Deep Learning Udacity Capstone of SVHN Project

Project Overview

Deep Learning is now state of the art for most of the perception problems like image analysis problems and text analysis problems. Deep learning shines when there is a lot of data and complex problems. In this project, I have trained a model that can decode sequences of digits from real images. I have implemented the project in a simple Python script using Tensor-flow deep learning library.

Problem Statement

The aim of this project was to understand the work done in the paper [1] that is how the researchers tried to solve the problem of localization, segmentation and recognition using only one model instead of separate models. Therefore same dataset which is Street View House Number (SVHN) dataset will be used with the same goal that is recognizing the multi-digit images from real world images. The convolutional neural network architecture that I have built is much shallower than the paper [1] due to computational limitations. Since less number of convolutional layer will be used I absolutely do not think that the similar results will be achieved which is given in the paper [1].

Metrics

I am focusing on two metrics. One is taking the average of the prediction accuracies of the digits in the sequence. Let's say even if one of the digit in the image is misclassified but remaining four digits are classified correctly then accuracy score in predicting that image will be 4/5 instead of 0. Second metric is measuring accuracy level of the sequence of the labels for example if the actual label look like “0”,”5”,”3”,”10”,”10” and the predicted label by the classifier is “0”,”4”,”3”,”10”,”10” then according to the third metric the accuracy score is zero because the there is/are digit(s) misclassified in the sequence.

The loss function that is to be minimized is softmax cross entropy function. Softmax is a neural transfer function that is generalized form of logistic function implemented in the output layer that turns the vectors z_j into the probabilities that add up and constraint to 1.

$$y_j = \frac{\exp\left(z_j\right)}{\sum_{i=1}^{k} \exp\left(z_i\right)}, where\, z_j = \sum_{i=1}^{k} x_i * w_{i,j} + b$$

Training the model is about finding weights and biases (parameters) such that they become good at prediction that is *y*. *y* can be seen as logits which means that it is score of each class label. Softmax is applied to turn this score into probabilities for being each class label that can sum up to one. After that one-hot encoding is applied in which the probability with highest value in softmax probabilities is turned into one and the rest of the probabilities are turned into zero so that we can compare with the actual class labels in one hot encoding form to get the accuracy level of the prediction performance. Cross Entropy is the cost measure to compare the actual class labels and the predicted class labels. It is also seen as the distance measure between two vectors. Formula of cross entropy is as follows.

$$\frac{-1}{n} \sum_{i=1}^{n} \log\left(f\left(x_i, l_i, W\right)\right)$$

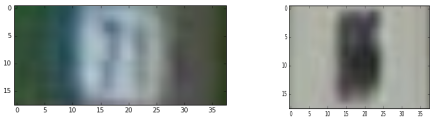
where the f function is the model's predicted probability for the input $x_i's$ label to be l_i, W are its parameters, and n is the training-batch size

In the case of softmax cross entropy with logits loss function, x inputs are replaced by y logits.

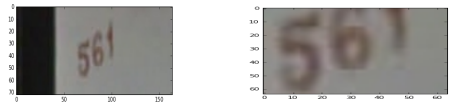
The objective of the training the classifier that is to get the parameters such that the distance between the correct predicted class label and the actual class label is low but the distance between the incorrect class label and actual class label is high. This can be averaged as a loss function. The machine learning problem can be turned into numerical optimization problem where gradient descent is applied to the loss function to minimize with respect to the derivatives of parameters like weights and biases.

Data Exploration

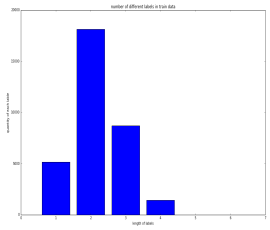
SVHN dataset has been download from [2] which is used for building the convolutional neural network model for digit classification. In the training data folder there are 33401 images which are used for training the model. In the test folder there are 13066 images however the 1000 images were tested. All these images have three color channels that is RGB. The length and width scales are varying in the dataset. There are various issues in the image data. For example there are blurry pictures like this ones below where it is hard for even human eye to recognize the digits. Actually there are images originally have small scale when they are resize to bigger scale they got blurry.



