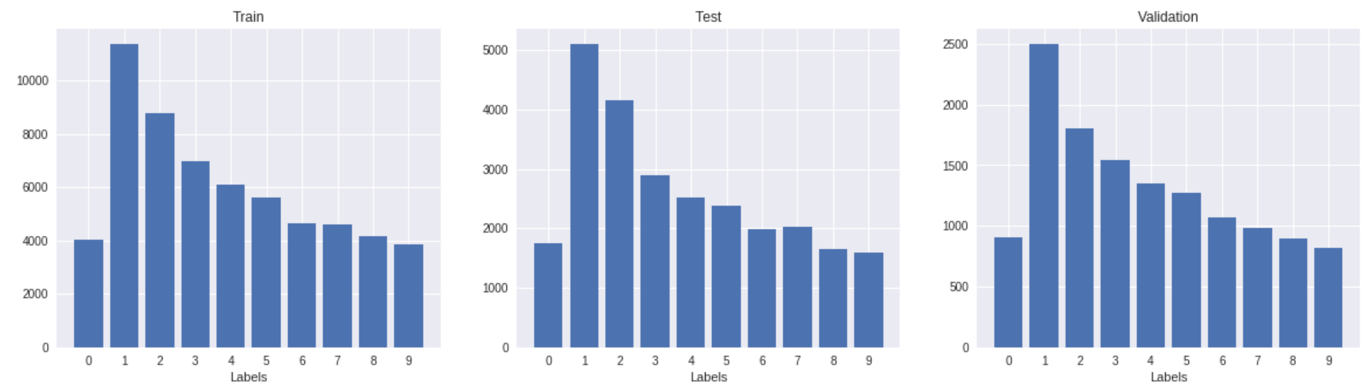
MNIST dataset contains examples of each digits in a fairly balanced way. The label 1 seems more than others and label 5 seems to be a bit less than others.

Amount of labels on Synthetic MNIST dataset



Generated MNIST dataset contains digits with the maximum length of 5. Since they were set randomly, the distribution stayed the same as on single digit MNIST.

Amount of labels on SVHN dataset



SVHN dataset looks like left-skewed bell curve that has the most examples of the label 1. The occurances of labels gets less and less starting from the label 2. Labels 0 and 9 seems to be the lowest for house numbers.

Algorithms and Techniques

For the problem at hand, the author used Convolutional Neural Networks to predict digits from the images. Convolutional Neural Networks are ideal for image recognition, since they don't flatten the nodes, which removes the logic out of images.

Another software that the author would normally use was developing an Android application but there were compilation problems that couldn't be fixed for months, which moved that part to the backlog, which will be developed after this nanodegree ends.

Benchmark

As benchmark we use the model that is specified in Google's paper. Image as input, hidden layers and an output layer that contains nodes that represent a digit each. The paper also mentions benchmark values for accuracy.

These benchmark values are coverage, overall accuracy and per character accuracy. The authors of the paper achieved 96.5% coverage, 96% overall accuracy and 97.8% per character accuracy. For coverage, we define a confidance threshold and discard the predictions that are less likelier than the threshold. Coverage is the proportion of non-discarded values to all values.

The model that was developed during the scope of this project, achieved the overall accuracy of 87.2% and per character accuracy of 96.8%. When the confidence threshold 70% was chosen, the coverage was 96.5%. However, the confidence threshold being 100% resulted the coverage being 46.7%. Therefore, algorithm is only 100% sure about the results of half of the data.

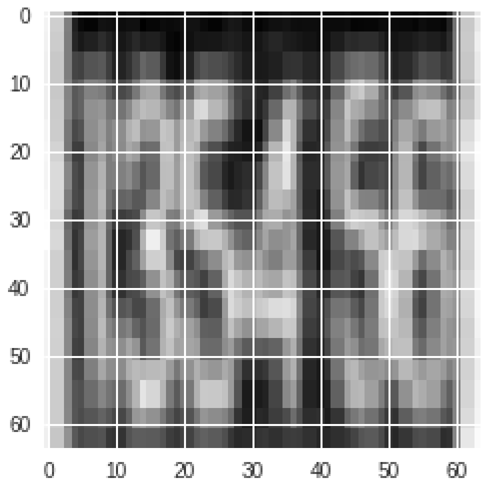
III. Methodology

Data Preprocessing

Data preprocessing is different for different steps of the project. For the first step, which is using MNIST for single digit recognition, the data is reshaped based on the image data format of the Keras instance, images are turned into gray to reduce complexity, then the RGB values values are normalised to be in the range 0 to 1 that is always beneficial for machine learning algorithms. Lastly, the labels are one-hot-encoded from class vector to binary class matrices, which again is required for machine learning algorithms to function well.

