Street View House Numbers (SVHN)

Capstone Project - Machine Learning Engineer Nanodegree

Amruta Kulkarni Nov 06, 2016

Definition

Project Overview

The Computer Vision[1] problem of identifying numbers within real-world images has been a fairly difficult problem to solve for a long time. But the advent of more sophisticated learning techniques in the area of Artificial Neural Networks and Machine Learning in general have made considerable progress in the last few years to provide great results, and the image recognition accuracy rates by computers have become closer to, or in some instances are at, human levels.

In this project, we will build a multi-layer Convolutional Neural Network (ConvNet)[2] - a Deep Learning network to recognize house numbers from the Google street view house numbers (SVHN)[3] 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[4] (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data and comes from a significantly harder, unsolved, real world problem of recognizing digits and numbers in natural scene images.

Problem Statement

Some of the intricacies that contribute in making it harder to correctly recognize house numbers within a real-world image are: orientation, location, fonts, sizes, spacing, color, resolution etc of the constituent digits as shown in samples here:

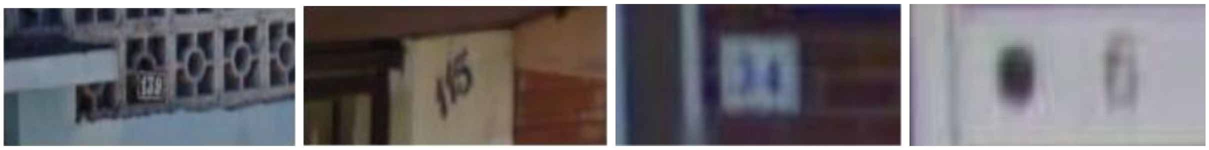


Fig 1. Shows some hard to recognize sample images.

The solution in this project using ConvNets has been demonstrated to work on relatively inexpensive infrastructure such as a Quad-core processor, 8GB memory and using various open source software libraries of Python and Tensorflow[5].

We shall witness that even the fairly complex problem of correctly identifying house numbers from the SVHN dataset, which is not an entirely curated dataset, can yield an accuracy rate of ~95% using a ConvNet architecture of 7 layers: 3 conv layers + 2 max pooling + 1 dropout + 1 fully connected layer before the output.

[1] https://en.wikipedia.org/wiki/Computer_vision
[2] https://en.wikipedia.org/wiki/Convolutional_neural_network
[3] http://ufldl.stanford.edu/housenumbers/
[4] http://yann.lecun.com/exdb/mnist/
[5] https://www.tensorflow.org/

Metrics

For the problem of correctly identifying digits in images, the metric we will use is: the summation of how many digits are correctly recognized in each image and this is calculated over the entire dataset.

The formula is:

$$\frac{\Sigma(\frac{Correctly\ classified\ digits}{\#\ of\ digits\ in\ number})}{Size\ of\ dataset}$$

Formula 1. Accuracy Metric

Analysis

Data Exploration

SVHN dataset consists of 3 datasets: Train, Test and Extra. Train dataset contains 33402 entries, Test 13068 and Extra over 200K. Each dataset also contains a metadata file digitStruct.mat with info about every image filename along with each digit's boundary box and label contained in that image. For instance, *digitStruct(100).bbox(2).width* contains width of the 2nd digit bounding box in the 100th image. Similarly height, label, top, left info is contained in the digitStruct for each image.

Here's sample statistics for all datasets:

	Mean Height	**Max Height**	**Min Height**	**Mean Width**	**Max Width**	**Min Width**
Train	57.21	501	12	128.29	876	25
Test	71.57	516	13	172.58	1083	31
Extra	60.80	415	13	100.38	668	22

Table 1. SVHN dataset sample stats.

Of all the images in 3 datasets, we will only work with images containing 1 - 5 digits, since there are enough samples to train the model for those classes and others would be considered outliers (explained further in data processing below). So in essence, we would be working with numbers in range 1 - 99999 inclusive.

I have added about 50K samples from the Extra dataset to Train, so over 83K samples were used for training. Also we would be using the terms Extra and Validation interchangeably in this report, since about 10K samples from Extra were used for validation during model training.

SVHN provided images are in various flavors i.e. color - RGB (3) channels, resolutions, sizes & orientation etc. Some samples are:

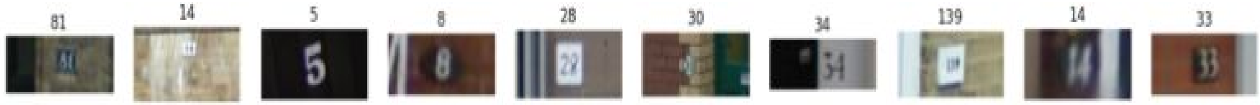


Fig 2. SVHN sample images - show various resolution, size, color, orientation etc.

All images were resized to 32x32 and converted to grayscale, as those details (size/color) are not important or relevant in correctly classifying the digits inside of images. This is further explained in the data processing section below.

Exploratory Visualization

Fig 3 below shows distribution for number of digits or class lengths within each dataset. It can be observed that all 3 datasets are fairly Gaussian/Normal distributions. The distribution of class lengths across datasets is also proportionate enough for us to train, validate and test results.

