Capstone Project

Machine Learning Engineer Nanodegree
Build a live Camera APP.

Chomnoue Nghemning
Alain-Michel
August 12th, 2016

Definition

Project Overview

Millions of today's devices have the ability to capture images (surveillance cameras, cell-phones, drones, satellites etc…). So we can have tons of images coming from different parts of the world and covering various event types.

Machines are very good at capturing and storing images, but usually human intervention is required to describe their contents. Enabling computers to recognize objects from images will help to turn that huge amount of binary data into useful information.

Fortunately, machine learning has evolved enough so that we can train software on existing data (images with labels for example) to make it able to recognise new images without or with limited human intervention.

The purpose of this project is to use machine learning (deep learning more precisely) to build and train a computer program that will be able to read digit sequences from images. It should take an image coming from any source and output the number that it contains. It can be street or house numbers, the amount of a scanned bill, or any other picture containing numbers.

To achieve that, we will train our model against the street view house numbers. It is a real word set of more than 240 000 images and the corresponding numbers they contain, along with information on the position of theses numbers in each picture. It is available for free so we can experiment deep leaning on it.

We will then deploy that model into an android application so that the images taken from a phone camera can be analyzed and the containing numbers showed on the device screen in live.

Problem Statement

Enabling a computer program to read data from images can be hard. A solution is suggested in this paper, making usage of convolutional neural networks (CNN). They apply CNN to extract a set of features that they use to predict the sequence of digits contained in an image. Most of the solutions before them used to separate the localisation, segmentation and recognition steps. Fortunately for us, they came out with a solution integrating the three steps.

I will try to implement a deep convolutional neural network that provides the same (or maybe better) results as the ones in that paper, using the Google's machine learning library Tensorflow.

I will then try to extend it by also predicting the location of the predicted numbers on the given image.

I will start by downloading and preprocessing the street view images. I will then build and train a deep convolutional neural network that will learn how to detect the digit sequences contained in the images.

I will then try to build an architecture similar to the one in the listed paper, with the difference that instead of predicting the length of the digit sequence, my model will simply predict a “blanc” digit for the positions greater than the length of the sequence.

I will also add a parameter to limit the number of digits that the model can predict from an image. I will start with two-length digit sequences, and then extend it once I get acceptable results.

Metrics

I will use the same metric as the one in the google’s paper. That is, the model will be evaluated by its ability to predict exact digit sequences. Having exact digit sequences is very important since changing a single digit can produce a sequence that is fair from the original one.

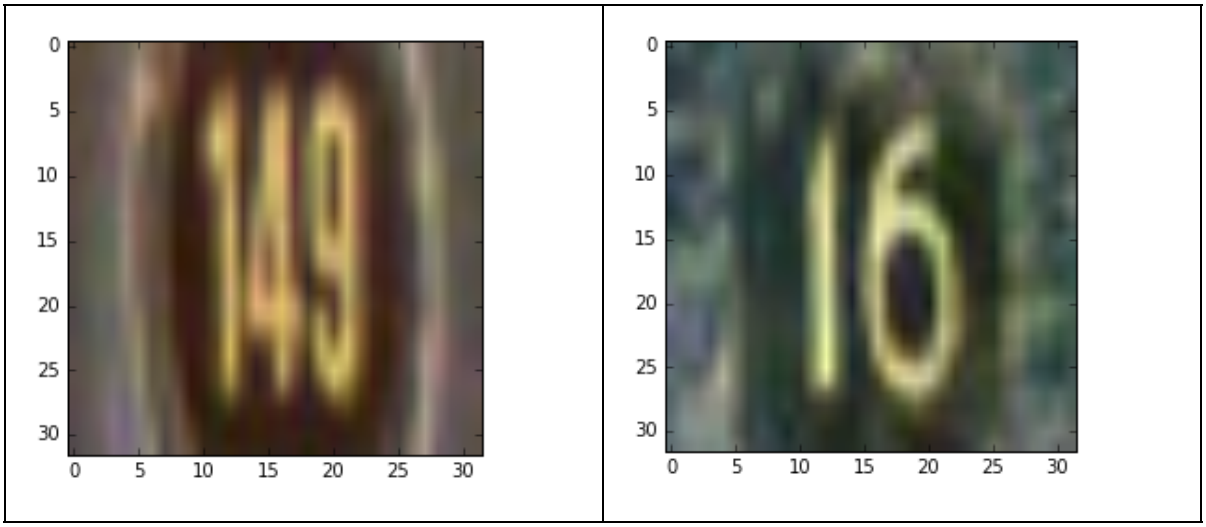
For example, 10001 and 00001 have only one different digit, but the difference between the two numbers is huge. So predicting one of them instead of the other can lead to serious issues. Thus I will not consider single-digit accuracy for this work, but only complete sequence accuracy

