1-Introduction

1.1 Project Overview

Optical Character Recognition (OCR) is one of the most important motivations of computer vision and machine learning. It involves the identification of characters or sequences of characters from images. It has significant applications like check processing, document transcription, package routing in couriers and post offices, automatic navigation, etc.

Several datasets have been proposed for benchmarking OCR techniques in different contexts, like handwriting recognition (MNIST Dataset), printed and glyph characters (not MNIST Dataset) and identification in real world images (Street View House Numbers Dataset). While classifying a single character is an easier challenge, identifying a full sequence of characters is a more challenging problem. Street View House Numbers is one such dataset of real world images of house numbers taken from Google Street View images. It has two datasets - character level images and full sequence images in highly varying backgrounds, colors, fonts, relative sizes, resolutions etc.

1.2 Problem Statement

The goal of this project is to predict a full sequence of digits present in an image. The digits must also be in the same left to right reading order. We want to predict every digit correctly, and there will be no partial credit for some correct digits, because, a single digit error can result in a completely different house number. We also want our architecture to be computationally efficient so that it can give near real time results on handheld devices.

For solving this problem, we will train a multi-input multi-output deep learning network. We will divide the problem into two parts – identifying the number of digits in an image, and then identifying each digit in left to right reading order. A counter will count the number of images and generate a range of indices of digits to be identified.

We use full sequence prediction accuracy as the primary performance metric. There's no partial credit for correctly predicting some digits. Since full sequence predictions can either be correct or wrong, accuracy of making correct predictions is an appropriate metric.

2. Analysis

Image analysis is the extraction of meaningful information from images; mainly from digital images by means of digital image processing techniques. Image analysis tasks can be as simple as reading bar coded tags or as sophisticated as identifying a person from their face.

Computers are indispensable for the analysis of large amounts of data, for tasks that require complex computation, or for the extraction of quantitative information. On the other hand, the human visual cortex is an excellent image analysis apparatus, especially for extracting higher-level information, and for many applications — including medicine, security, and remote sensing — human analysts still cannot be replaced by computers. For this reason, many important image analysis tools such as edge detectors and neural networks are inspired by human visual perception models.

2.1 Data Exploration

Street View House Numbers Dataset has two formats, full sequence house numbers and cropped individual digits. In our setup, we will only use the full sequence format, since we'll train our deep learning network to read directly from full sequence images.

Following are the number of samples in each dataset:

Training samples	33402
Testing samples	13068
Flattened training samples	73257

The datasets also contain bounding box information for each digit in every image. However, we will not be using the bounding box data since our algorithm doesn't depend on it. Also, the dataset represents digit 0 with a label value 10. We will change this and represent a digit 0 with a 0. Following are some samples from training dataset.

Figure 1 : Samples of training images

2.2 Exploratory Visualization

We can visualize the distribution of the number of digits in training and test datasets as shown below.

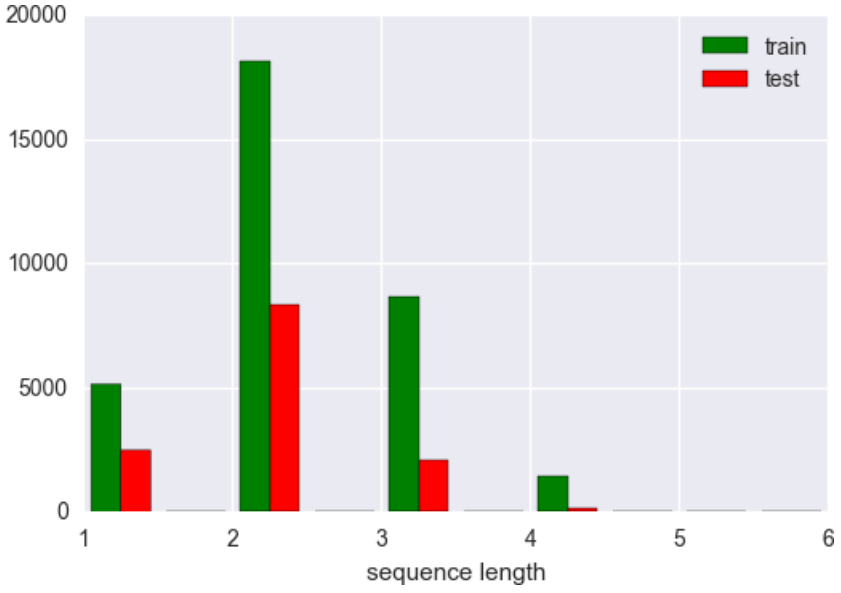


Figure 2 : Distribution of sequence length

Here we can see that the distribution of number of digits is quite asymmetric. Two-digit house numbers are much more common that the rest. Also, house numbers longer than four digits are negligible.

We can also visualize the distribution of digits that make up house numbers as shown below.

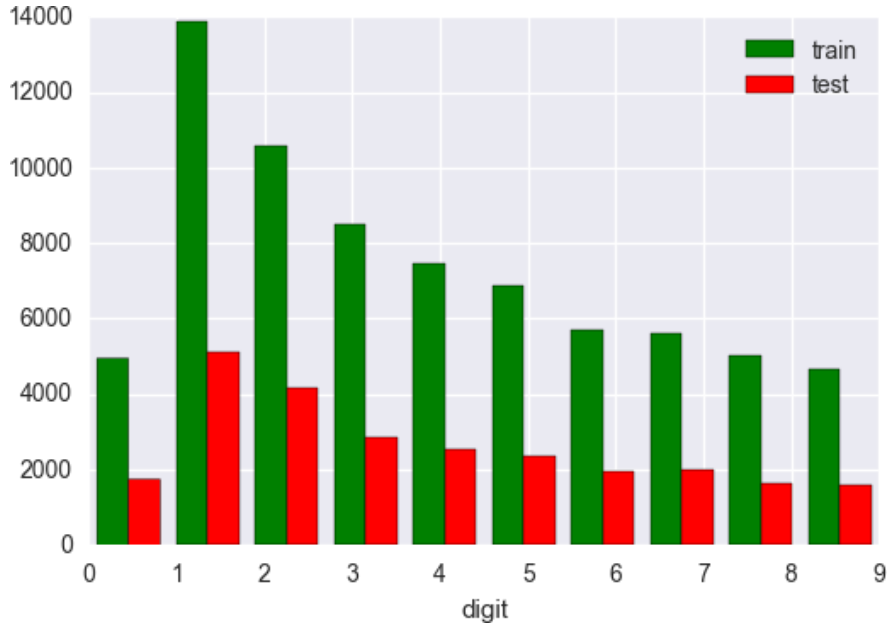


Figure 3 : Distribution of digits

Interestingly, the digit distribution is also quite asymmetric. Higher digits are less common than the lower digits, with digit 1 being the most common.

2.3 Algorithms and Techniques

Our goal is to train a model that predicts a sequence Y composed of individual digits $y_1 y_2 \ldots y_n$ in natural left to right reading order, given an image X, by maximizing the probability $\log P(Y|X)$. We will divide our problem into two parts – predicting a length of sequence n' that maximizes the probability $(n' = n|X)$, and predicting labels y_2' that maximize $P(y_2' = y_2|X)$ for $0 \leq i < n$. Hence, we can express our model as one that maximizes

$$P(Y' = Y|X) = P(n' = n|X) \prod_{i=0}^{n-1} P(y_i' = y_i|X)$$

which is a combination of a digit counter and a digit detector for each index i such that $0 \leq i < n$.

We will use a multi-layer convolutional neural network followed by fully connected layers to form the counter and label detector models. Convolutional networks are well suited for inputs with spatial structures like images and videos.

Since the transformations between detecting digits in an image and making their counts should relatively be simple, we can hope to find a deterministic intermediate output H given input X, such that $P(Y|X) = P(Y|H)$. H will be a one-dimensional tensor which should contain information about all the digits in the image. Using H, the counter model should be able to count the number of digits, and label detector should be able to detect each digit, given its index i. With the shared convolutional model, we can hope to save a lot on computations.

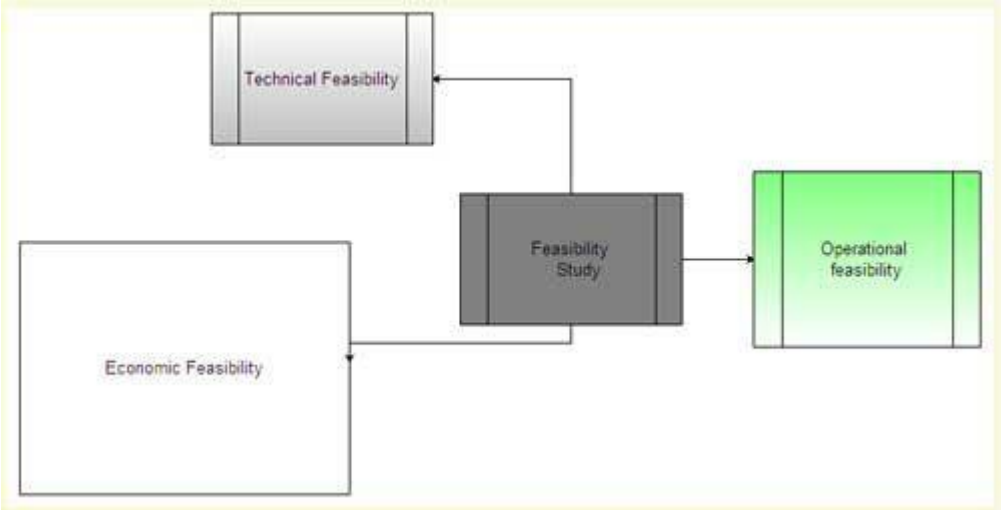
After each activation layer, we use batch normalization to center the mean and have unit variance. During training, batch normalization layers use per batch mean and variance to center node outputs. During prediction, they use the values learnt during training to center the node outputs. In the paper, it was suggested that batch normalization should be used before non-linear activation. However, later research and trends show that better results are achieved if it is used after activation.

Since deep learning networks are quite susceptible to overfitting, we use dropout layers and image augmentation to ensure that training and validation errors don't diverge.