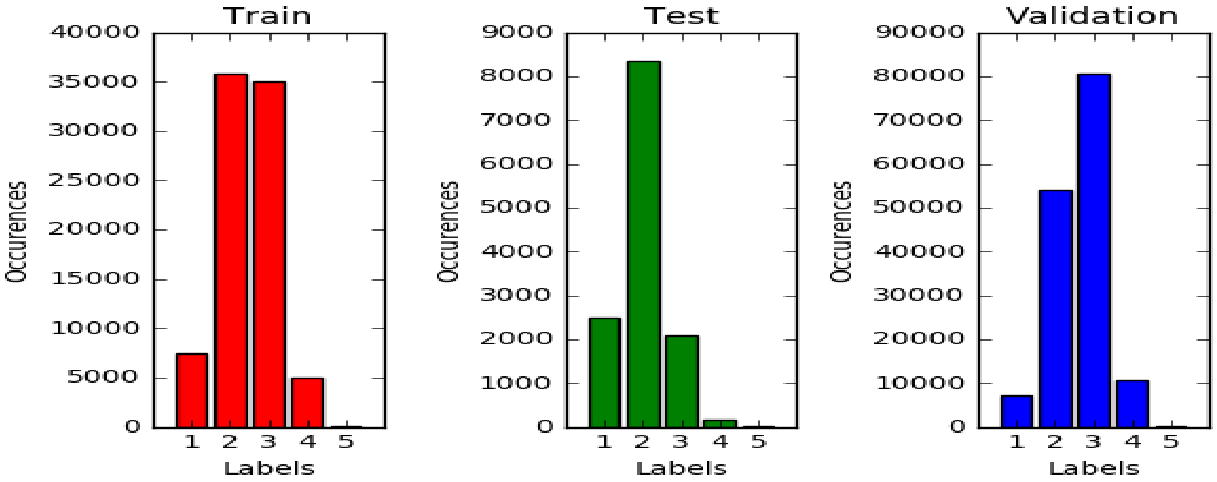


Fig 3. SVHN distribution of class lengths.

As seen in the graphs above, 90% of Train dataset is constituted by images with class length 2 & 3. About 80% of Test dataset is composed of Images with class length 2 and class lengths 2 & 3 constitute the majority of Extra/Validation data.

Fig 4 below shows the distribution of individual digits in the 3 datasets. Digits 1-3 have the highest frequencies in all 3, followed by digit 4, 5, 0 and so on.

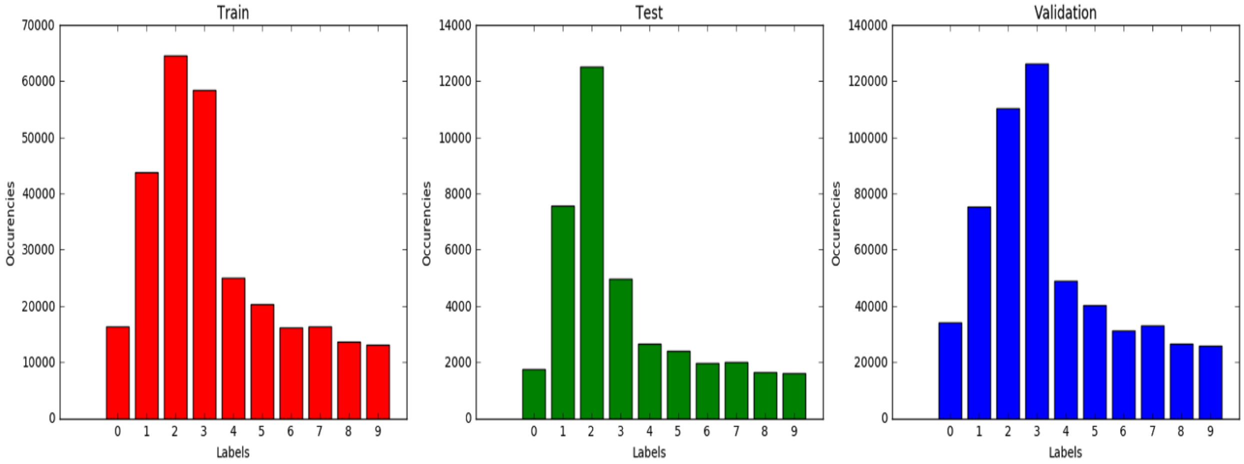


Fig 4. SVHN distribution of digits in Train, Test and Validation datasets.

Since about 50K more samples from Validation were added to Train, the dataset size is adequate for training - since typically the model learns/trains better with more data.

Algorithms and Techniques

We will utilize the widely popular technique of Convolutional Neural Networks (CNNs / ConvNets) to identify digits within images of the SVHN dataset.

ConvNets are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases. 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. And they have a loss function (e.g. Softmax/SVM) on the fully-connected/last layer and all the tips/tricks in learning regular Neural Networks are mostly applicable. A ConvNet is made up of Layers, hence it is a form of Deep Learning network. 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/weights. As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. The final output layer by the end of the ConvNet architecture would reduce the full image into a single vector of class scores, arranged along the depth dimension.