Analysis

Data Exploration

The data used for this work, can be downloaded on the [street view house numbers] web site.

For the version I have downloaded, it contains **248 823** images with corresponding labels (contained numbers). Following are some of the images with their corresponding labels.



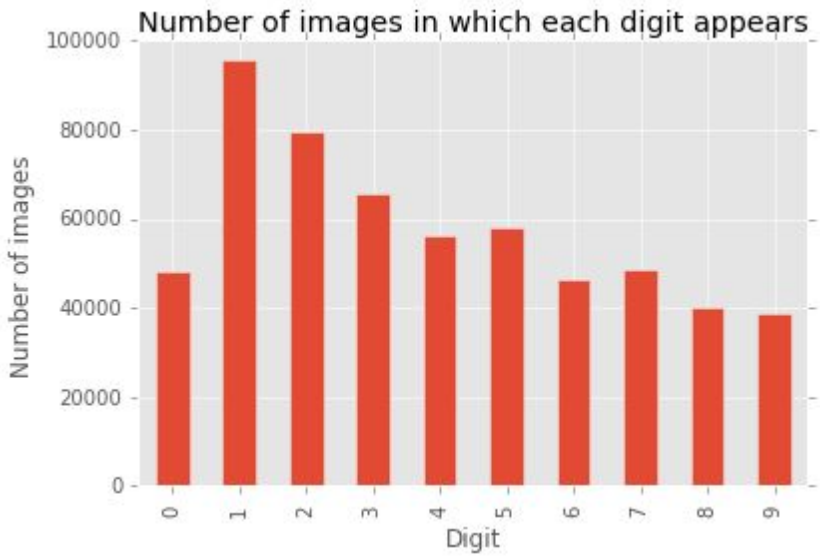
It contains a total of **630 420** digits with an average of **2.53** digits per images.
Following are, for each sequence length, the number of images showing exactly that number of digits

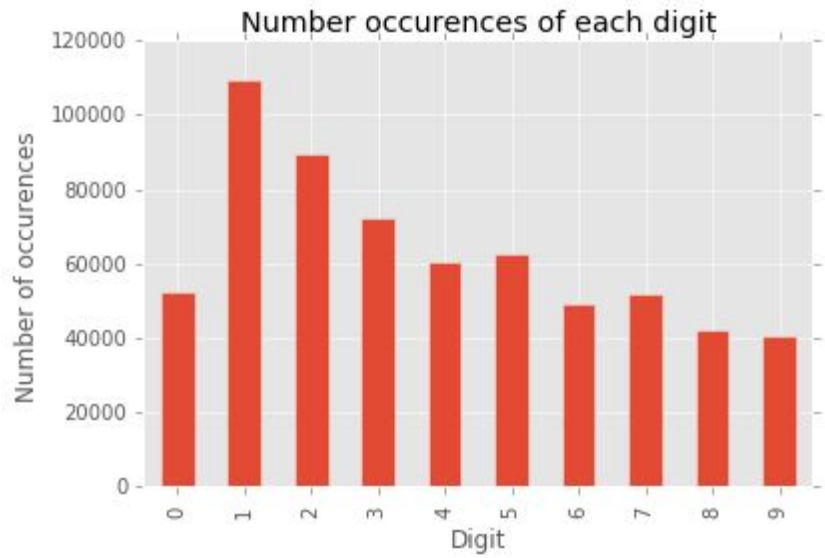
Sequence length	Number of images
1	17005
2	98212
3	117561
4	15918
5	126
6	1

You can notice that most of the images have 3 digits and there is only 1 image showing 6 digits, and 126 images with 5 digits. This means that to build a model that can predict digit sequences with 5 or more digits, we might need additional training data.