We also use tensor concatenation to input index and transcoded image to the label detector. To train both the networks, we'll try to minimize categorical cross-entropy log loss using Adamax optimizer. Through experimentation, we found that Adamax outperformed other optimizers for this problem in terms of convergence speed.

3. Feasibility study:

Feasibility is defined as the practical extent to which a project can be performed successfully. To evaluate feasibility, a feasibility study is performed, which determines whether the solution considered to accomplish the requirements is practical and workable in the software. Information such as resource availability, cost estimation for software development, benefits of the software to the organization after it is developed and cost to be incurred on its maintenance are considered during the feasibility study. The objective of the feasibility study is to establish the reasons for developing the software that is acceptable to users, adaptable to change and conformable to established standards. Various other objectives of feasibility study are listed below.



3.1 **Technical Feasibility**

Technical feasibility assesses the current resources (such as hardware and software) and technology, which are required to accomplish user requirements in the software within the allocated time and budget. For this, the software development team ascertains whether the current resources and technology can be upgraded or added in the software to accomplish specified user requirements. Technical feasibility also performs the following tasks.

- Analyzes the technical skills and capabilities of the software development team members
- Determines whether the relevant technology is stable and established
- Ascertains that the technology chosen for software development has a large number of users so that they can be consulted when problems arise or improvements are required.

3.2 Operation Feasibility

Operational feasibility assesses the extent to which the required software performs a series of steps to solve business problems and user requirements. This feasibility is dependent on human resources (software development team) and involves visualizing whether the software will operate after it is developed and be operative once it is installed. Operational feasibility also performs the following tasks.

- Determines whether the problems anticipated in user requirements are of high priority
- Determines whether the solution suggested by the software development team is acceptable
- Analyzes whether users will adapt to a new software
- Determines whether the organization is satisfied by the alternative solutions proposed by the software development team.

3.3 Economic Feasibility

Economic feasibility determines whether the required software is capable of generating financial gains for an organization. It involves the cost incurred on the software development team, estimated cost of hardware and software, cost of performing feasibility study, and so on. For this, it is essential to consider expenses made on purchases (such as hardware purchase) and activities required to carry out software development. In addition, it is necessary to consider the benefits that can be achieved by developing the software. Software is said to be economically feasible if it focuses on the issues listed below.

- Cost incurred on software development to produce long-term gains for an organization
- Cost required to conduct full software investigation (such as requirements elicitation and requirements analysis)
- Cost of hardware, software, development team, and training.

4-Methodology

4.1 Data Preprocessing

For training, we flatten the dataset so that each sample point has a processed image, an index of the digit to be detected as inputs, and number of digits in image and the digit present at the given index as output. We do not do any flattening for test data since we have to predict all labels in an image together.

All images for training and testing are scaled to a fixed size of 54 x 128 pixels. An image augmentation system is also used that introduces slight horizontal, vertical, shear and zoom variations in the training images. It also transforms the images to have zero mean and unit variance. Validation and test images are also transformed for zero mean and unit variance. The image augmentation system generates a batch of 64 augmented training images and corresponding data per batch.

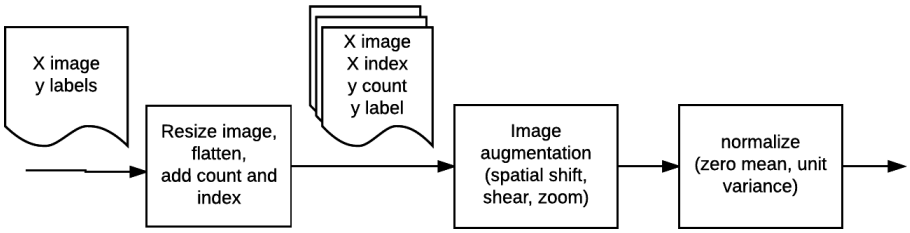


Figure 4 : Data preprocessing for training

Figure 5: sample of image augmentation

4.2 Convolution Neural Network

The power of artificial intelligence is beyond our imagination. We all know robots have already reached a testing phase in some of the powerful countries of the world. Governments, large companies are spending billions in developing this ultra-intelligence creature. The recent existence of robots have gained attention of many research houses across the world.

The earliest research in computer vision started way back in 1950s. Since then, we have come a long way but still find ourselves far from the ultimate objective. But with neural networks and deep learning, we have become empowered like never before.

Applications of deep learning in vision have taken this technology to a different level and made sophisticated things like self-driven cars possible in near future. In this article, I will also introduce you to Convolution Neural Networks which form the crux of deep learning applications in computer vision.

Challenges in Computer Vision

As the name suggests, the aim of computer vision (CV) is to imitate the functionality of human eye and brain components responsible for your sense of sight.

Doing actions such as recognizing an animal, describing a view, differentiating among visible objects are really a cake-walk for humans. You'd be surprised to know that it took decades of research to discover and impart the ability of detecting an object to a computer with reasonable accuracy.

The field of computer vision has witnessed continual advancements in the past 5 years. One of the most stated advancement is Convolution Neural Networks (CNNs). Today, deep CNNs form the crux of most sophisticated fancy computer vision application, such as self-driving cars, auto-tagging of friends in our facebook pictures, facial security features, gesture recognition, automatic number plate recognition, etc.

Object detection is considered to be the most basic application of computer vision. Rest of the other developments in computer vision are achieved by making small enhancements on top of this. In real life, every time we (humans) open our eyes, we unconsciously detect objects.

Since it is super-intuitive for us, we fail to appreciate the key challenges involved when we try to design systems similar to our eye. Lets start by looking at some of the key roadblocks:

1. Variations in Viewpoint

- The same object can have different positions and angles in an image depending on the relative position of the object and the observer.
- There can also be different positions.
- Though its obvious to know that these are the same object, it is not very easy to teach this aspect to a computer (robots or machines).

2. Difference in Illumination

- Different images can have different light conditions.
- Though this image is so dark, we can still recognize that it is a cat. Teaching this to a computer is another challenge.

3. Hidden parts of images

- Images need not necessarily be complete. Small or large proportions of the images might be hidden which makes the detection task difficult.
- If you observe carefully, you can find a man in this image. As simple as it looks, it's an uphill task for a computer to learn.

These are just some of the challenges which I brought up so that you can appreciate the complexity of the tasks which your eye and brain duo does with such utter ease. Breaking up all these challenges and solving individually is still possible today in computer vision. But we're still decades away from a system which can get anywhere close to our human eye (which can do everything!).

This brilliance of our human body is the reason why researchers have been trying to break the enigma of computer vision by **analysing the visual mechanics of humans** or other animals.

Overview of Traditional Approaches

Various techniques, other than deep learning are available enhancing computer vision. Though, they work well for simpler problems, but as the data become huge and the task becomes complex, they are no substitute for deep CNNs. Let's briefly discuss two simple approaches.

KNN (K-Nearest Neighbours)

1. Each image is matched with all images in training data. The top K with minimum distances are selected. The majority class of those top K is predicted as output class of the image.
2. Various distance metrics can be used like L1 distance (sum of absolute distance), L2 distance (sum of squares), etc.
3. Even if we take the image of same object with same illumination and orientation, the object might lie in different locations of image, i.e. left, right or center of image.

Linear Classifiers

1. They use a parametric approach where each pixel value is considered as a parameter.
2. It's like a weighted sum of the pixel values with the dimension of the weights matrix depending on the number of outcomes.
3. Intuitively, we can understand this in terms of a template. The weighted sum of pixels forms a template image which is matched with every image. This will also face difficulty in overcoming the challenges discussed in section 1 as single template is difficult to design for all the different cases.

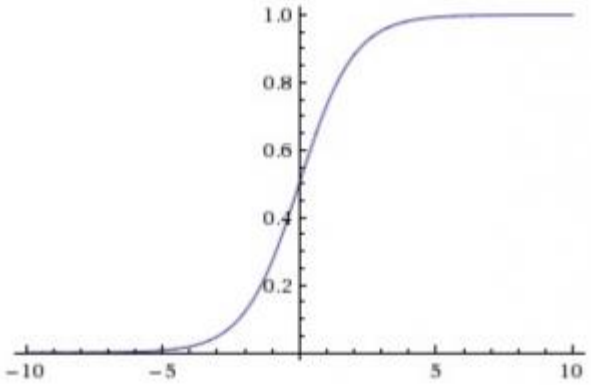
4.3 Review of Neural Networks Fundamentals

Activation Functions

There are various activation functions which can be used and this is an active area of research. Let's discuss some of the popular options:

1. **Sigmoid Function**

- Equation: $\sigma(x) = 1/(1+e^{-x})$



- Sigmoid activation, also used in logistic regression regression, squashes the input space from (-inf,inf) to (0,1)
- But it has various problems and it is **almost never used** in CNNs:

Saturated neurons kill the gradient

1. If you observe the above graph carefully, if the input is beyond -5 or 5, the output will be very close to 0 and 1 respectively. Also, in this region the gradients are almost zero. Notice that the tangents in this region will be almost parallel to x-axis thus ~0 slope.
2. As we know that gradients get multiplied in back-propagation, so this small gradient will virtually stop back-propagation into further layers, thus killing the gradient.

Outputs are not zero-centered

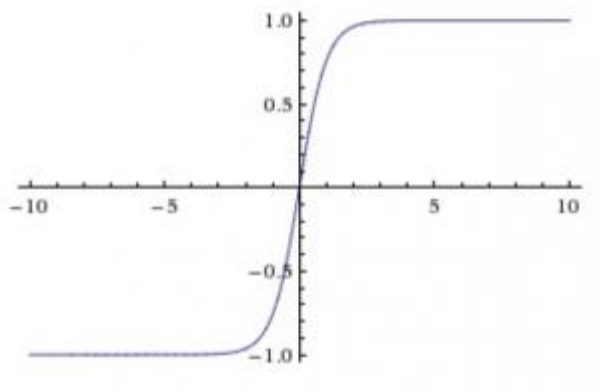
1. As you can see that all the outputs are between 0 and 1. As these become inputs to the next layer, all the gradients of the next layer will be either positive or negative. So the path to optimum will be zig-zag. I will skip the mathematics here. Please refer the Stanford class referred above for details.