The difference between regular Neural Networks and ConvNets is that the latter architectures make the explicit assumption that the inputs are images, which allows to encode certain properties into the architecture. These then make the feed forward function more efficient to implement and vastly reduce the amount of parameters in the network[6] and thus the save the computations in learning these parameters.

ConvNets take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth - forming a 3D volumes of neurons.

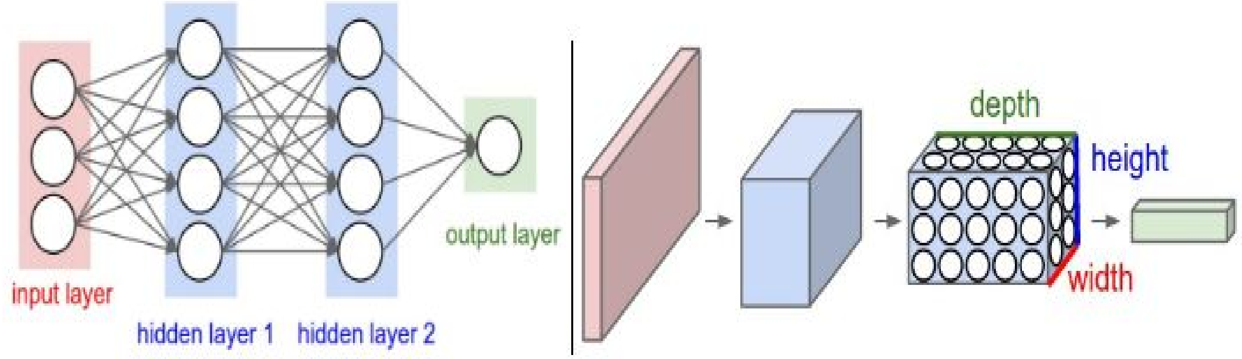


Fig 5. Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions: width, height, depth (visualized in one of the layers).

[6] http://cs231n.github.io/convolutional-networks/

Three main types of layers are used to build ConvNet architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer (exactly as seen in regular Neural Networks). These main 3 essential layers are stacked to form a full ConvNet architecture. Other layers discussed below often aid in learning the model better, but may or may not be useful depending upon the problem statement and ConvNet architecture. In our solution to SVHN image recognition, all layers discussed herein were found useful.

Our ConvNets architecture contains the following component layers:

INPUT layer holds the raw pixel values of the image, in this case an image of width 32, height 32, and with 1 grayscale color channel [32x32x1].

CONV layer is the core building block of a Convolutional Network that does most of the computational heavy lifting. The conv layer parameters consists of a set of learnable filters. And the network learns which filters to activate depending upon type of visual feature input. Computationally, a Conv layer computes the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume.

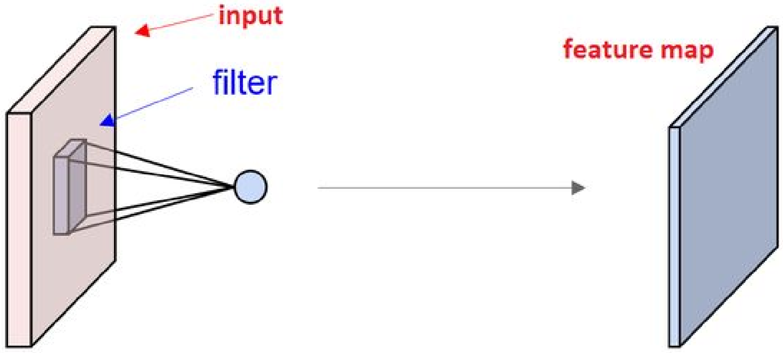


Fig 6. Shows Input and ConvNet Filter layers.

RELU layer will apply an elementwise activation function, such as the max(0,x) thresholding at zero. This leaves the size of the input volume unchanged.

POOLING layer progressively reduces the spatial size of the representation in terms of the amount of parameters and computation in the network, and hence also controls overfitting. The Pooling layer operates independently on every depth slice of the input and resizes it spatially, by using some function. In our application MAX operation will be used, as it is the most common one.

Figure 7 below illustrates Input->Conv->MaxPool layers and a result of MAX function with 2x2 filter and stride = 2.