Exploratory Visualization

Following are two plots that will help us visualize how the differents digits are represented in the provided data. The first plot shows the number disting images each digit appears in while the second one displays the total number of occurrences of each digit:

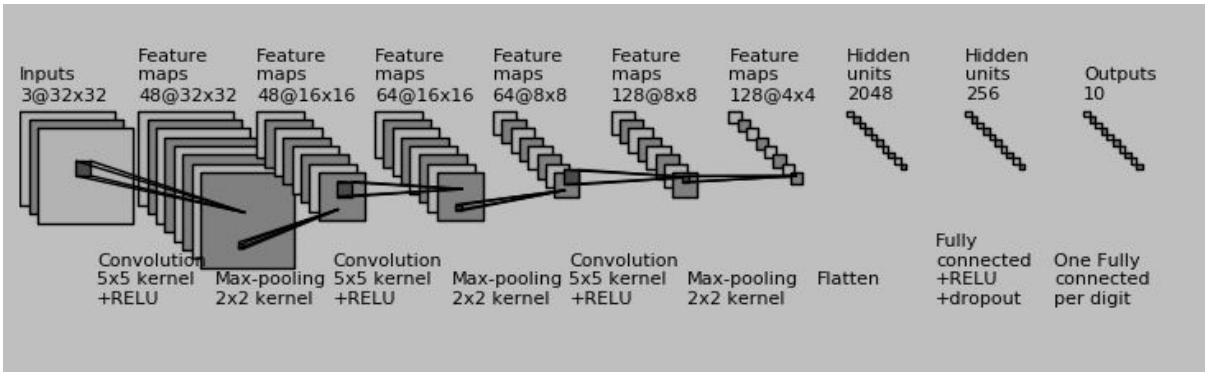




From the previous plots we can take that 1 and 2 are the dominating digits. The other digits are quite balanced, with 8 and 9 being the less represented.

Algorithms and Techniques

To solve this problem, I will try to build a convolutional neural network with the following architecture:



It consists of three **convolutional layers** of sizes 48, 64 and 128, used to convolve around the input images (or feature maps from previous layers) to extract important features.

There is a rectified linear unit (**ReLU**) after each convolutional layer, which replaces all negative features by 0, thus increasing the non-linearity properties of the model.

Each ReLU is followed by a **max pooling** layer that will reduce each 2*2 subregion of the feature map to its maximum pixel value, making the obtained feature map less sensitive to small variations in the input and also reducing the feature sizes.

The first **fully connected** unit is followed by a rectified linear unit and a **dropout** layer that randomly ignores some values from the fully connected unit by setting them to 0. The dropout is only applied in the training phase to reduce overfitting .

The last fully connected layer consists of one predictor per digit. That is, if we are predicting 6 digits per image, then we will have six distinct fully connected units at the last layer.

The inputs are images represented as 32*32*3 matrices. The labels are N*10 matrices, with N being the number of digits to predict per image. As all the images will not have N digits, I add an 11th digit, 10, to represent blanc label. For example, if an image has label [1,2] and we are trying to predict 4 digits, then we will represent that label as [1,2,10,10].

That representation of the labels leads to a sparse matrix, with 10 dominating the digits we are trying to predict. So to train the model we will use **sparse softmax entropy** instead of **softmax cross entropy**.

Benchmark

In this work we will try to achieve the same results as this work that we referenced earlier. They scored 96.03% at predicting entire digits sequences on real images of the street view house numbers data set.

Other results we could compare our work with can be found here. I will first try to achieve the 96% accuracy, and then try to beat the state of the art.

Methodology

Data Preprocessing

The house numbers images comes as .png image files. I cannot use them as this to feed the model that I am building using tensorflow. So the first data preprocessing step is to transform the images into a numpy array.

Since the images have distinct sizes, I also needed to resize them to 32*32*3 matrices. And divide the the values by 255.