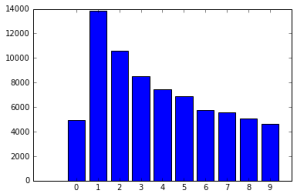
Below picture is translational which means that digits are not horizontal. When bounding box is applied to those images then edges of those second digit and so may not appear in the resulting cropped images. In order to solve this problem bounding box co-ordinates have been adjusted so that digits are captured properly in the images. However if the digits are vertically aligned then this approach may not work and so the vertical digits in the images will be problematic in this project.



The following is the frequency distribution of the length of the digits in the image.



Images with 2 digits are the most frequent in the SVHN dataset. And there less image with five or four digit sequences.



This is the frequency distribution of the digits in the dataset. Digit '1' has the most frequency with 14,000 number of occurrence while digit '0' has least frequency of occurrence with around 5000.

Data Pre-processing

The images have 3 color channels that is RGB. So in the prep-processing steps, images have been converted to one grayscale by changing the number of the channels to one. In this way training will be less computationally intensive. In order to change the RGB channel into the grayscale channel, Skimage has been used to change the number of channels and along with that image pixel values have also been rescaled into [0 1].

The way to get the greyscale image from RBG image is changing the pixel values according to the equation below, where R represents red, G represents green and B represents blue.

Y = 0.2125 R + 0.7154 G + 0.0721 B

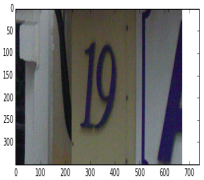
RGB image

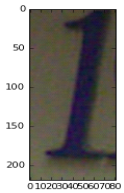
Grayscale Image

Having a grayscale image will have less computational cost. There has also been rescaling of the arrays from 0 to 1.

Images are resized to 64 by 64 image size as they were of varying length and width.

In digitStruct.mat information about the labels and bounding boxes is given. The matlab data file is extracted by python's h5py module. This module is used to extract the data into python which can be given in any format. Bounding boxes are used to crop the images. Below is the full image and the cropped image using the bounding box of the first digit.

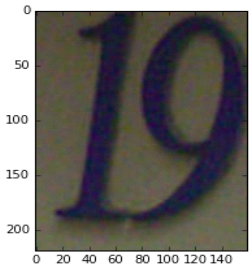




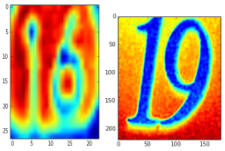
By using bounding boxes of each digit cropped images can be obtained. In each bounding box, co-ordinates like height, top, width, left have been in the digitStruct.mat data file.

After increasing the width given in the bounding box, the cropped images have been combined.

Below is the example image.



The above image will be used as an input to the deep learning classifier for the recognition of multi-digit images. Multi-digits are also converted from RGB to grayscale and rescaled within the range of [0,1]. Below are some the pictures in the grayscale channel.



Matplotlib library was used to view the images that's why we can see grayscale like in the above.

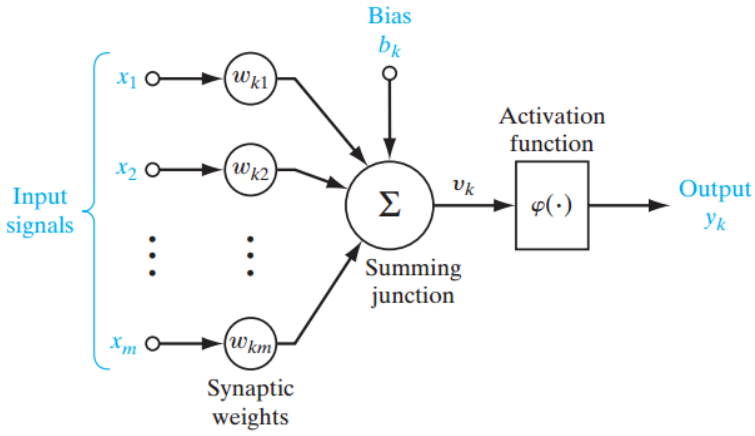
Labels were one-hot encoded. '10' character in the labels in the image data was changed into '0'. And '10' was inserted as the blank space as we have to get a five sequence of labels for each image. Let's say image of 19 has the following labels; ['1','9','10','10','10'].

Algorithms

I will start off by explaining feed-forward multi-layer perceptron which is considered to be the building block of all the neural networks techniques like recurrent neural nets and convolution neural nets.

A multilayer perceptron (MLP) has one or more hidden layers along with the input and output layers, each layer contains several neurons that interconnect with each other by weight links. The number of neurons in the input layer will be the number of attributes in the dataset, neurons in the output layer will be the number of classes given in the dataset.