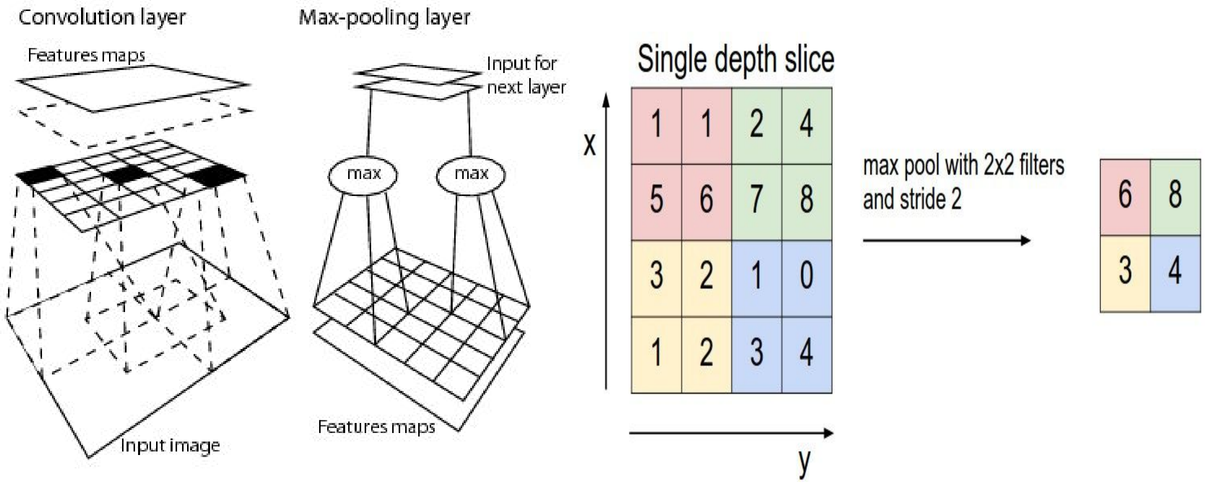


Fig 7. Left: Input->Conv->MaxPool layers.
Right: Illustrates result of MAX function with 2x2 filter and stride = 2.

FC (fully-connected) layer computes the class scores and has full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

LOSS layer specifies how the network training penalizes the deviation between the predicted and true labels and is normally the last layer in the network. Various loss functions appropriate for different tasks may be used there. We will use Softmax loss function for predicting a single class of K mutually exclusive classes.

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function without parameters. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

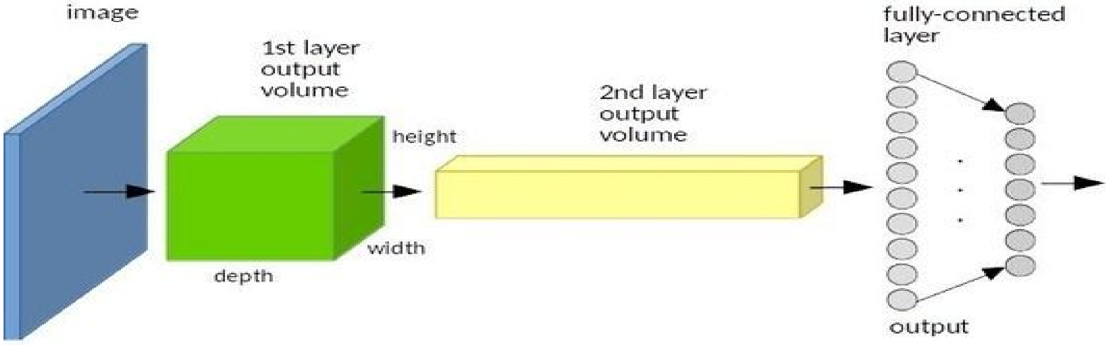


Fig 8. A simplified ConvNet architecture for illustration.

The ConvNet architecture used to learn the model in our project utilizes all the components layers discussed above. Once the basic version is designed and is working, we conduct a hyperparameter search trying to obtain the best possible accuracy and minimal errors.

Other ancillary layers such as Local Response Normalization and image preprocessing steps viz Local Contrast Normalization were used and found to be very helpful in terms of the model accuracy.

Benchmark

As discussed in the Problem Statement section above, the SVHN is a real-world dataset containing images of various resolution, sizes, orientation etc that are significantly harder compared to some of the other publicly available datasets for image recognition. Given the nature of SVHN and the ConvNets, we will target an accuracy rate of at least 90% as the benchmark - which has been demonstrated in public forums to be reasonably achievable.

Methodology

Data Preprocessing

The following image preprocessing steps are implemented to make the data more manageable and remove any unnecessary noise from the images in all 3 datasets.

1. The images/digits are cropped near the edges based on their BBox details. With this operation, the data with signal is retained and noise is eliminated to begin with.
2. Then images are resized to 32x32 from their original varied sizes.
3. Next they are converted to grayscale using a known technique[7] - since the color is not a factor/signal in recognizing the digits. In general, we need to multiply red channel by a factor of 0.2989, green by 0.5870 and a blue by 0.1140.

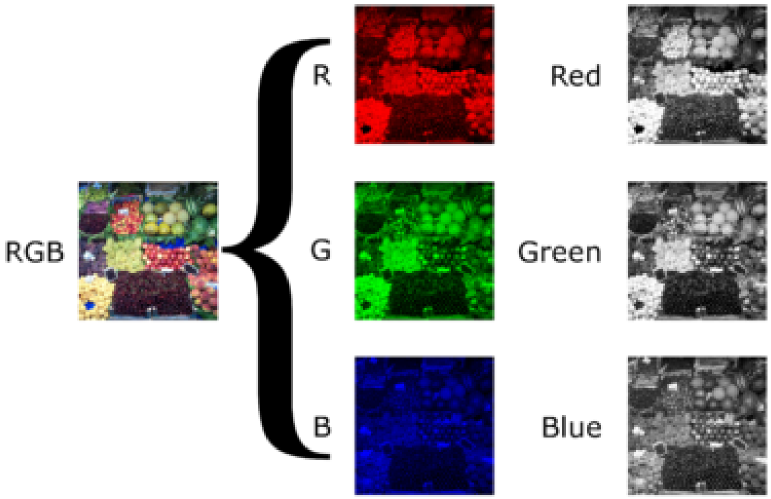


Fig 9. Sample image grayscale conversion illustration.