I thought that the model will not need to recognize the different colors to predict the numbers, so I added RGB to grayscale transformation in the preprocessing phase. It also helped me so save some memory, since grayscale images required about the third of memory and disk space than RGB ones. However, with the poor results of the model on the test data, I switched back to RGB to see if the trained model could better generalize.

To get more training data, I first generated additional images using the method described here. It involves the following steps for each image:

1. Identify the smaller rectangular bounding box containing all the digits
2. Expand the bounding box by 30%
3. Crop random sub boxes from the bounding box

Applying that method in the preprocessing phase helped me generate more training data, but I had a hard time trying to store them in order to reuse in the training phase, always running out of memory. So I switched to the keras image preprocessing module, which enabled me to generate infinite image data during the training phase. It also has several additional features like rotation, whitening, shearing, zooming normalization and more.

The labels in the other hand come in a **digitsruct.mat** file. I failed to read such files using scipy.io library, so I needed to install h5py to extract labels. The loaded labels consists of a list of digits representing the number contained in each image. I also noticed that the 0

digit was represented by 10. So I first needed to replace 10 by 0 in the labels data before reshaping it to a 6*10 matrix, since the largest number in the dataset had 6 digits.

Implementation

The code for this project can be found here.

The first limitation I have is computation resources. For my memory to support it, I could load and work with just part of the training data (100 000 images).

As the labels were not balanced in the data set, with blank labels (represented by 10) dominating, I first tried to feed each individual digit predictor with its own batch of training data at each step. So I implemented a batch generator that, given list of digits appearing at a particular position, could produce batches of training data with balanced labels for that particular digit position. So if I had to predict two digit for example, both positions would have its own predictor as explained in the previous session, and each of the predictors would be fed by its own batch generator. That method produced very poor results, with the model rapidly converging to predict a single random digit at each step.

Feeding the models with a single batch of data and using sparse softmax cross entropy has been more efficient for me. Also, as stated earlier, I switched to the keras image preprocessing module for batch generation and data augmentation.

For the Tensorflow graph, I implemented it just as described in the Algorithms and Techniques section. Following are more implementation details:

- All the convolutional layers use **“same” padding**.
- The max pools have stride 2 and a kernel size 2
- **Adagrad Optimizer** has been used with exponential decay (learning rate: 0.05, variable global steps, decay steps: 1000 and decay rate: 0.95)

For the input data, I started with one tensor of shape None*Image size*Image size*num channels. Having None as the length of the tensor was intended to reuse it for training, validation and test data, thus avoiding to repeat code for each usage. After some difficulties and research, I have been advised to use a fixed-length tensor for each usage, as it is more memory efficient.

After passing the data through the convolutional layers, the different digits are predicted by as many fully connected units as the maximum number of digits we are trying to predict, with each predictor having its own weights and biases. The different predictions are then concatenated to produce a single tensor.

The loss is also calculated for single digit predictors. The obtained losses are then reduced to their mean, which is passed to the optimiser to be minimised.

For the training phase, a batch of 550 images is passed to the model at each step. I noticed that the keras library sometimes produced less data than expected, thus raising errors since that data was not fitting the fixed input tensors. So I added a line to verify that the data returned by the generator was of good shape before passing it to the model.

I ran the training with a train set of 100000 images, a validation set of 5000 images and a test set of 13000 images.

At the moment of writing this report, the training had been running for one day, and 65 epochs (iterations through the entire training set).

Refinement

Most of the refinements that I did were intended to reduce the gap between the training and test performances.

The first acceptable solution was performing very well on the training data (about 93% accuracy), less on the validation data (about 72%) but very bad on the test data (abut 30%). This is a major concern as our model is intended to be applied on new, previously unseen images.

The first refinement was to add a dropout layer between the different convolutional layers. This improved the performance of the model on the validation set (rising to 80%), and the test data (40%), this should still be improved.

Another refinement I made was to add local contrast normalisation *before the convolutional layers, but it didn't seemed to improve the results.*

At is time I had RGB to grayscale conversion in the preprocessing phase, so I removed that step, to see how the deep convolutional network could learn from images with three channels. Maybe some useful features are lost during the conversion. I hope to get better results at the end of the current training.