The second step is generating synthetic MNIST data and building image recognition model up to 5 consecutive digits. There, the author did what he did on the first step, plus a synthetic dataset is built. To do that, first a random length for each data is selected. Then, random indices are selected for each element. Next, the images and labels are stitched together to resemble actual data points and images. Lastly, blank images and their labels 10 are added to the required locations followed by resizing the resulting images to their right size, which is 64 to 64.

Last step was doing the same with the SVHN dataset. However, SVHN dataset is much harder to import than MNIST. That's why it needs more steps. First, the dataset gets downloaded, unpacked and extracted. We use `h5py` to import the `.mat` files, so that we can reach file contents for each digit, like the position of the boxes around digits, label and the file name. After that, we use the the information about the box to crop the parts outside to remove irrelevant sections of the images. Then the concrete training and testing data points are acquired. There are 33402 data points, labels in the training set and 13068 in the training set. Only one of them in training set has the digit length more than 5. Since we set the the maximum length to 5, this data point is removed by the algorithm.



Removed element of SVHN dataset that contains 6 digits

Another concern that the author had, was the fact that SVHN dataset only provides testing and traning sets by default. However, we also want to have a validation set to prevent overfitting and check which model doing better with which parameters. To do that, training and testing datasets are first shuffled, then 6000 of the training set are removed and saved as validation set. Lastly, we save datasets on one notebook to load them from another one.