4. Then images are mean-normalized i.e. mean value of each dataset is subtracted from each image and is divided by the standard deviation of the dataset.
5. Later during model training, images are processed using Local Contrast Normalization (LCN)[8] technique before being fed into the network for training/testing. With our ConvNet architecture, the LCN operation resulted in very good results in terms of model accuracy.
6. Then each sample class is split by digit. So if a number has less than 5 digits, label 10 is added at the end e.g. if the image contains labels 823 then subclasses represented as ['8','2','3','10','10'].

[7] http://www.eyemaginary.com/Rendering/TurnColorsGray.pdf
[8] http://yann.lecun.com/exdb/publis/pdf/jarrett-iccv-09.pdf

7. Class ‘0’ is replaced from ‘10’ to ‘0’ during preprocessing to be consistent with rest of the classes and class ‘10’ itself is used to identify the non existence of a digit. So altogether there are 11 classes: 0 - 9 and 10.

After all images from the 3 datasets are preprocessed, the resultant image datasets are stored in a single .pickle file. Python “pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream is converted back into an object hierarchy. This allows for us to store the results preprocessed images to disk and recover the same later, and not having to redo all preprocessing over again every time we need to work with the datasets used with ConvNets.

The pickle file thus created was less than 1GB when stored on the disk, which is comparatively lesser than the actual raw dataset sizes combined for the same number of images.

Implementation

As discussed earlier, our solution described above uses ConvNets, which have been shown to outperform other approaches in image recognition.[9]

In our design, a 7 layer ConvNet produced an accuracy rate of close to 95%:

Input: batch_size x 32 x 32 x 1 (batch_size = 64)
C1: Conv layer with filter size 5 x 5 x 1 x 32
P2: Max pooling layer
C3: Conv layer with filter size 5 x 5 x 32 x 64
P4: Max pooling layer
C5: Conv layer with filter size 5 x 5 x 64 x 512
D6: Dropout
F7: Fully connected layer, weight size: 64 x 11

The ConvNet has a set of subclasses of size N = 5 to separately recognize each digit in a number of maximum length N. On the output, we receive a set of logits of size N which are passed to the softmax function to get probabilities of a digit belonging to some particular class.

[9] https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

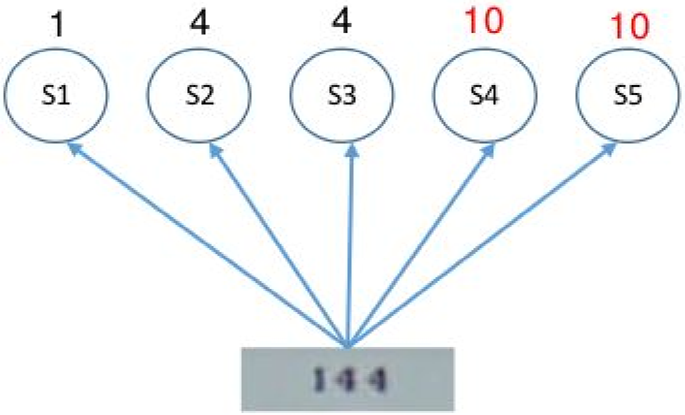


Fig 10. Number recognition on the output layer.

We use the same architecture to recognize all 5 digits in any image. Figure 11 illustrates architecture of the ConvNet layers used for recognizing digits (S1, ... , S5) in an image.

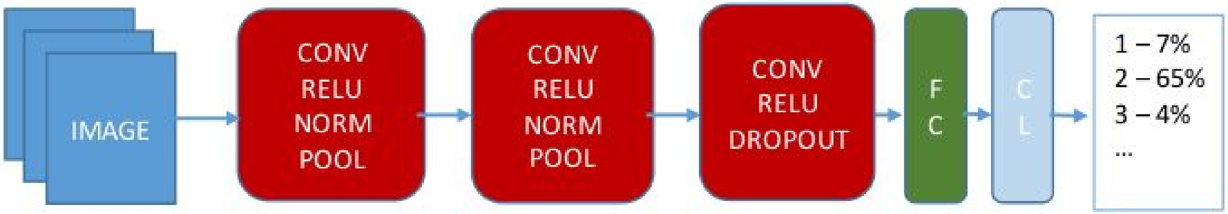


Fig 11. Digit recognition ConvNet.