Results

Model Evaluation and Validation

As of writing this report, the training process is still running on my machine. After running for one day and 65 iterations over the training data set, the current version of the model is performing with an accuracy of 79% on the training set, 75% on the validation set and 45% on the test set.

For now I am choosing this model because the gap between the performances on the different data sets is reduced in comparison with the other parameters that I have tried.

I cannot yet decide if that model can be trusted as it is still training. I have not tested it on other input than the data from the street view house numbers site, as I am not yet satisfied by the performance on the test set.

The performances I reported above are changing over training steps, so the model has not converged yet, but I can expect some good results after several training days, since there is an overall performance improvement on the training, validation and test data.

Justification

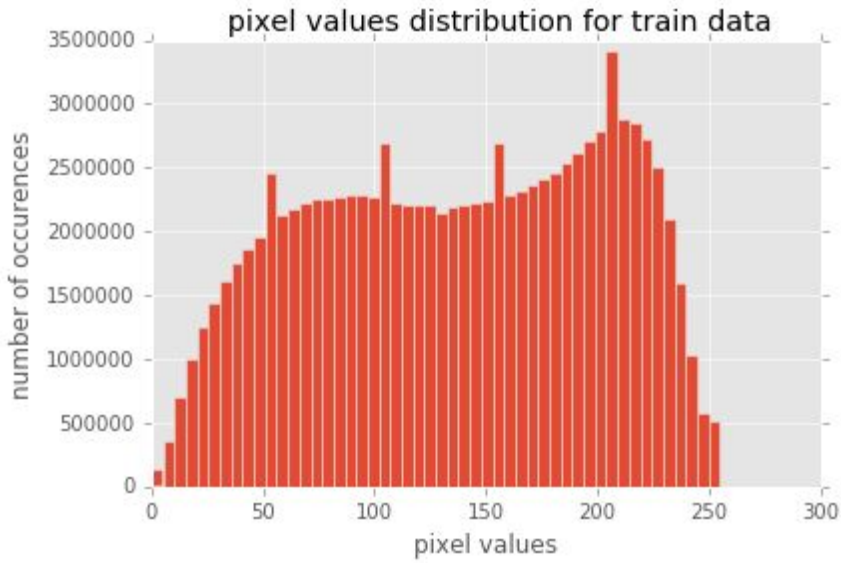
The current performance of the model is fair less from what I expect to produce on the test data, but since it is still improving, I can expect it to perform better.

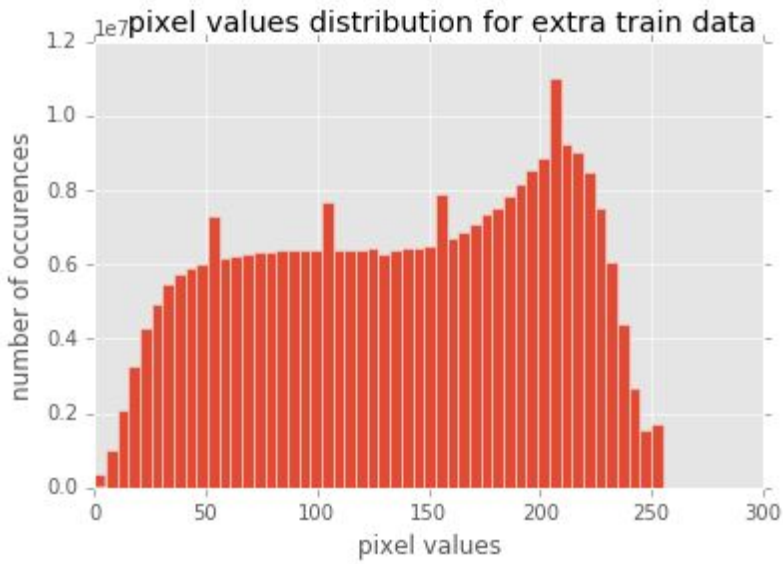
The benchmark accuracy I am targeting Is 96% on the test data so there is still work to achieve it. I am not sure yet if I should change something in my architecture or add a predictor to predict the the length of each sequence along with the digits at different positions, or maybe I should just let the model to train for at least one week as they reported here.