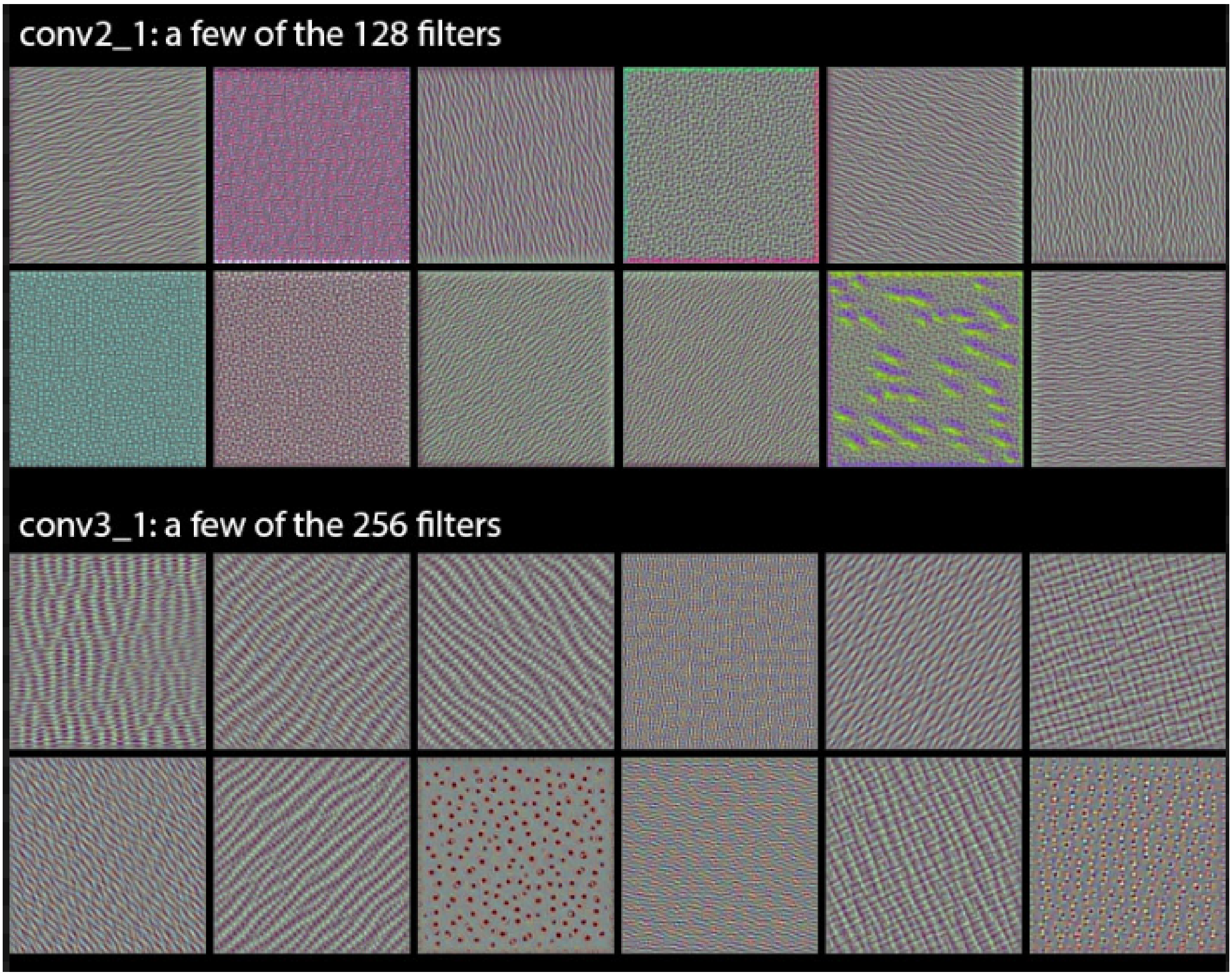
Implementation

First Phase

In the first phase of the project, only a single digit was acquired from images using the MNIST dataset. To do that, model is built after the data pre-processing is done. The code snippet below shows all of the model building process for single digit recognition.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
```

The snippet starts by ensuring that the model layers will be sequential. The first layer that we add is a 2D Convolutional Layer that filters through the image pixels and produces tensor of outputs. For the first layer, there are 32 filters that have sizes 3 by 3. The input shape is (28, 28, 1) or (1, 28, 28) regarding image data format of Keras. 28 corresponds to the size of the images, which are 28 by 28. The 1 is the color channels of the image, which are 1 for the black and white case. The activation function that is used is `ReLU`, which means `f(x) = max(0, x)`. This type of activation is common for convolutional neural networks, because its computationally easy calculation for the network.



View of the Filters, source: *https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html*

The layer is is also a convolutional layer but this time with 64 filters. Increasing the amount of filters on Convolutional Neural Networks is a technique that helps the model capturing details of the image. Typically, filter amount just gets doubled as we go deeper through the model. After that, there is max pooling layer. Max pooling is a technique that extracts the value of the biggest value on a desired pool size. In this context, the pool size is 2 by 2. Max pooling is used to ease the computation without losing much of information and prevent overfitting. We also add dropout to the output to prevent overfitting. Adding a dropout, makes sure `X` percent of the randomly selected input nodes are not used. Next, we flatten the input, meaning that the input is turned to 1D layer. Then, add another dropout with 50% this time. Lastly, we add a 1D layer with 10 units and as the activation function, softmax is chosen. In the last node, each unit correspond to a different labels, being digits from 0 to 10. Multiclass softmax is the solution to use when the topic is classification, as it outputs percentages for the data point belonging to the different classes.

When compiling the model, the loss method, optimizer and the metric is set. The loss function is categorical cross entropy, it is beneficial to use since we use classification and want to minimize the logarithmic loss. The optimizer that is used is Adadelta, which is a adaptive learning rate method. Metric we are checking is the accuracy, which should be trivial.

The snippet continues by training the model with the training set we have using batch size as 128 and 12 epochs. In the first phase, validation data and test data both are the same, which will trigger overfitting. However, this is left in the first phase as it is and solved in the next phases.

Second Phase

Second phase of the project implements detection of multiple digits that have the maximum length of 5. As it was discussed in the data preprocessing step, this phase also adds the empty character as class, since the house number can have a length between 1 to 5. Therefore, there are 11 classes instead of 10. The size of the images also increased from 28 by 28 to 64 by 64. This is a decision that is was made, since the first one was only one MNIST image and the latter one is up to 5 MNIST images stitched together. The size needed to get bigger to preserve quality.