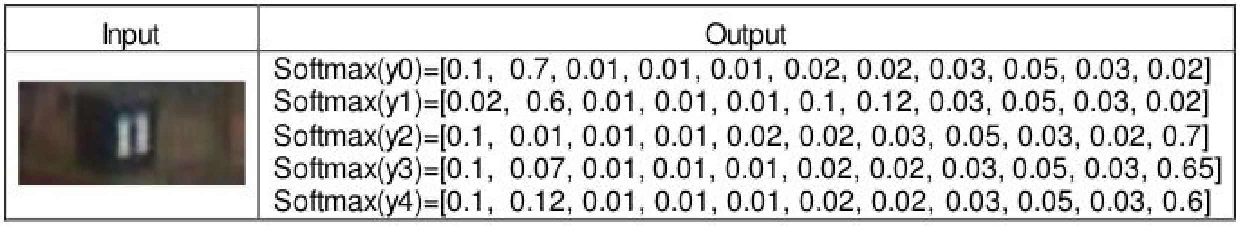
To start with, the ConvNet receives 32 x 32 images of batch size 64. The 1st Conv layer applied has size of 5 x 5 x 1 x 32. The 2nd Conv layer has size of 5 x 5 x 32 x 64. And the 3rd (last one) has a size of 5 x 5 x 64 x 512.

On the output of 1st, 2nd and 3rd convolutional layers, data is sent to a ReLU layer which does not change input image dimension, but applies element wise activation function MAX(X, 0) thresholding at zero. Then after the 1st and 2nd ReLU layers, the data is sent to a Local Response Normalization layer. Although this kind of normalization layer has been discussed as one which has a minimal practical impact, it showed very good results for our solution.

Towards the end, before the data is sent to a single fully-connected layer, dropout with keep probability 60% is applied and the data is reshaped into a 2D matrix to feed it to the Fully Connected (FC) layer. The FC layer being the last has weight 64 x 11. The results thus received are passed to a classification layer which applies softmax function and produces probability of each digit belonging to some class.

During training, we use Stochastic Gradient Descent (SGD), which is a simple yet very efficient approach to discriminative learning. It makes use of a learning rate parameter to determine how fast it needs to converge. High learning rate helps converge faster, but typically has higher error rates, while a low learning rate translates to learning slower, but produces more accurate results.

On the output of our trained model we have 5 vectors (S1, ... , S5) of size 11. Each entry contains a probability of a digit belonging to some particular class, i-th vector contains probability distribution of i-th digit in a number. If an image has a number length less than 5 digits, the vectors for the non-existent digit would have the highest probability for class 10.



Input	Output
	Softmax(y0)=[0.1, 0.7, 0.01, 0.01, 0.01, 0.02, 0.02, 0.03, 0.05, 0.03, 0.02] Softmax(y1)=[0.02, 0.6, 0.01, 0.01, 0.01, 0.1, 0.12, 0.03, 0.05, 0.03, 0.02] Softmax(y2)=[0.1, 0.01, 0.01, 0.01, 0.02, 0.02, 0.03, 0.05, 0.03, 0.02, 0.7] Softmax(y3)=[0.1, 0.07, 0.01, 0.01, 0.01, 0.02, 0.02, 0.03, 0.05, 0.03, 0.65] Softmax(y4)=[0.1, 0.12, 0.01, 0.01, 0.01, 0.02, 0.02, 0.03, 0.05, 0.03, 0.6]

Fig 12. Illustrates example of input and softmax output.

Then an array Z of size 5 is produced with output of the function: $Z[i] = argmax(softmax(y_i))$

By using example provided in Figure 12, the array would be: Z = [1,1,10,10,10]. Finally, all elements from Z are concatenated except the 10s (class represents non existent digit). Thus the output number itself is predicted/derived.

The model described above recognizes numbers from SVHN images with accuracy ~95%. Figure 13 shows pictures of actual vs predicted numbers using our ConvNet.

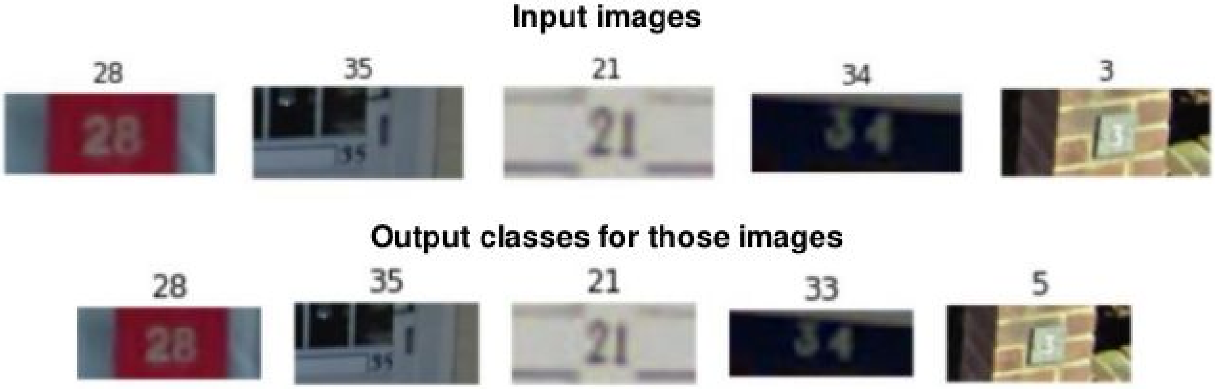


Fig 13. Sample predicted images using our ConvNet.