Taking the exp() is computationally expensive

1. Though not a big drawback, it has a slight negative impact

2. **tanh activation**

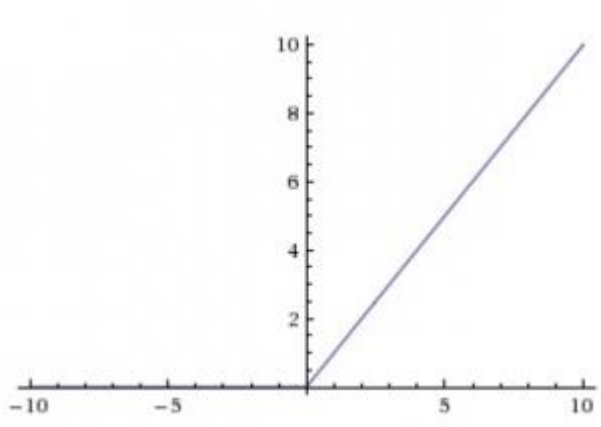
- It is simply the hyperbolic tangent function with form:



- It is always preferred over sigmoid because it solved problem #2, i.e. the outputs are in range (-1,1).
- But it will still result in killing the gradient and thus not recommended choice.

3. ReLU (Rectified Linear Unit)

1. Equation: f(x) = max(0 , x)



2. It is the most commonly used activation function for CNNs. It has following advantages:
    - Gradient won't saturate in the positive region
    - Computationally very efficient as simple thresholding is required
    - Empirically found to converge faster than sigmoid or tanh.
3. But still it has the following disadvantages:
    - Output is not zero-centered and always positive
    - Gradient is killed for x<0. Few techniques like leaky ReLU and parametric ReLU are used to overcome this and I encourage you to find these

- Gradient is not defined at x=0. But this can be easily catered using sub-gradients and posts less practical challenges as x=0 is generally a rare case

To summarize, ReLU is mostly the activation function of choice. If the caveats are kept in mind, these can be used very efficiently.

Data Preprocessing

For images, generally the following preprocessing steps are done:

1. **Same Size Images:** All images are converted to the same size and generally in square shape.
2. **Mean Centering:** For each pixel, its mean value among all images can be subtracted from each pixel. Sometimes (but rarely) mean centering along red, green and blue channels can also be done

Note that normalization is generally not done in images.

Weight Initialization

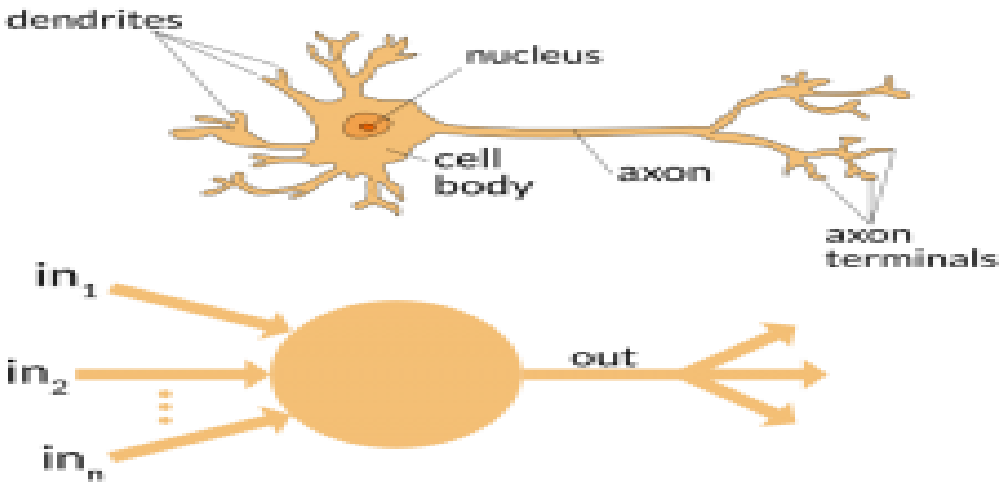
There can be various techniques for initializing weights. Lets consider a few of them:

1. **All zeros**
    - This is generally a **bad idea** because in this case all the neuron will generate the same output initially and similar gradients would flow back in back-propagation
    - The results are generally undesirable as network won't train properly.
2. **Gaussian Random Variables**
    - The weights can be initialized with random gaussian distribution of 0 mean and small standard deviation (0.1 to 1e-5)
    - This works for shallow networks, i.e. ~5 hidden layers but **not for deep networks**
    - In case of deep networks, the small weights make the outputs small and as you move towards the end, the values become even smaller. Thus the gradients will also become small resulting in gradient killing at the end.
    - Note that you need to play with the standard deviation of the gaussian distribution which works well for your network.

3. **Xavier Initialization**
    - It suggests that variance of the gaussian distribution of weights for each neuron should depend on the number of inputs to the layer.
    - The recommended variance is square root of inputs. So the numpy code for initializing the weights of layer with n inputs is: **np.random.randn(n_in, n_out)*sqrt(1/n_in)**
    - A recent research suggested that for ReLU neurons, the recommended update is: **np.random.randn(n_in, n_out)*sqrt(2/n_in)**.

4.3 **Basics of Neural Networks**

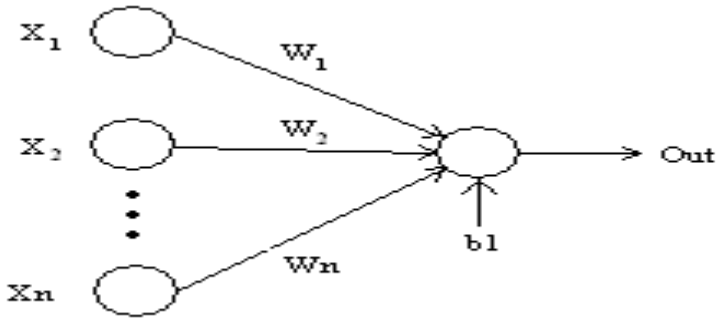
1) Neuron- Just like a neuron forms the basic element of our brain, a neuron forms the basic structure of a neural network. Just think of what we do when we get new information. When we get the information, we process it and then we generate an output. Similarly, in case of a neural network, a neuron receives an input, processes it and generates an output which is either sent to other neurons for further processing or it is the final output.



2) Weights – When input enters the neuron, it is multiplied by a weight. For example, if a neuron has two inputs, then each input will have has an associated weight assigned to it. We initialize the weights randomly and these weights are updated during the model training process. The neural network after training assigns a higher weight to the input it considers more important as compared

to the ones which are considered less important. A weight of zero denotes that the particular feature is insignificant.

Let's assume the input to be a, and the weight associated to be W1. Then after passing through the node the input becomes a*W1



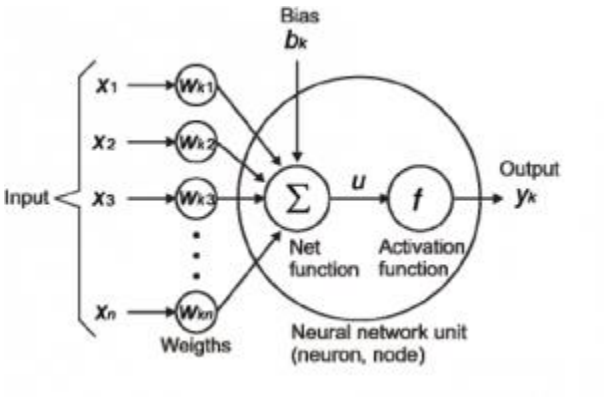
3) Bias – In addition to the weights, another linear component is applied to the input, called as the bias. It is added to the result of weight multiplication to the input. The bias is basically added to change the range of the weight multiplied input. After adding the bias, the result would look like a*W1+bias. This is the final linear component of the input transformation.

4) Activation Function – Once the linear component is applied to the input, a non-linear function is applied to it. This is done by applying the activation function to the linear combination. The activation function translates the input signals to output signals. The output after application of the activation function would look something like f(a*W1+b) where f() is the activation function.

In the below diagram we have "n" inputs given as X1 to Xn and corresponding weights Wk1 to Wkn. We have a bias given as bk. The weights are first multiplied to its corresponding input and are then added together along with the bias. Let this be called as u.

$u=\sum w*x+b$

The activation function is applied to u i.e. f(u) and we receive the final output from the neuron as yk = f(u)

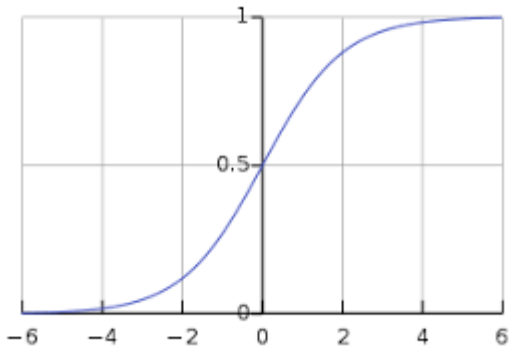


Commonly applied Activation Functions

The most commonly applied activation functions are – Sigmoid, ReLU and softmax

a) Sigmoid – One of the most common activation functions used is Sigmoid. It is defined as:

$$\text{sigmoid}(x) = 1/(1+e^{-x})$$

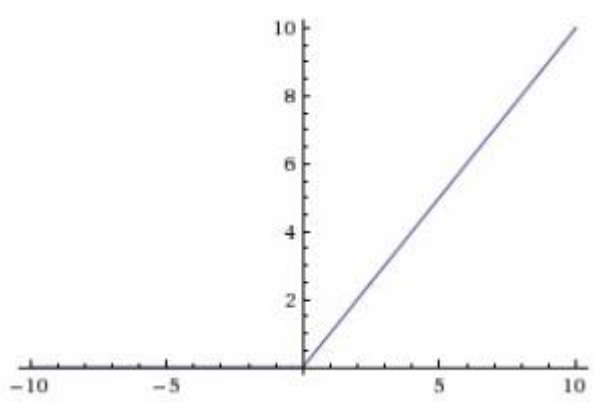


The sigmoid transformation generates a more smooth range of values between 0 and 1. We might need to observe the changes in the output with slight changes in the input values. Smooth curves allow us to do that and are hence preferred over step functions.

b) ReLU(Rectified Linear Units) – Instead of sigmoids, the recent networks prefer using ReLu activation functions for the hidden layers. The function is defined as:

f(x) = max(x,0).