```
# Prediction layers
c0 = Dense(nb_classes, activation='softmax')(cov2)
c1 = Dense(nb_classes, activation='softmax')(cov2)
c2 = Dense(nb_classes, activation='softmax')(cov2)
c3 = Dense(nb_classes, activation='softmax')(cov2)
c4 = Dense(nb_classes, activation='softmax')(cov2)
```

Another thing that is new, are the last layers of the neural network. Previously, the last part of the neural network consisted of one layer with 10 classes, In scope of this phase, it was changed to have 5 layers with 11 classes each. Each of those layer correspond to one digit and they are not connected to each other.

Third Phase

Third phase introduces the usage of the Street View House Number dataset instead of the synthetic dataset that was derived from the MNIST. Pre-processing data is different on this phase, which was explained in the previous sections. The main difference in the model is the number of convolutional layers. As it was mentioned before, there were only 2 convolutional layers in the previous phases. Now, there are 8 convolutional layers, which makes it better to capture details of the images.

Another step that is added is batch normalization between convolutional layers. batch normalization makes sure the input data zero mean and the variance of 1. According to Google's paper, `When training with Batch Normalization, a training example is seen in conjunction with other examples in the mini-batch, and the training network no longer producing deterministic values for a given training example. In our experiments, we found this effect to be advantageous to the generalization of the network.`[Source: https://arxiv.org/pdf/1502.03167.pdf]. Therefore, it can be said that batch normalization reduces overfitting. In this phase, the bias vectors are also removed from the convolutional layers, which is a necessity for batch normalization to properly work.

This phase also uses the advantage of using a seperate validation set. Using this decreases the testing accuracy for sure, but it also helps generalization of the model.

Early stopping is added as a last measure not to decrease the accuracy. If the validation accuracy doesn't improve after 5 epochs, training stops.

Other than the model, there is a function that calculates the indivial accuracy, global accuracy and the coverage. There are counters for each of them and individiual one gets incremented at every correct prediction, global one at every correct sequence prediction and coverage only gets incremented if the confidence of the prediction is high enough.

A last step for this phase is exporting the model as a `.protobuffer` file. To achieve that, model is frozen and saved as a constant graph so that it could be imported in the Android App, which would be able to use the model on-the-fly.

Refinement

Most of the refinement of the parameters are done in the so called third phase of this project. The third phases extracts multi digit data trained with the SVHN dataset. Initially the number of epochs was set to 12, there was no batch normalization, the optimizer method was `adadelta`, padding method was `valid` number of convolutional layers was only 2. In this initial state, the global accuracy was 44.5%, which was quite low.

First adjusments are done with the number of epochs. Increasing the number of epochs to 24 increased the accuracy to 48. Another try with the epoch number 48, increased accuracy to 49.2. Last try with epoch 192 decreased the accuracy and the training took more than two hours, so the author stuck with 48 as number of epochs.

Next variable to experiment with, was the size of batches. Using batches divides each epoch into multiple iterations with different batch sizes. Batch size determines size of those batches and indirectly determines the number of iterations in each epoch. Trying out different batch sizes from 32 to 128 only affected the accuracy by ~0.1%. That's why the author stuck with 32 as it produced relatively better results.

Another refinement was done for the padding mode. Changing it from valid to same, which was explained before, increased the accuracy 51%. A next change was experimenting with the optimizer methods. The optimizer method that produced the best result was `Adam`. Adam is a stochastic optimization method used for stochastic gradient descent, that converges better than Adadelta[Source: https://arxiv.org/abs/1412.6980v8]. After this change, the global accuracy reached 52%. The author also did some experiments with the learning rate but couldn't produce a better results.