Refinement

Various combinations of different techniques were tried during the development of our project, but not all were useful. Discussed here are those which improved overall accuracy. Firstly, the Local Response Normalization layer helped train the model better in terms of producing better accuracy, although it is said to be fallen out of favor. But it turned out to be particularly helpful in our case.

The next parameter tried was the dropout rate. Of the dropout rates tried, a dropout of 60% keep probability produced the best results. With a dropout rate of 50%, test accuracy was at about 90%, while with the dropout rate 60%, the test accuracy is at about 95%. Figure 14 illustrates minimization of loss value with dropout 0.5(left) and 0.6(right).

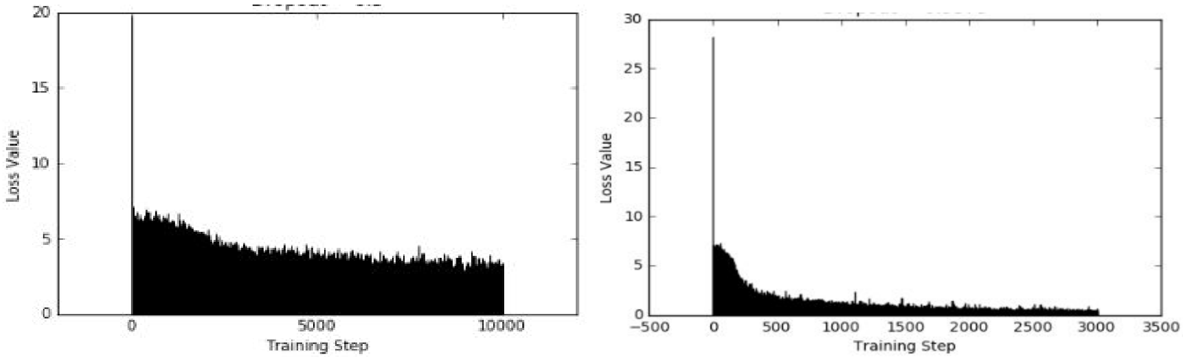


Fig 14. Left: 50% Dropout. Right: 60% Dropout (multiply x-scale with factor of 10).

Results

Model Evaluation and Validation

Figure 15 shows the accuracy results we obtained for Train vs Validation during the course of our ConvNet model learning.

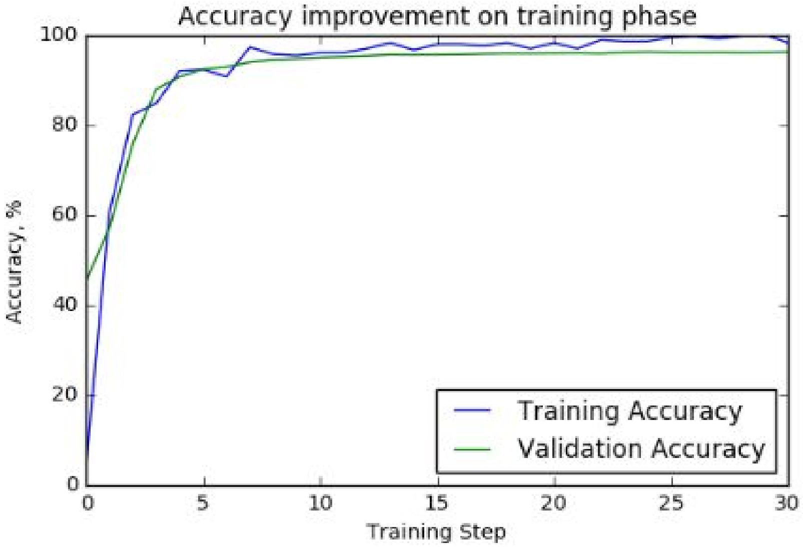


Fig 15. Learning Curve: Training vs Validation accuracy progression for our ConvNet.

As we can see in Fig 15, the model is not overfitting, and if it were, the graph would have resembled the blue line Fig 16 below[10].

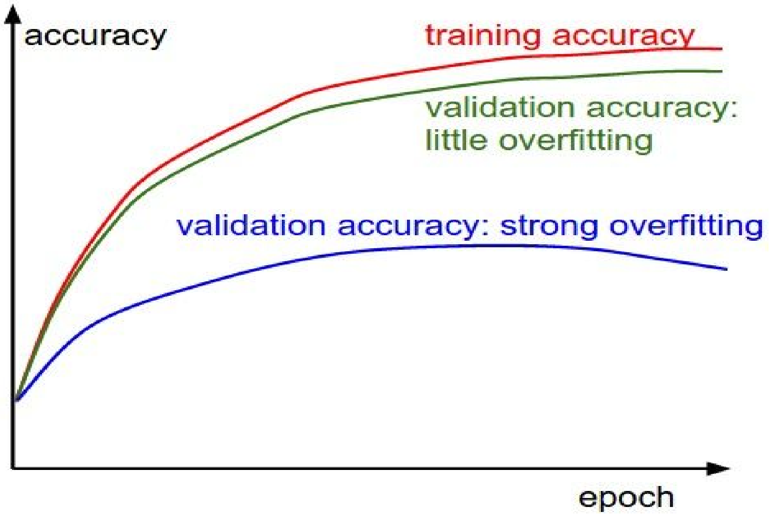


Fig 16. Learning Curve: Training vs Validation Accuracy - Shows Overfitting (blue line).