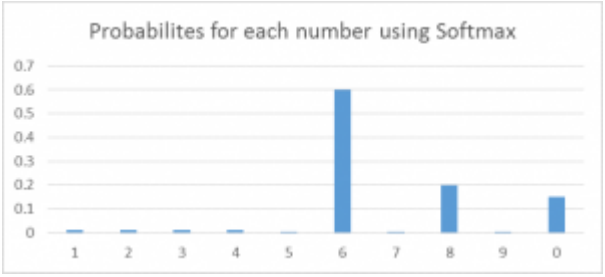
The output of the function is X when X>0 and 0 for X<=0. The function looks like this:



The major benefit of using ReLU is that it has a constant derivative value for all inputs greater than 0. The constant derivative value helps the network to train faster.

c) Softmax – Softmax activation functions are normally used in the output layer for classification problems. It is similar to the sigmoid function, with the only difference being that the outputs are normalized to sum up to 1. The sigmoid function would work in case we have a binary output, however in case we have a multiclass classification problem, softmax makes it really easy to assign values to each class which can be easily interpreted as probabilities.

It's very easy to see it this way – Suppose you're trying to identify a 6 which might also look a bit like 8. The function would assign values to each number as below. We can easily see that the highest probability is assigned to 6, with the next highest assigned to 8 and so on…

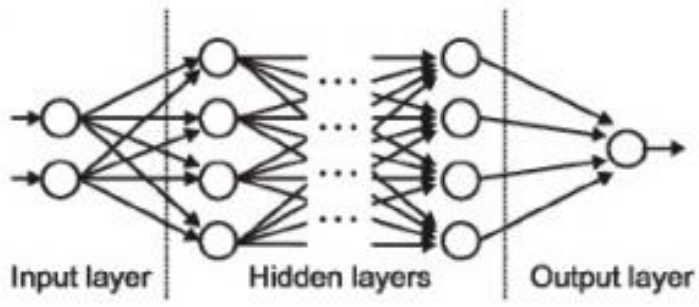


5) Neural Network – Neural Networks form the backbone of deep learning. The goal of a neural network is to find an approximation of an unknown function. It is formed by interconnected neurons. These neurons have weights, and bias which is updated during the network training depending upon the error. The activation function puts a nonlinear transformation to the linear combination which then generates the output. The combinations of the activated neurons give the output.

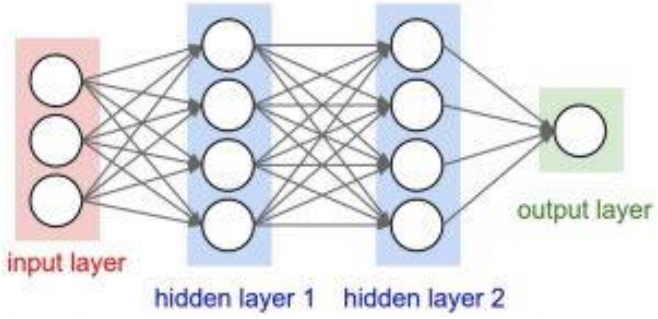
A neural network is best defined by "Liping Yang" as –

"Neural networks are made up of numerous interconnected conceptualized artificial neurons, which pass data between themselves, and which have associated weights which are tuned based upon the network's "experience." Neurons have activation thresholds which, if met by a combination of their associated weights and data passed to them, are fired; combinations of fired neurons result in "learning".

6) Input / Output / Hidden Layer – Simply as the name suggests the input layer is the one which receives the input and is essentially the first layer of the network. The output layer is the one which generates the output or is the final layer of the network. The processing layers are the hidden layers within the network. These hidden layers are the ones which perform specific tasks on the incoming data and pass on the output generated by them to the next layer. The input and output layers are the ones visible to us, while are the intermediate layers are hidden.



7) MLP (Multi Layer perceptron) – A single neuron would not be able to perform highly complex tasks. Therefore, we use stacks of neurons to generate the desired outputs. In the simplest network we would have an input layer, a hidden layer and an output layer. Each layer has multiple neurons and all the neurons in each layer are connected to all the neurons in the next layer. These networks can also be called as fully connected networks.



8) Forward Propagation – Forward Propagation refers to the movement of the input through the hidden layers to the output layers. In forward propagation, the information travels in a single direction FORWARD. The input layer supplies the input to the hidden layers and then the output is generated. There is no backward movement.

9) Cost Function – When we build a network, the network tries to predict the output as close as possible to the actual value. We measure this accuracy of the network using the cost/loss function. The cost or loss function tries to penalize the network when it makes errors.

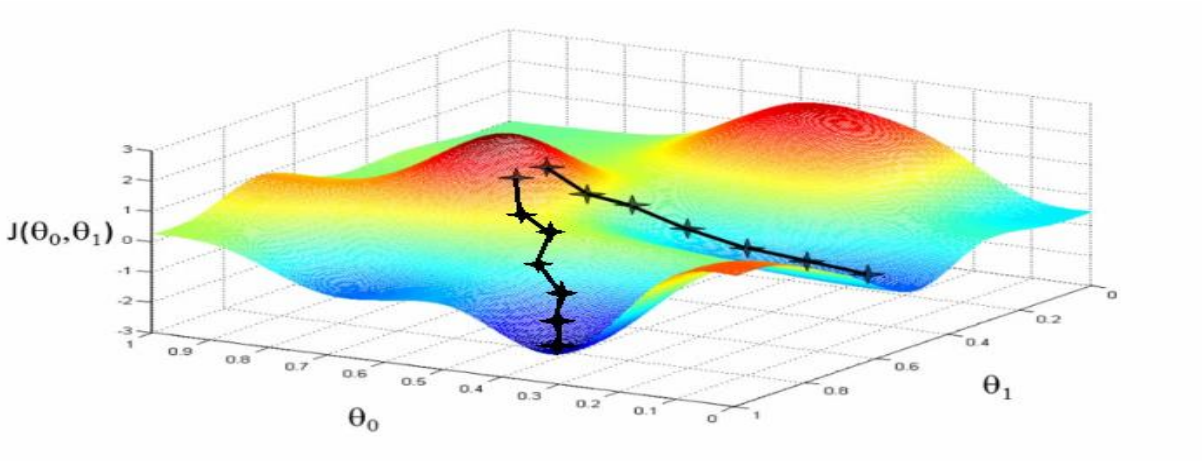
Our objective while running the network is to increase our prediction accuracy and to reduce the error, hence minimizing the cost function. The most optimized output is the one with least value of the cost or loss function.

If I define the cost function to be the mean squared error, it can be written as –

$C= 1/m \sum(y - a)^2$ where m is the number of training inputs, a is the predicted value and y is the actual value of that particular example.

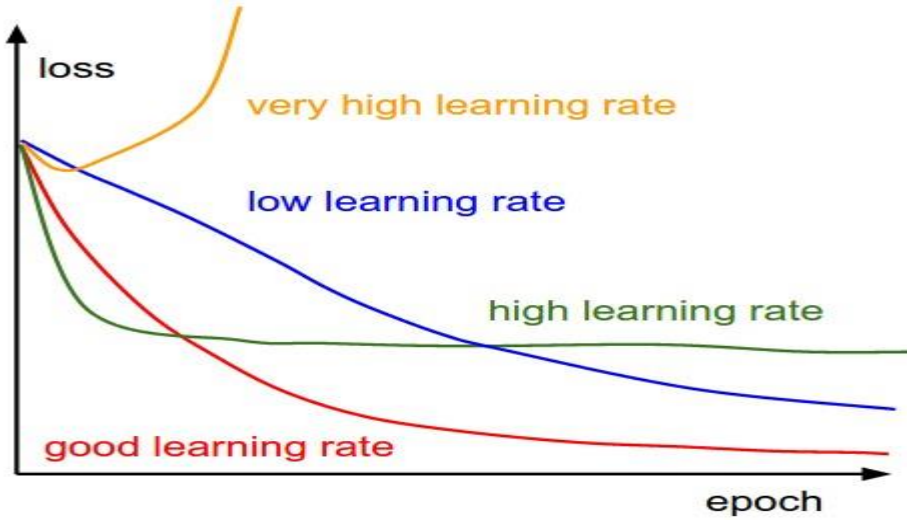
The learning process revolves around minimizing the cost.

10) Gradient Descent – Gradient descent is an optimization algorithm for minimizing the cost. To think of it intuitively, while climbing down a hill you should take small steps and walk down instead of just jumping down at once. Therefore, what we do is, if we start from a point x, we move down a little i.e. delta h, and update our position to x-delta h and we keep doing the same till we reach the bottom. Consider bottom to be the minimum cost point.



Mathematically, to find the local minimum of a function one takes steps proportional to the negative of the gradient of the function.

11) Learning Rate – The learning rate is defined as the amount of minimization in the cost function in each iteration. In simple terms, the rate at which we descend towards the minima of the cost function is the learning rate. We should choose the learning rate very carefully since it should neither be very large that the optimal solution is missed and nor should be



very low that it takes forever for the network to converge.

12) Backpropagation – When we define a neural network, we assign random weights and bias values to our nodes. Once we have received the output for a single iteration, we can calculate the error of the network. This error is then fed back to the network along with the gradient of the cost function to update the weights of the network. These weights are then updated so that the errors in the subsequent iterations is reduced. This updating of weights using the gradient of the cost function is known as back-propagation.

In back-propagation the movement of the network is backwards, the error along with the gradient flows back from the out layer through the hidden layers and the weights are updated.