Below is the diagram of the architecture of Multilayer Perceptron



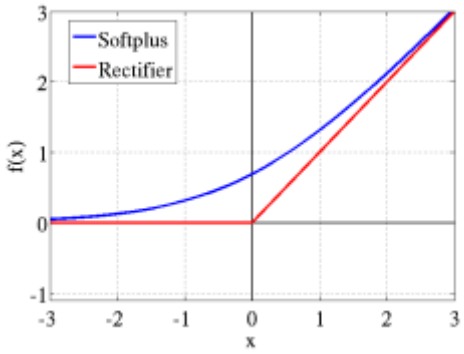
In the figure, W_{km} represents the m weights that are seen as a set of synapses or connecting links

between one layer and another layer within the network. This parameter indicates how important each feature x_j is.

Initialisation of the parameters, weights and biases, plays an important role in determining the final model. There is lot of literature on initialisation strategy. A good random initialisation strategy can avoid getting stuck at local minima. Local minima problem is when the network gets stuck in the error surface and does not go down while training although there is still capacity for learning. The initialisation strategy should be selected according to the activation function used. In this project weights are initialised by using normal distribution. In order to ensure numerical stability weights should follow standardised normal distribution with mean zero and variance of 1 which is what have been implemented in the project.

The activation function defines the output of a neuron in terms of the induced local field v. $y_k = \varphi(u_k + b_k)$ where $\varphi(.)$ is the activation function.

There are various types of activation functions, the rectified linear units (RELUs) have been very popular lately. ReLUs are the smooth approximation to the the sum of many logistic units and produce sparse activity vectors. Below is the diagram of the function;



(softplus $f(x) = \log(1 + e^x)$ is the smooth approximation to the rectifier)

In neural networks the data flows forward from input layer to output layer via hidden layer(s). In order to train the network data is back-propagated from output layer to input layer by taking derivatives of the parameter using chain rule. The back-propagation algorithm can be used to train neural networks. It is a method to minimise the cost function by changing weights and biases in the network. To learn and make better predictions a number of epochs (training cycles) are executed where the error determined by the cost function is backward propagated by gradient descent until a sufficiently small error is achieved. The activation function takes wX + b function and squashes into the non-linear form.

if wX + b > 0 then wX+b

wX + b <= 0 then 0.

We can add another layer of RELUs (or any other activation function) on top of previous layer into the model. The output from the previous RELU (hidden layer 1) will then be an input to the added layer (hidden layer 2) and computation takes place in a similar non-linear manner. If there are 2 such layers then our model is under the realm of deep learning.

Making the model scalable is possible by Stochastic Gradient Descent (SGD). In SGD instead of model computing the loss through all the training examples at once on each iteration (epoch), the model computes the estimate of loss of average of random sample of the data (this is also called

mini-batching). Each sample of the data updates the weights and biases from previous sample of the data's updates of the weights and biases at each training cycle (epoch). The downside is that there is noise in the loss function's optimizer. Exponential learning rate decay of weights enables the implementation of SGD successful. At each training step (epoch), the learning rate goes down exponentially. Learning rate regulates the value of the change in the weights during training.

Let's say in 100-sized mini-batch, 100 training examples are shown to the learning algorithm, and then weights are updated. After all mini-batches are presented sequentially, then average of the accuracy levels and training cost levels are calculated for each epoch.
Learning rate controls the change in the weight from one iteration to another. As a general rule smaller learning rates are considered as stable but cause slower learning. On the other hand higher learning rates can be unstable causing oscillations and numerical errors in the error surface.

$$\begin{pmatrix} Weight \\ correction \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} learning-rate\ parameter \\ \eta \end{pmatrix} * \begin{pmatrix} local \\ gradient \\ \delta_j(n) \end{pmatrix} * \begin{pmatrix} input\ signal \\ of\ neuron\ j, \\ y_i(n) \end{pmatrix}$$

Momentum provides inertia to escape local minima; the idea is to simply add a certain fraction of the previous weight update to the current one, helping to avoid becoming stuck in local minima.

$$\Delta w_{ij}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$, where α is the momentum

Dropouts reduce over-fitting by being equivalent to training an exponential number of models that share weights. There exists an exponential number of different dropout configurations for a given training iteration, so a different model is almost certainly trained every time. At test time, the average of all models is used, which acts as a powerful ensemble method. In this project dropout rate of 0.5 have been applied at the final fully connected layer. However I am not really sure if the weight in the model were averaged at the test time in the Tensorflow implementation.