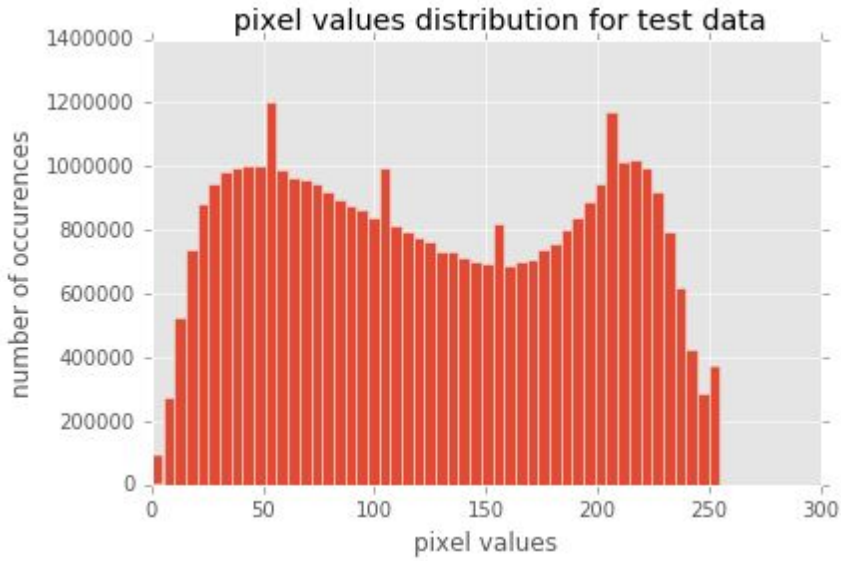
Conclusion

Free-Form Visualization

The data from street view house numbers comes in three sets: train.tar.gz with 33402 images, extra.tar.gz with 202353 images and test.tar.gz with 13068 images. The train and validation data used in this project are subsets of both train.tar.gz and extra.tar.gz, while test data comes from test.tar.gz. To understand why the model performs well both on test and validation data but not on train data, I plotted for each data set an histogram of the pixel values, and following are the resulting plots.







You can notice that train and extra histograms have similar shapes, with one peak around 210.

The test data histogram on the other has a different shape, with two peaks, one around 50 and the other around 210, and one valley around 160.

This means that the pixels value distribution is different from train/validation and test sets, and maybe it is why the model is not generalizing on test data.

Finding a way to have a similar distribution for training and test data, maybe after the preprocessing phase, might be the key to obtain a better generalizing model.

Reflection

In this report, I went through the process of building a classification model for the street view house numbers. The proposed solution starts by some data preprocessing, involving data acquisition and transformation into numpy arrays, easier to use to feed machine learning algorithms. We also described a method to augment train data during the train phase.

We then described the convolutional neural network architecture and how we trained it to predict the numbers appearing in the images.

The model (still training) already performs well on train and validation data, but there is a concern about how it performs on test data, with about half of the validation accuracy, so I will not use it on real world images before some improvements.

The main difficult thing I experienced is the time it needs to train the model. You should let if run for at least one night before you can guess that some refinements are needed. It is reported that it can take one week for the model to converge. Unfortunately I don't have a dedicated computer for that.

Something interesting is that I have been able to show, using a visualization, how the pixel values distribution is different between train images and test ones. I am very excited to do some research to understand what the different distributions mean and if I can overcome that to improve my model performances on test data.

Improvement

The obtained results are fair from my expectation, there is still more to do to improve. I will do the following:

- Review the code to make sure that there is not a bug that makes test accuracy so bad
- Try to obtain test and train data with similar distribution of pixel values after the preprocessing phase, and see how it can improve the model.
- Try to build an architecture with the same parameters as described here and see if I can obtain at least their results, before trying to improve it.
- Try to train with different values of dropout, to see If I find one that makes the model perform good on the training data as well as on test data.
- I am not sure that the keras image preprocessing module fits well with this problem since it does not take into account the position of the digits on the images, so it can produce new images with digits out of shape. I should try to implement my own image generator using the process described here, but generating images as stream during the train phase instead of trying to store them.

Once I get the expected results, I will also try to train a classifier to predict the position and the bounding boxes around the images, so that the number can be framed in live on a camera screen.