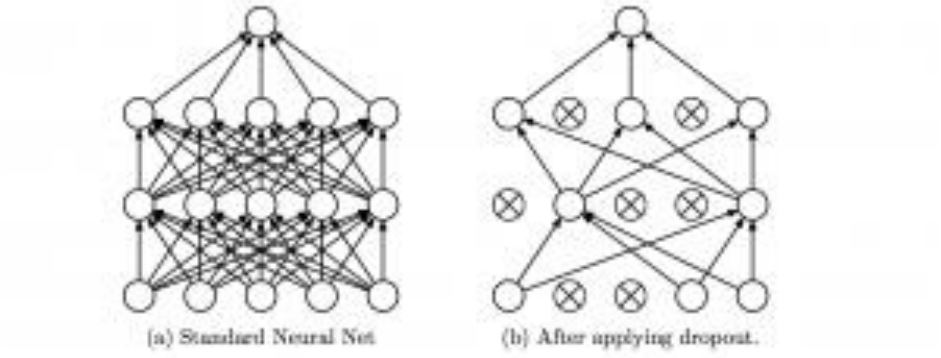
13) Batches – While training a neural network, instead of sending the entire input in one go, we divide in input into several chunks of equal size randomly. Training the data on batches makes the model more generalized as compared to the model built when the entire data set is fed to the network in one go.

14) Epochs – An epoch is defined as a single training iteration of all batches in both forward and back propagation. This means 1 epoch is a single forward and backward pass of the entire input data.

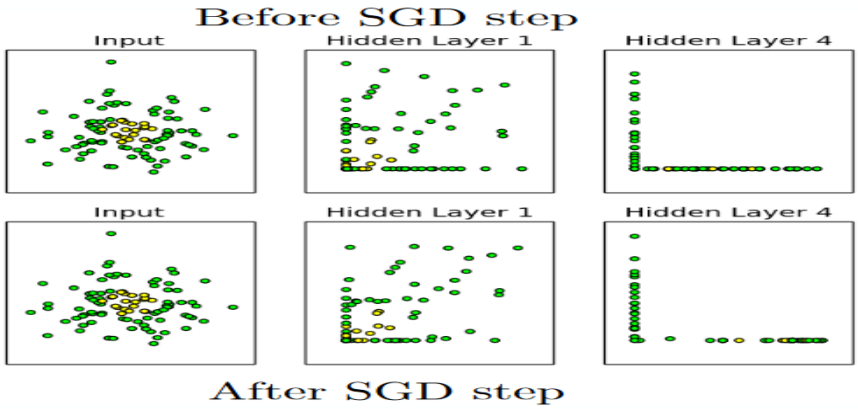
The number of epochs you would use to train your network can be chosen by you. It's highly likely that more number of epochs would show higher accuracy of the network, however, it would also

take longer for the network to converge. Also you must take care that if the number of epochs are too high, the network might be over-fit.

15) Dropout – Dropout is a regularization technique which prevents over-fitting of the network. As the name suggests, during training a certain number of neurons in the hidden layer is randomly dropped. This means that the training happens on several architectures of the neural network on different combinations of the neurons. You can think of drop out as an ensemble technique, where the output of multiple networks is then used to produce the final output.



16) Batch Normalization – As a concept, batch normalization can be considered as a dam we have set as specific checkpoints in a river. This is done to ensure that distribution of data is the same as the next layer hoped to get. When we are training the neural network, the weights are changed after each step of gradient descent. This changes the how the shape of data is sent to the next layer.



"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Ioffe and Szegedy 2015

But the next layer was expecting the distribution similar to what it had previously seen. So we explicitly normalize the data before sending it to the next layer.

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{W}$$

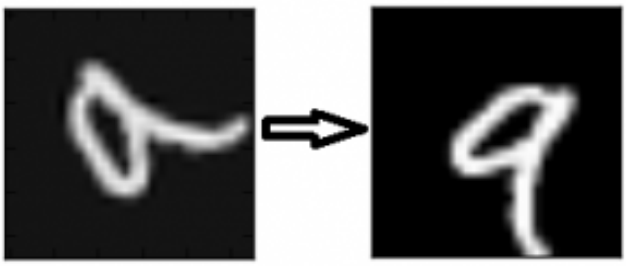
$$\tilde{\boldsymbol{Z}} = \boldsymbol{Z} - \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{Z}_{i,:}$$

$$\hat{\boldsymbol{Z}} = \frac{\tilde{\boldsymbol{Z}}}{\sqrt{\epsilon + \frac{1}{m}\sum_{i=1}^{m} \tilde{\boldsymbol{Z}}_{i,:}^2}}$$

$$\boldsymbol{H} = \max\{0, \boldsymbol{\gamma}\hat{Z} + \boldsymbol{\beta}\}$$

"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Ioffe and Szegedy 2015

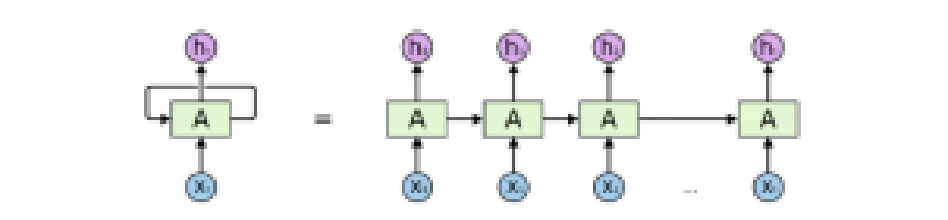
17) Data Augmentation – Data Augmentation refers to the addition of new data derived from the given data, which might prove to be beneficial for prediction. For example, it might be easier to view the cat in a dark image if you brighten it, or for instance, a 9 in the digit recognition might be slightly tilted or rotated. In this case, rotation would solve the problem and increase the accuracy of our model. By rotating or brightening we're improving the quality of our data. This is known as Data augmentation.



Recurrent Neural Network

18) Recurrent Neuron – A recurrent neuron is one in which the output of the neuron is sent back to it for t time stamps. If you look at the diagram the output is sent back as input t times. The

unrolled neuron looks like t different neurons connected together. The basic advantage of this neuron is that it gives a more generalized output.



Source: cs231n

19) RNN(Recurrent Neural Network) – Recurrent neural networks are used especially for sequential data where the previous output is used to predict the next one. In this case the networks have loops within them. The loops within the hidden neuron gives them the capability to store information about the previous words for some time to be able to predict the output. The output of the hidden layer is sent again to the hidden layer for t time stamps. The unfolded neuron looks like the above diagram. The output of the recurrent neuron goes to the next layer only after completing all the time stamps. The output sent is more generalized and the previous information is retained for a longer period.The error is then back propagated according to the unfolded network to update the weights. This is known as **backpropagation through time (BPTT).**

20) Vanishing Gradient Problem – Vanishing gradient problem arises in cases where the gradient of the activation function is very small. During back propagation when the weights are multiplied with these low gradients, they tend to become very small and "vanish" as they go further deep in the network. This makes the neural network to forget the long range dependency. This generally becomes a problem in cases of recurrent neural networks where long term dependencies are very important for the network to remember.

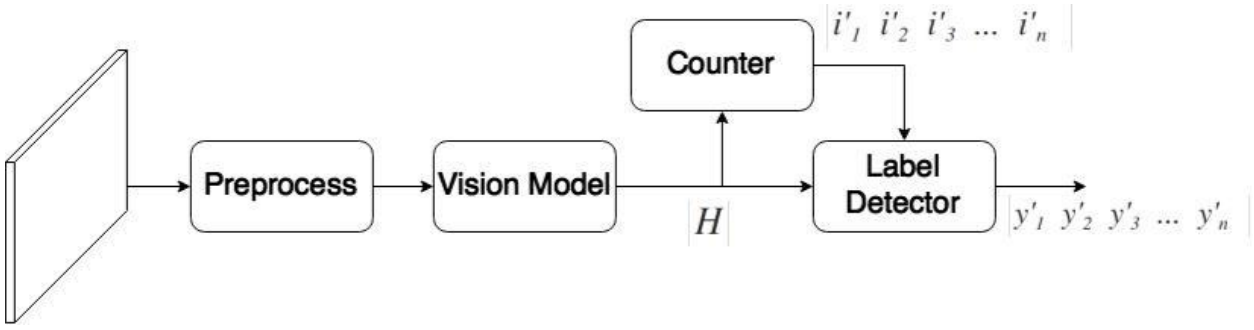
This can be solved by using activation functions like ReLu which do not have small gradients.

21) Exploding Gradient Problem – This is the exact opposite of the vanishing gradient problem, where the gradient of the activation function is too large. During back propagation, it makes the weight of a particular node very high with respect to the others rendering them insignificant. This can be easily solved by clipping the gradient so that it doesn't exceed a certain value.

5. Project Design

We'll create a shared vision model that will feed convoluted images to a counter model and a label de-tector. This vision model will have a series of convolutional, dropout and pooling layers, and a fully connected layer in the end. This shared model approach will process each image with convolution filters only once and produce a dense tensor which can be ingested by both counter and label detector. The label detector will take an additional in-put *i,* the index of the digit to be detected.

The idea of having a shared vision model is to process the input image only once for both digit counting and label detection. The vision model will produce a convolution ten-sor that can be further processed by both counter and label detection models. Counter will generate a single output n , the number of digits in the image. We can then pass in-dices 1 to n, with the convoluted tensor H to label detector, which will generate the in-dexed sequence.



6. Implementation

We implemented our deep learning network in Keras with Tensorflow backend. Our deep learning network is composed of following three macro models:

6.1 Create Deep Learning Network

Our deep learning network consists of a shared vision model, a counter and a label detector.

6.1.1 Vision

Vision model is a shared convolutional network that processes input images and transforms them into a dense tensor of length 1024. It has six convolution layers with feature sizes 32, 32, 64, 64, 128 and 128. All convolution layers have a window size of 3x3. Every convolution layer is followed by rectified linear unit activation and batch normalization. The input layer uses tanh activation as it provided better results. After every two convolution/activation/normalization pairs, we also use dropout and max pooling layers. After the convolution layers we have two fully connected layers with ReLU activations and batch normalizations. The output of last fully connected layer is the intermediate output H of width 1024 from the vision model, which is input to counter and detector models.