Also comparing the Train, Validation and Test accuracy values shown in Table 2 below gives an idea that the model obtained is not underfitting, since the accuracy values across the 3 datasets are fairly close - which is typically not a symptom of underfitting (or overfitting).

	Train	**Validation**	**Test**
Accuracy %	**98.1**	**96.1**	**94.6**

Table 2. Accuracy values of Train, Validation and Test.

Justification

Our ConvNet model produced an accuracy rate of ~95%, which is much above the established benchmark of 90% discussed in the Benchmark section above. Also as discussed in earlier sections, the SVHN dataset contains images that are fairly hard to recognize, even for humans. Considering all these details, we shall accept the results thus obtained.

[10] http://cs231n.github.io/neural-networks-3/#accuracy

Conclusion

Free-Form Visualization

Figure 17 below shows the actual vs predicted image numbers by our ConvNet model on the SVHN data. The data/images were chosen randomly for display.

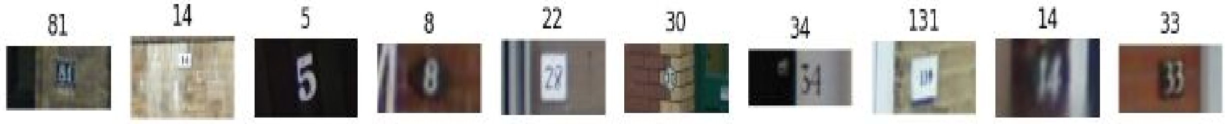


Fig 17. Actual vs ConvNet Predicted Digits.

As shown with the above images, the computer is able to “see” the numbers contained in images with the help of the ConvNet model we have learned. Also it can be observed that the model is predicting the numbers correctly, as in examples 3(5), 4(8), 7(34) etc, where the images are clear and can be seen with bare eyes. This is not the case in examples 1(81), 2(14), 6(30) etc, where it is fairly difficult even for us humans to see the numbers & we can only resolve to guessing these numbers - which is perhaps even the computer is doing (based on the softmax results)!

Reflection

During development of this project, I observed that with the ConvNet architecture, even small changes produce big impact in terms of the model accuracy and training times. One such example is of the Local Response Normalization layer, without which initially the model reached an accuracy of about 55% on first 10000 steps and got stuck there.

Figure 18 below shows holding/consistency (without decline) of the loss values in the absence of Local Response Normalization (LRN) layer.

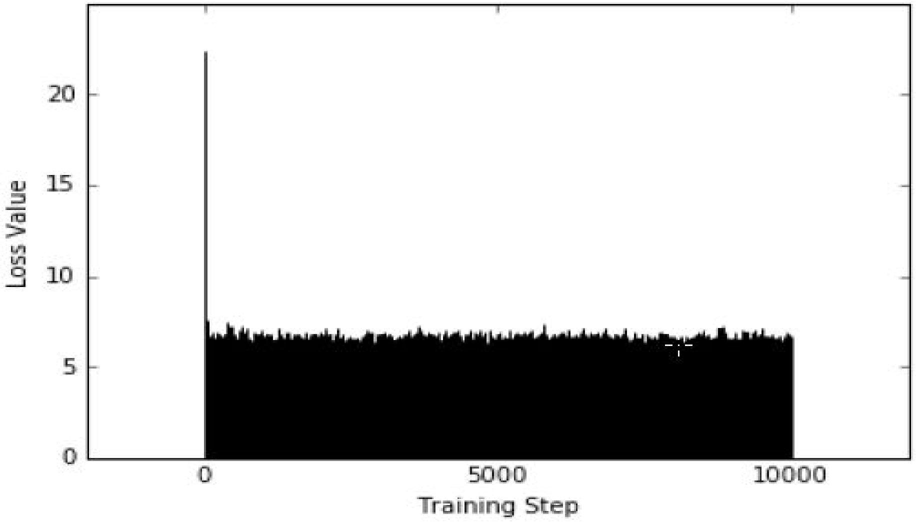


Fig 18. Consistency of Loss Value without LRN.

Improvement

The ConvNets have a high hyperparameter space - in terms of the number of total layers in the architecture, number of Conv layers, Filter sizes, Patch/spatial input sizes, Padding options, Stride size, the kind of pooling layer used, the hidden layers, number of units in the hidden layers, dropout/keep rates, other layers such as Local Response Layer etc

Trying various combinations of all the above options can potentially produce even better results. Also other types of ConvNet architectures such as 1 x 1 Convs, Inceptions etc can be tried as well. Using of GPUs, which are highly suited for utilization with Tensorflow or similar libraries for ConvNets can produce even better results.