The biggest difference was achieved by adding the regression method batch normalization that was explained in the previous subsection. With the introduction of batch normalization after each convolutional layer, the author expected to see an introduced global accuracy but it decreased to 45%. After reading and trying a lot, it was obvious that batch normalization doesn't function well if the bias vector is enabled. That's why, the bias vector is disabled on all convolutional layers. Disabling the bias vectors finally showed the effect of batch normalization and the global accuracy jumped from 52% to 77%. The reason was clear, the model could generalize better after regularization.

Lastly, the author added more convolutional layers until adding them doesn't increase the global accuracy anymore. In the end, the author have added 6 more convolutional layers, making them 8 in total and carrying the global accuracy to 87%.

IV. Results

Model Evaluation and Validation

Although the model at hand, couldn't beat Google's values of 96.5% coverage, 96% overall accuracy and 97.8% per character accuracy, it came close with overall accuracy of 87.2%, per character accuracy of 96.8% and 96.5% coverage. The model consists of many convolutional layers to capture details. It also able to generalize and predict unseen data well. Some techniques were used to minimize the overfitting and generalize the data better. These techniques included adding dropouts, batch normalization and max pooling layers. Dropouts force neural networks to learn robust features that are useful together with many different random subsets of the other neurons thus preventing overfitting. Batch normalization reduces the dependence of the network from the weight initialization, which also prevents overfitting. Using max pooling layers prevent overfitting by providing an abstracted form of the representation. The previous section also explains in detail, why specific attributes of the model was chosen. Alternatives of the parameters are tried and chosen if they did yield higher accuracy.

A correctly guessed street number

The model also worked well, when the author ran the model on a Google Colab notebook, uploaded some images and let the model predict what digits the image contains. Those data were completely seperate from the database. Pictures of images that the author drew by hand or found online. The only important thing is the fact that this solution with SVHN has a detailed pre-processing step that crops the bounding boxes in the images and resizes the whole to 64 by 64. This means that the images that are taken from other sources need to pre-processed the same way to reach the accuracy that is reached with the testing set. However, this practically not always possible, since foreign images mostly don't have bounding boxes. That's why cropped images of digits will work better than images that contain digits and also other distractions.

The algorithm thinks the author lives at house 51

Since all of the training data came in the same format, unseen data similar to that format gets predicted well. However, there are problems if the image format is not known. Therefore, the model needs to be improved with more training data in different sizes and should also contain non-cropped data for future improvement.

Justification

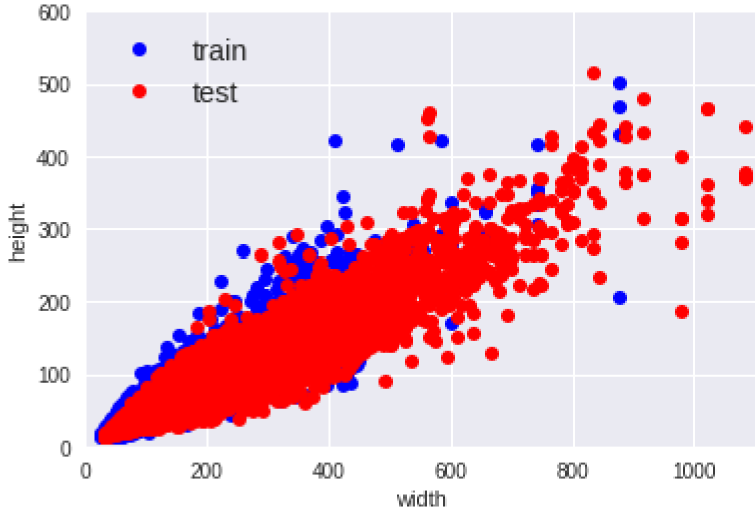
Final results that are derived by the software got really close to the benchmark results but unfortunately, they couldn't pass them. The benchmark results were achieved by Google scientists and their unlimited resources. However, the results at hand were generated by a laptop without a GPU in collaboration with free services of `Google Colab` that delete all data every 12 hours.

Benchmark results achieved 96.5% coverage, 96% overall accuracy and 97.8% per character accuracy. The author of this paper got close with overall accuracy of 87.2%, per character accuracy of 96.8% and 96.5% coverage. Like it was said before, the results with unseen images could be better with more data from different sources and different formats.

Another thing that should be mentioned is the fact that, the author couldn't finish the Android app which was optional. This would only import the `.pb` file that was exported from the model, which is a frozen version of the model that can be easily portable and usable on different architectures without having the need of a strong GPU. The author tried couple of weeks just to run basic applications but Tensorflow never functioned on the reference phone. That's why it will be developed after this nanodegree ends.