Today one of the most important factors for the increased success of deep learning techniques is advancement in the computing power. Graphical Processing Units (GPU) and cloud computing are crucial for applying deep learning to many problems. Cloud computing allows clustering of computers and on demand processing that helps to reduce the computation time by parallelising the training of the neural network. GPUs, on the other hand, are special purpose chips for high performance mathematical calculations, speeding up the computation of matrices.

In machine learning overfitting is a major issue. Neural Networks with large parameter space are prone to over-fitting i.e. the network performs well on the training set but not on the test set. Regularization is one of the popular ways to deal with over-fitting. L2-norm regularizers are added in the loss function and regulates/penalizes the large update of the value of weights.

Convolutional Neural Networks are the most popular algorithms used for images. Image input can be seen as the function of 3 dimensions that is height, width and depth. Depth with value of 1 represents grayscale. Depth with value of 3 usually represents RGB. Convolutional neural network allows us to take advantage of local connectivity in the data. Convnet accepts the image data in the form of 3D space. There are filters in the convolutional layer, these filters convolve over the image i.e. slide over the image spatially, computing dot products. Let's say if there are six filters, there will be six separate activation maps that can be stacked together to get a new image. Due to the sliding filters and local connectivity we can share the parameters and so end up with lesser number of parameters than the feed forward neural networks. Hence convolution neural networks are much more efficient computationally.

Strides are number of slides that the kernel can slide over the image and that are how weights are shared together. Having a large number of strides may result in greater information loss. Instead

pooling is applied, max-pooling is the most common one. Max-pooling applies max-operation on the information extracted by kernel on each stride. Finally fully layered network is connected at the top of the convolutional architecture and in this way a classifier is constructed.

Implementation

Tensor-flow is used for the implementation in this project. The entire purpose of TensorFlow is to have a so-called computational graph that can be executed much more efficiently than if the same calculations were to be performed directly in Python. TensorFlow can be more efficient than NumPy because TensorFlow knows the entire computation graph that must be executed, while NumPy only knows the computation of a single mathematical operation at a time.

Tensor-Flow can also automatically calculate the gradients that are needed to optimize the variables of the graph so as to make the model perform better. This is because the graph is a combination of simple mathematical expressions so the gradient of the entire graph can be calculated using the chain-rule for derivatives.

Placeholder variables serve as the input to the TensorFlow computational graph that we may change each time we execute the graph. Input in this project are the image pixel intensity values.

The configuration of the architecture has been developed by the approach used in [1]. In the traditional approaches there are separate models for the localization, segmentation and recognition steps. In [1] unified approach has been taken successfully that is all these steps are integrated by deep convolution neural network that operates directly on image pixels. 97.84% accuracy level has been reported on the Street View House Numbers in [1]. DistBelief infrastructure of Google has been used for the experimentation.

Mini-batch gradient descent of batch size with 16 has been used. There are optimizer function that are used for loss minimization. The input image is processed in the first convolutional layer using the filter-weights. In the convolutional layer the filter is being moved to different positions of the image. For each position of the filter, the dot-product is being calculated between the filter and the image pixels under the filter, which results in a single pixel in the output image. So moving the filter across the entire input image results in a new image being generated. When the filter reaches the end of the right-side as well as the bottom of the input image, then it can be padded with zeroes (“same”). This causes the output image to be of the exact same dimension as the input image. Furthermore, the output of the convolution is passed through a Rectified Linear Unit (ReLU), which merely ensures that the output is positive because negative values are set to zero. The output may also be down-sampled by so-called max-pooling, which considers small windows of 2x2 pixels and only keeps the largest of those pixels. This halves the resolution of the input image. And then again the set of convolution, RELU and max-pooling layers are stacked that processes the output of the first set and so on.

The final fully connected layer in the convolutional neural network extracts the set of features from images. Only five softmax classifiers have been used for classifying the labels, sixth softmax classifiers for classifying the length of the sequences of the digits have not been used. Five softmax classifiers are connected with the final dense layer of the convolutional neural network and those classifiers have separate weights and biases.