The vision model processes input image of fixed size using Convolutional Neural Networks and produces a dense tensor of shape (1024,). This tensor is then processed by a counter which is made of fully connected layers. The counter output is used to generate indices, which are combined with the vision model output and fed to label detector, which outputs a label for each index.

```
from keras.layers import Input, Dense, Merge, Flatten, Dropout, merge
from keras.layers.convolutional import Convolution2D
from keras.layers.pooling import MaxPooling2D
from keras.models import Model, Sequential
from keras.layers.normalization import BatchNormalization

# define vision model
image_in_vision = Input(shape=(image_size[0],image_size[1],3))
x = BatchNormalization(axis=3)(image_in_vision)
x = Convolution2D(32, 3, 3, activation='tanh')(x)
x = BatchNormalization(axis=3)(x)
x = Convolution2D(32, 3, 3, activation='relu')(x)
x = BatchNormalization(axis=3)(x)
x = Dropout(0.2)(x)
x = MaxPooling2D(pool_size=(2,2))(x)
x = Convolution2D(64, 3, 3, activation='relu')(x)
x = BatchNormalization(axis=3)(x)
x = Convolution2D(64, 3, 3, activation='relu')(x)
x = BatchNormalization(axis=3)(x)
x = Dropout(0.2)(x)
x = MaxPooling2D(pool_size=(2,2))(x)
x = Convolution2D(128, 3, 3, activation='relu')(x)
x = BatchNormalization(axis=3)(x)
x = Convolution2D(128, 3, 3, activation='relu')(x)
x = BatchNormalization(axis=3)(x)
x = Dropout(0.2)(x)
x = MaxPooling2D(pool_size=(2,2))(x)
x = Flatten()(x)
x = Dense(1024, activation='relu')(x)
x = BatchNormalization()(x)
x = Dense(1024, activation='relu')(x)
h = BatchNormalization()(x)
vision_model = Model(input=image_in_vision, output=h, name='vision')
```

6.1.2 Counter

The counter model takes the output from vision model and predicts the number of digits present in the input image. It contains a fully connected layer of 256 nodes with ReLU activations and an output layer of 7 nodes with softmax activations.

```
# define counter model
h_in_counter = Input(shape=(1024,))
yc = Dense(256, activation='relu')(h_in_counter)
yc = BatchNormalization()(yc)
yc = Dropout(0.2)(yc)
yc = Dense(max_digits, activation='softmax')(yc)
counter_model = Model(input=h_in_counter, output=yc, name='counter')
```

6.1.3 Detector

The detector model has two inputs, *H* and the one hot encoded index of the digit to be detected. The first layer of detector merges these two inputs by concatenating. It then has two fully connected layers of width 512 nodes and ReLU activations, and an output layer of 10 nodes with softmax activations.

```
# define detector model
h_in_detector = Input(shape=(1024,))
idx_in_detector = Input(shape=(max_digits,))
yl = merge([h_in_detector, idx_in_detector], mode='concat')
yl = Dense(512, activation='relu')(yl)
yl = BatchNormalization()(yl)
yl = Dense(512, activation='relu')(yl)
yl = BatchNormalization()(yl)
yl = Dropout(0.2)(yl)
yl = Dense(10, activation='softmax')(yl)

detector_model = Model(input=[h_in_detector, idx_in_detector], output=yl, name='detector')
```

6.2 Graph Creation

The organization of these models in training and testing graphs is slightly different. During training phase, the index input to detector is provided from flattened training data, whereas during testing, this input is generated from the range of count that counter predicts. To achieve this reorganization, the individual models were first created using Keras' functional api with names clearly defined. For example,

```
define counter model
h_in_counter = Input(shape = (1024, ))
yc = Dense(256, activation = 'relu')(h_in_counter)
yc = BatchNormalization()(yc)
yc = Dropout(0.2)(yc)
yc = Dense(max_digits, activation = 'softmax')(yc)
counter_model = Model(input = h_in_counter, output = yc, name = 'counter')
```

We can then define the training graph by calling the models like functions

```
Ximg_in = Input(shape = (image_size[0], image_size[1], 3), name ='train_input_ img')
Xidx_in = Input(shape = (max_digits, ), name = 'train_input_idx')
```

```
h = vision_model(Ximg_in)
yc = counter_model(h)
yl = detector_model([h, Xidx_in])
train_graph = Model(input = [Ximg_in, Xidx_in], output = [yc, yl])
```

The training graph is then saved using Keras CheckPoint callback. The individual models are retrieved later during testing by enumerating graph layers and referencing appropriately.

```
extract the individual models from training graph
vision = model.layers[1]
counter = model.layers[3]
detector = model.layers[4]
```

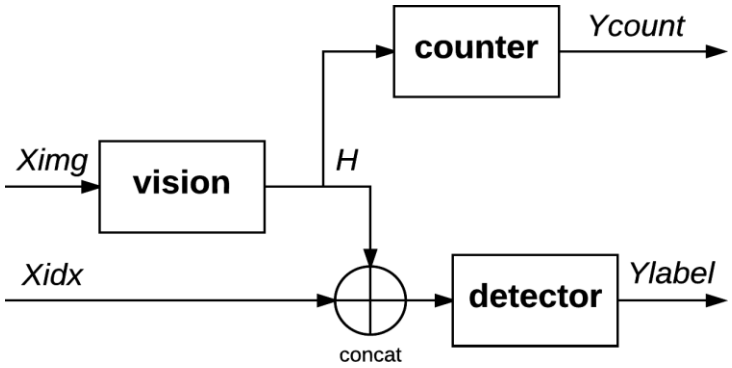


Figure 6 : architecture for training

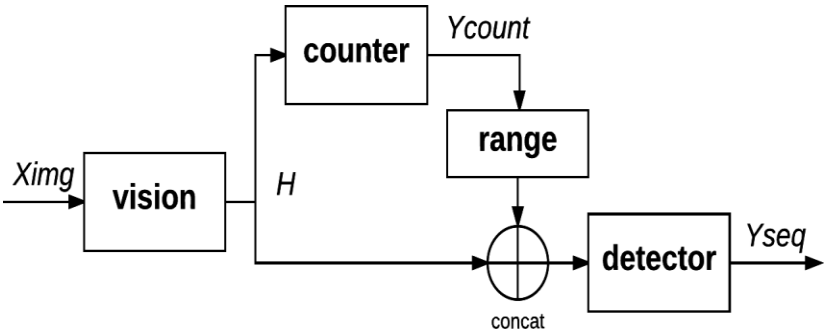


Figure 7 : architecture for prediction

The model was trained for 50 epochs on an Amazon AWS p2.xlarge instance using a custom AMI that had Keras, Tensorflow, CUDA and CuDNN installed among other libraries. 5% of training samples were used for validation.

6.3 Refinement

In initial models, we did not use image augmentation and batch normalization, and only used dropouts for regularization. These models had significant overfitting and very poor accuracy on flattened validation data. Increasing dropouts didn't help as the network would then stop.

Training History

```
plt.figure(figsize=(12,4))
plt.subplot(1, 2, 1)
ctl_plot = plt.plot(history.history['counter_loss'], 'r', label='training')
cvl_plot = plt.plot(history.history['val_counter_loss'], 'g', label='validation')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.title('counter loss')
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(history.history['counter_acc'], 'r', label='training')
plt.plot(history.history['val_counter_acc'], 'g', label='validation')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('counter accuracy')
plt.legend(loc=4)
```

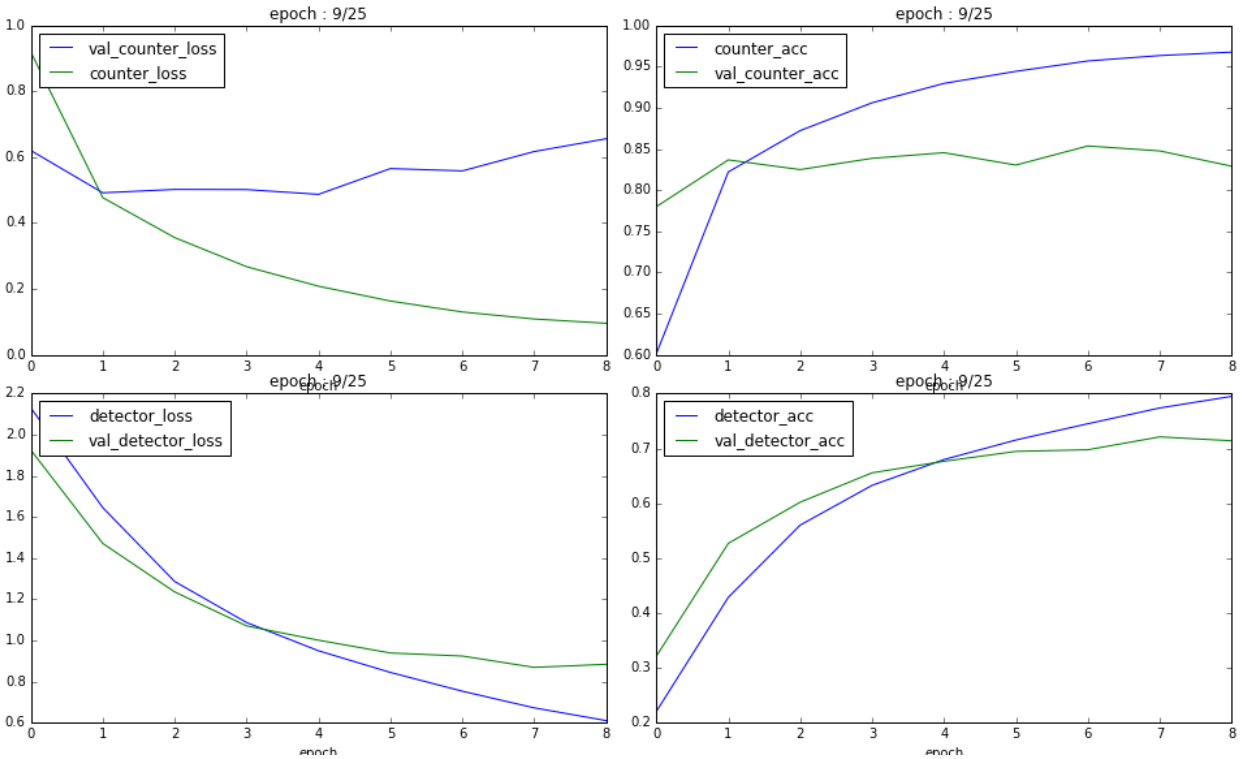


Figure 8 : training session without image augmentation

Converging. Above is one training session.