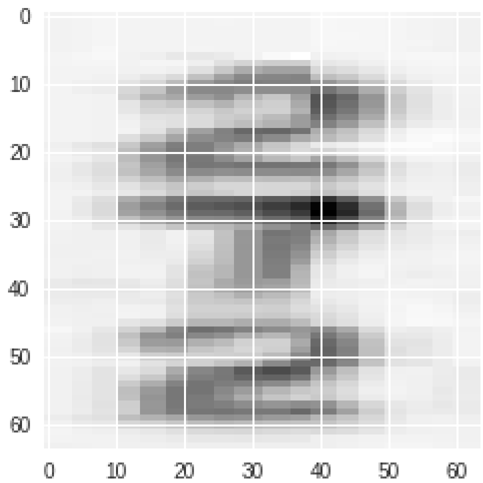
V. Conclusion

Free-Form Visualization



Height and width of the data in SVHN dataset

The image above shows the size of the data in SVHN dataset. Although the width and height increase proportionally, it is obvious that the size of individual data differ a lot. After downloading the data, everything gets resized to a fix size anyways but when this operation is done, the images also lose some characteristics, which decreases the accuracy in the end. However, some fixed size for convolutional networks are also needed since the convolutions filter data and the filters need a fixed size etc. Because of these reason it is hard to process images in general.



Black and white image of the street number 272 that was predicted as 2

There are also some data points that have marginally different format like the one above. As most of the house numbers are written horizontally, the ones that are vertical couldn't get predicted right. That's why many vertical street number data points are crucial to make sure that the algorithm can predict different formats well.

Reflection

The development process of this project have started with the easiest solution. This was detecting only one digit and used MNIST dataset. The second step was generating a synthetic dataset with MNIST that consisted of maximum 5 digits. The third phase was using the SVHN dataset with more a more improved model and again guessing digits up to the length 5.

Normally, the next optional step would exporting the model from the notebook and then importing it on the Android Tensorflow application. This way, the app would import and use that frozen model, which would be used to predict digits from the live camera image. However, this part didt't take place because problems with Tensorflow Mobile and Android.

Interesting aspects of the project was going into details of every algorithm, parameter etc. and see directly what kind of effect they have. Researching a lot before applying something and knowing how the model is going to reflect gave the author confidence.

A frustrating experience was seeing that most of the popular documents about Machine Learning are either too basic(like a tutorial) or too complex(a scientific paper). Finding some relevant sources was sometimes really hard and exhausting.

Lastly, although the solution at hand is a good starting point, it didn't fit the expectations. It is obvious that more training with different kind of data is necessary. Also an Android app would also make the experience much more interesting and better. An improved version of this, that will be built later will meet the expectations for sure.

Improvement

The first improvement to make is, training the model with more data from different sources and different format. This would increase the accuracy of the algorithm with unseen data that are in different formats. A second improvement would be regularizing the model more which will decrease overfitting like the first improvement.

Another improvement will definetely be developing the Android app. Tensorflow Mobile didn't get compiled and there were many errors that sources couldn't help. When the more improved version `Tensorflow Lite` becomes more stable, the author will definetely develop the relevant Android app, that will use the frozen model.

As it was also mentioned before, this solution couldn't beat the benchmark solution but came close. With enough research, exploration and resources it will be able to be the new benchmark.

Before submitting, ask yourself. . .

- Does the project report you’ve written follow a well-organized structure similar to that of the project template?
- Is each section (particularly **Analysis** and **Methodology**) written in a clear, concise and specific fashion? Are there any ambiguous terms or phrases that need clarification?
- Would the intended audience of your project be able to understand your analysis, methods, and results?
- Have you properly proof-read your project report to assure there are minimal grammatical and spelling mistakes?
- Are all the resources used for this project correctly cited and referenced?
- Is the code that implements your solution easily readable and properly commented?
- Does the code execute without error and produce results similar to those reported?