The main challenges in the implementation I had were that CPU resources were not enough for testing the whole test data at the same time I performed mini-batching to the test data in a similar that was done on the training data by using feed dictionary of the batch data and labels to the placeholder variables. Another challenge was that as I scaled up the number of layers in the

network, it was getting difficult to handle large lines of Python codes.

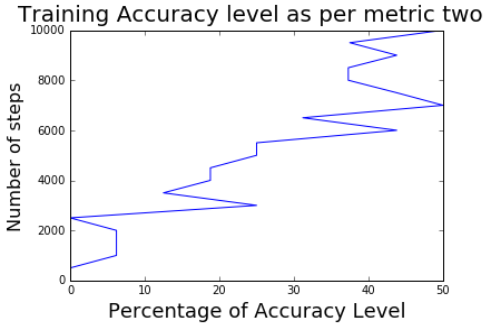
Results

Google's research team [1] used eight convolutional neural networks along with three fully connected layers which is very compute instensive as their model ran for 6 days on big GPUs to be able to get over 96% accuracy level. Although I will not attempt to achieve the similar results due to computational constraints, I will try to replicate their work as much feasible as it can be for me.

Initially I built two convolution layer with one fully connected layer and I was not getting good results. I saw that loss function value was propagating between 7 and 5 over the 5500 epochs which means that the network is unable to learn with the 2 convolution layers and 1 dense layer. I tuned with hyper-parameters like depth size, patch size, number of hidden nodes but still results were disappointing. Then I switched the optimiser of gradient descent accompanied by momentum to adagrad optimiser. That improved the test accuracy (first metric) from 60% to 84% of test accuracy as per metric one. Adagrad is change of learning rate value with respect to the parameters. If the parameters are sparse then the learning rate increase and vice versa. It turns out that changing the optimization function brought the drastic improvement. Changing the patch-size from 5 to 3 decreased the computational time by halve but it also lowered the accuracy level. Therefore I kept the patch-size of 5 for most of the experiments. I also saw that training for longer can also bring improvement in the results, in one of experiments it brought 0.5% improvement in the test accuracy.

The initial architecture that got 84% of accuracy level as per metric number one and 43% of accuracy level as per metric number two is as follows. Two convolution layers with strides of one, patch size of five and depth size of 16, along with rectifier linear units layers, and max-pooling layers of 2 by 2 blocks are used and then fully connected layer of rectified linear unit activation function with number of hidden nodes of 64 is used and then five softmax classifiers are used for classification. Adagrad optimization function with 0.05 learning rate is used. In each batch there are 16 input images. The model is trained for 10,000 number of steps.

It turns out that two convolution layers with learning rate of 0.05, depth 16, number of hidden nodes of 32, patch size of 5 with test accuracy level of 47.1% as per metric two is better than the three convolution layers with the hyper-parameters values having 44% test accuracy level as per metric two. Hence the two convolution layers learning rate of 0.05, depth 16, number of hidden nodes of 32 has been chosen as final model for this project. Below is the training accuracy level as per metric two over the 10,000 number of steps.



Below is the hyper-parameters experimental results.

	Patch size changed from 5 to 3 with 10000 number of steps	Patch size changed from 5 to 3 with 15000 number of steps	hidden nodes changed from 64 to	learning rate changed from 0.05 to 0.01	3 layer convolutions
Accuracy level as per metric one	75.50%	76.00%	83.40%	72.00%	82.50%
Accuracy level as per metric two	-	-	47.10%	15.80%	44.1.%
Computational time (in mins)	15	23	13.5	14	15

Reflections

This is really interesting project that gives hands on experience in solving simple real world problem of digit recognition. It also made me realize that deep learning technique is really computationally heavy. Since I didn't have access to the GPUs or AWS Cloud Computing, it was difficult for to work on deeper networks and get a good accuracy of numbers in the images. There was major issue of compute resources that limited me to work further on this project.

References:

[1] Goodfellow Ian J, Bulatov Yaroslav, Ibarz Julian, Arnold Sacha, Shet Vinay. Multi-digit Number
Recognition from Street View Imagery using Deep Convolutional Neural Networks
Google Inc.

[2] (http://ufldl.stanford.edu/housenumbers/)

[3] Neural Networks and Learning Machines (3rd Edition) by Simon O. Haykin

[4] Improving neural networks by preventing co-adaptation of feature detectors by Geoff Hinton

[5] https://github.com/Hvass-Labs/TensorFlow-Tutorials/blob/master/02_Convolutional_Neural_Network.ipynb