However, after an image augmentation generator was included, we saw almost no overfitting as evident from following training and validation graphs. Image augmentation makes sure that the network being trained doesn't necessarily see the same training image twice.

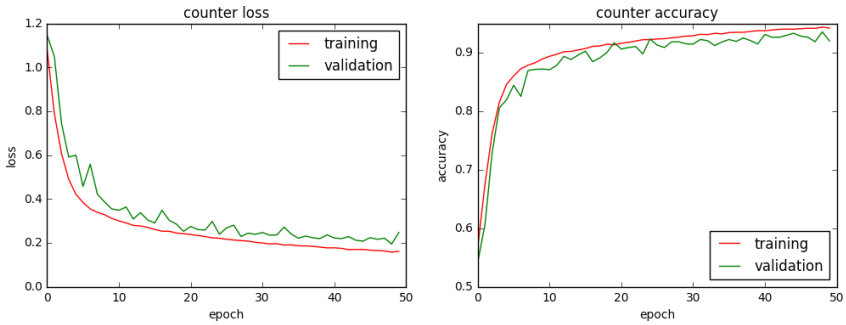


Figure 9 : counter training with image augmentation and batch normalization

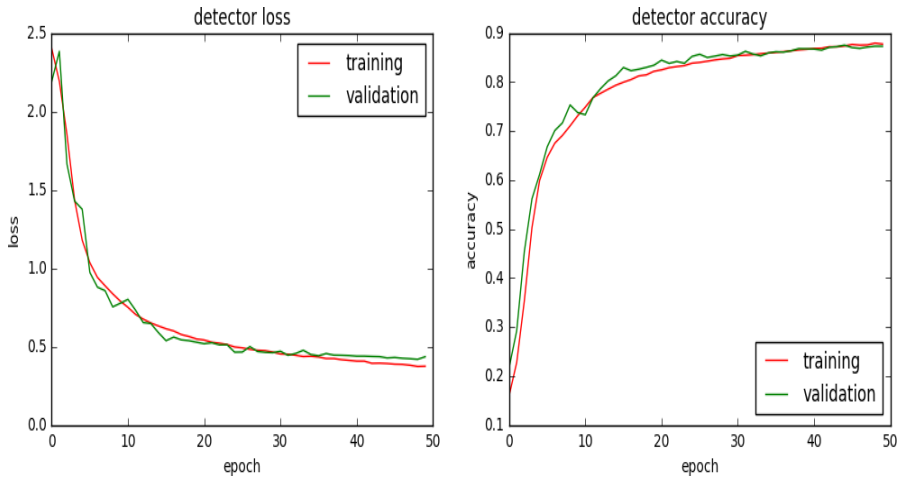


Figure 10 : detector training with image augmentation and batch normalization

On adding batch normalization, the individual accuracies of counter and detector further shot up by 4% on validation data.

6.4 Integration with Flask Web Framework

Flask is a micro web framework written in Python and based on the Werkzeug toolkit and Jinja2 template engine. Applications that use the Flask framework include Pinterest, LinkedIn, and the community web page for Flask itself.

Flask is classified as a micro framework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Extensions exist for object-relational mappers, form validation, and upload handling, various open authentication technologies and several common framework related tools. Extensions are updated far more regularly than the core Flask program.

We have used flask web framework to integrate python code and host it on local server The Input image is uploaded with flask uploader function. Input image is stored in a file and later function call occurs to start detection of digits in image. When detection completes it will take the user to next page to show the digit sequence. Following are Some Code Snippet which is the code of Uploader function

```
import pefile
import os
import array
import math
import pickle
from sklearn.externals import joblib
import sys
import argparse
import os, sys, shutil, time
import re
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import keras
from scipy import misc
from keras.utils import np_utils
import graphics
import os

image_size = (54,128)
max_digits = 7


from flask import Flask, request, jsonify, render_template,abort,redirect,url_for
from werkzeug import secure_filename
from sklearn.externals import joblib
```

```
app=Flask(__name__)
# app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
@app.route('/')
def index():
    return render_template('index.html')




@app.route('/uploader', methods = ['GET', 'POST'])
def upload_file():
   if request.method == 'POST':
    f = request.files['file']
    file_name_input=secure_filename(f.filename)
    # print(os.path)
    path='../inputs/custom/'+file_name_input
    f.save(path)
    # print('jdfjsd')
    ans=Predict()
    print('hidshf')
    ansstr=''.join(str(x) for x in ans)
    return render_template('result1.html',hello=ansstr)
```

7 Tools & Framework

7.1 Anaconda

Anaconda is a free and open source distribution of the Python and R programming languages for data science and machine learning related applications (large-scale data processing, predictive analytics, scientific computing), that aims to simplify package management and deployment. Package versions are managed by the package management system conda. The Anaconda distribution is used by over 6 million users, and it includes more than 250 popular data science packages suitable for Windows, Linux, and MacOS.

The Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more**.**

7.2 Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.
Use Keras if you need a deep learning library that:

Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
Supports both convolutional networks and recurrent networks, as well as combinations of the two.
Runs seamlessly on CPU and GPU.

Guiding principles

- **User friendliness**. Keras is an API designed for human beings, not machines. It puts user experience front and center. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.

- **Modularity**. A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models.

- **Easy extensibility**. New modules are simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new modules allows for total expressiveness, making Keras suitable for advanced research.

- **Work with Python**. No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easier to debug, and allows for ease of extensibility.

Why use Keras?

There are countless deep learning frameworks available today. Why use Keras rather than any other? Here are some of the areas in which Keras compares favorably to existing alternatives.

Keras prioritizes developer experience

- Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.

- This makes Keras easy to learn and easy to use. As a Keras user, you are more productive, allowing you to try more ideas than your competition, faster -- which in turn helps you win machine learning competitions.

- This ease of use does not come at the cost of reduced flexibility: because Keras integrates with lower-level deep learning languages (in particular TensorFlow), it enables you to implement anything you could have built in the base language.

Keras is a high-level API for neural networks. It is written in Python and its biggest advantage is its ability to run on top of state-of-art deep learning libraries/frameworks such as TensorFlow, CNTK or Theano. If you are looking for fast prototyping with deep learning, then Keras is the optimal choice.

Deep learning is the new buzzword among machine learning researchers and practitioners. It has certainly opened the doors to solving problems that were almost unsolvable earlier. Examples of such problems are image recognition, speaker-independent voice recognition, video understanding, etc. Neural networks are at the core of deep learning methodologies for solving problems. The improvements in these networks, such as convolutional neural networks (CNN) and recurrent networks, have certainly raised expectations and the results they yield are also promising.

To make the approach simple, there are already powerful frameworks/libraries such as TensorFlow from Google and CNTK (Cognitive Toolkit) from Microsoft. The TensorFlow approach has already simplified the implementation of deep learning for coders. Keras is a high-level API for neural networks written in Python, which makes things even simpler. The uniqueness of Keras is that it can be executed on top of libraries such as TensorFlow and CNTK. This article assumes that the reader is familiar with the fundamental concepts of machine learning.

The primary reasons for using Keras are:

- Instant prototyping: This is ability to implement the deep learning concepts with higher levels of abstraction with a ‘keep it simple’ approach.
- Keras has the potential to execute without any barriers on CPUs and GPUs.
- Keras supports convolutional and recurrent networks — combinations of both can also be used with it.
- It is an API designed with user friendly implementation as the core principle. The API is designed to be simple and consistent, and it minimises the effort programmers are required to put in to convert theory into action.
- Keras’ modular design is another important feature. The primary idea of Keras is layers, which can be connected seamlessly.

- Keras is extensible. If you are a researcher trying to bring in your own novel functionality, Keras can accommodate such extensions.
- Keras is all Python, so there is no need for tricky declarative configuration files.

Keras has various types of pre-built layers. Some of the prominent types are:

- Regular Dense
- Recurrent Layers, LSTM, GRU, etc
- One- and two-dimension convolutional layers
- Dropout
- Noise
- Pooling
- Normalisation, etc

Similarly, Keras supports most of the popularly used activation functions. Some of these are:

- Sigmoid
- ReLu
- Softplus
- ELU
- LeakyReLu, etc

7.3 Tensor Flow

TensorFlow is an open source software library for numerical computation using dataflow graphs. Nodes in the graph represents mathematical operations, while graph edges represent multi-dimensional data arrays (aka tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

The advantages of using TensorFlow are:

- **It has an intuitive construct**, because as the name suggests it has *"flow of tensors".* You can easily visualize each and every part of the graph.
- **Easily train on cpu/gpu for distributed computing**
- **Platform flexibility**. You can run the models wherever you want, whether it is on mobile, server or PC.

A typical "flow" of TensorFlow

Every library has its own "implementation details", i.e. a way to write which follows its coding paradigm. For example, when implementing scikit-learn, you first create object of the desired algorithm, then build a model on train and get predictions on test set, something like this:

The usual workflow of running a program in TensorFlow is as follows:

- **Build a computational graph,** this can be any mathematical operation TensorFlow supports.
- **Initialize variables,** to compile the variables defined previously
- **Create session,** this is where the magic starts!
- **Run graph in session,** the compiled graph is passed to the session, which starts its execution.
- **Close session,** shutdown the session.

Implementing Neural Network in TensorFlow

A typical implementation of Neural Network would be as follows:

- Define Neural Network architecture to be compiled
- Transfer data to your model
- Under the hood, the data is first divided into batches, so that it can be ingested. The batches are first preprocessed, augmented and then fed into Neural Network for training
- The model then gets trained incrementally
- Display the accuracy for a specific number of time steps
- After training save the model for future use
- Test the model on a new data and check how it performs

Limitations of TensorFlow

- Even though TensorFlow is powerful, it's still a low level library. For example, it can be considered as a machine level language. But for most of the purpose, you need modularity and high level interface such as keras
- It's still in development, so much more awesomeness to come!
- It depends on your hardware specs, the more the merrier
- Still not an API for many languages.
- There are still many things yet to be included in TensorFlow, such as OpenCL support.

TensorFlow vs. Other Libraries

TensorFlow is built on similar principles as Theano and Torch of using mathematical computational graphs. But with the additional support of distributed computing, TensorFlow comes out to be better at solving complex problems. Also deployment of TensorFlow models is already supported which makes it easier to use for industrial purposes, giving a fight to commercial libraries such as Deeplearning4j, H2O and Turi. TensorFlow has APIs for Python, C++ and Matlab. There's also a recent surge for support for other languages such as Ruby and R.

8 Prediction on Test data

8.1 Load Trained Model

Saved model is loaded using keras and hd5y library. Model have 3 layers vision, counter and label Input image is first fed in vision layer then counter and then label.

```
# load models
model_yaml = open('../checkpoints/model.yaml','r')
model = keras.models.model_from_yaml(model_yaml.read())
model_yaml.close()
model.load_weights('../checkpoints/model.hdf5')

vision = model.layers[1]
counter = model.layers[3]
detector = model.layers[4]
```

8.2 Load Input Data

Input data is stored in a folder inside input folder of current directory"../input/custom/".Data is Loaded and sent into model to predict.

```
Ximg = loadImages('../inputs/custom/')
Xs = np.array([standardize(x) for x in Ximg])

h = vision.predict(Xs)
ycount = counter.predict(h)
ycount = np.argmax(ycount, axis=1)

ylabel = []
for i in range(len(ycount)):
    # generate range for each count
    indices = np.arange(ycount[i])
    # one hot encoding for each index
    indices = np_utils.to_categorical(indices, max_digits)
    # tile h to match shape of indices matrix
    hs = np.tile(h[i], (ycount[i],1))

    # predict labels for the sample
    sample_seq = detector.predict([hs, indices])
    sample_seq = np.argmax(sample_seq,1)
    ylabel.append(sample_seq)
```

8.3 Prediction

For prediction, we will first generate the intermediate output h, from vision model. We will then pass it to counter first. Then the detector will be called for each sample with all indices in one go.

The crucial part here is that we want to calculate the intermediate output h only once to save on computations.

```
h = vision.predict(Xs)
```

```
ycount_ = counter.predict(h)
ycount_ = np.argmax(ycount_, axis=1)
```

```
ylabel_ = []
for i in range(len(ycount_)):
    # generate range for each count
    indices = np.arange(ycount_[i])
    # one hot encoding for each index
    indices = np_utils.to_categorical(indices, max_digits)
    # tile h to match shape of indices matrix
    hs = np.tile(h[i], (ycount_[i],1))

    # predict labels for the sample
    sample_seq = detector.predict([hs, indices])
    sample_seq = np.argmax(sample_seq,1)
    ylabel_.append(sample_seq)
```

8.5 Evaluation

We can evaluate the performance of both counter and label detector to get a better insight on the fit. In the end, we will evaluate the performance of the whole system. We will consider a predicted sequence to be correct only if all labels have been identified successfully, as any wrong classification can result in a totally different house number.

Detector Metrics

Here we need to presume that counter has made a perfect prediction. If the counter itself has predicted wrong count, then it shouldn't be counted as detector's failure. So we'll evaluate the detector's performance for digits in a sequence till an index such that index is the minimum of true count value and predicted count value.

```
ycmin = np.minimum(ycount, ycount_)

# extract labels from ylabel and ylabel_ using ycmin
ylabel_det = np.array([ylabelrow[0:ycminc] for ylabelrow,ycminc in zip(ylabel, ycmin)])
ylabel_det = np.concatenate(ylabel_det)

ylabel_det_= np.array([ylabelrow[0:ycminc] for ylabelrow,ycminc in zip(ylabel_, ycmin)])
ylabel_det_= np.concatenate(ylabel_det_)
```

```
print classification_report(ylabel_det, ylabel_det_)
```

	precision	recall	f1-score	support
0	0.77	0.88	0.82	1691
1	0.82	0.89	0.85	5031
2	0.83	0.87	0.85	4108
3	0.81	0.79	0.80	2831
4	0.88	0.82	0.85	2476
5	0.89	0.79	0.83	2345
6	0.82	0.78	0.80	1948
7	0.81	0.83	0.82	1989
8	0.81	0.75	0.78	1624
9	0.85	0.78	0.81	1572
avg / total	0.83	0.83	0.83	25615

Overall Sequence Prediction Accuracy

The overall accuracy on the test images which are around 13000 thousands we got accuracy of 68%.

```
def matchSequence(seq, seq_):
    return [np.array_equal(seqi, seqi_) for seqi, seqi_ in zip(seq, seq_)]
seqmatch = matchSequence(ylabel, ylabel_)
print "Sequence prediction accuracy : {}".format(np.average(seqmatch))
```

```
Sequence prediction accuracy : 0.682812978268
```

Result:

Model Evaluation and Validation

The final model which was described previously was chosen on the basis of validation and testing accuracies. It was observed and is understandable that higher accuracies of both counter and label detector did in fact translate to a higher sequence transcription accuracy on the test set. The final model achieved the following accuracies after training for 50 epochs:

Model	Training	Validation	Test
Counter	94.21%	92%	86%
Detector	87.73%	87.28%	83%
Sequence	-	-	68.28%

Following are some image samples with the predictions made by the system. Values in brackets are true values, and red indicates a wrong prediction.

As can be seen from accuracy metrics and samples above from test dataset, we can say that the model has done fairly well on unseen data.

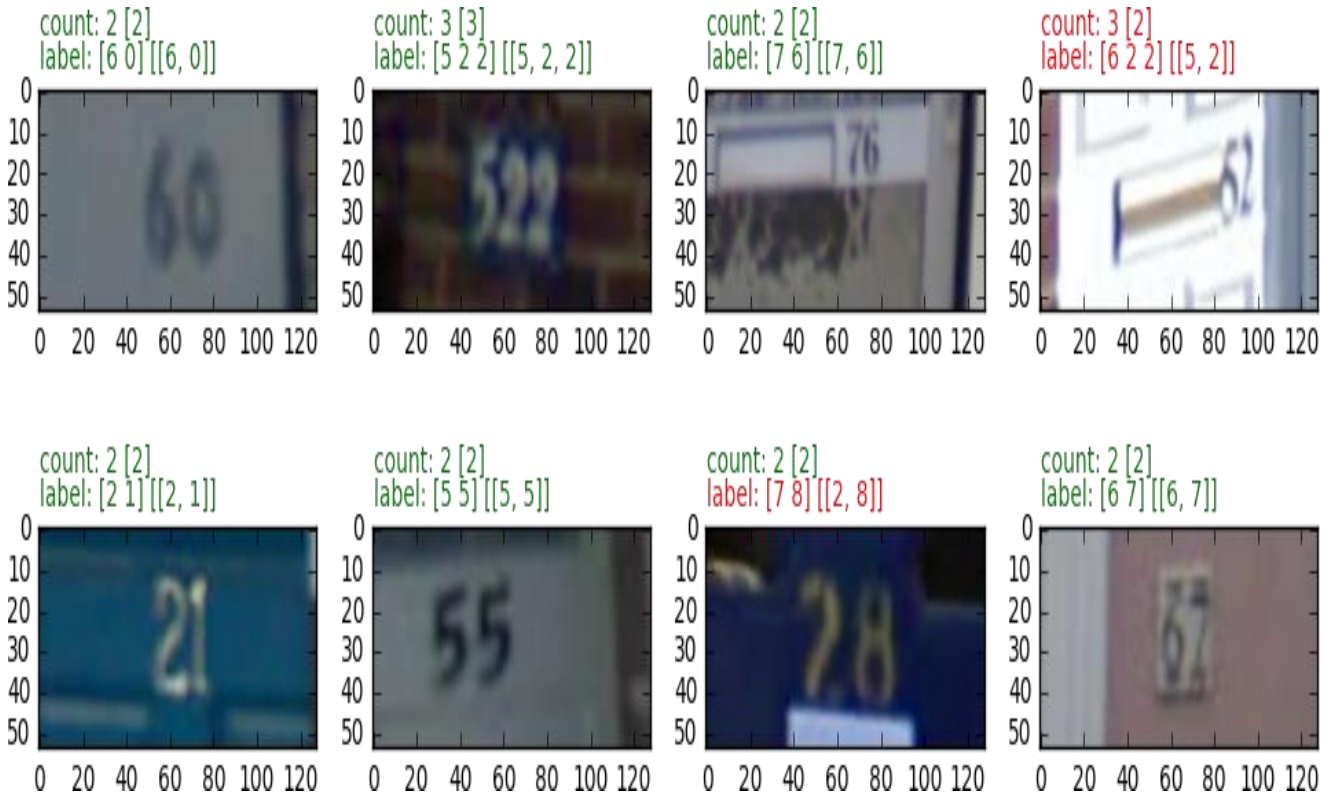


Figure 11 : samples with prediction from test set (values in brackets are true and red shows a mismatch)

Conclusion

Free-Form Visualization

Using Keras functional api, we can actually see the convolution features at different layers in the vision model for a given input. The convolutions of the first two convolution layers are shown for a sample input.

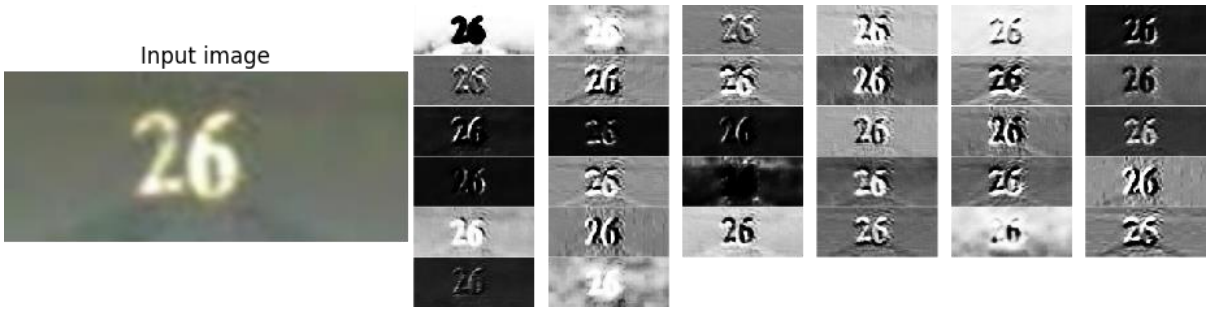


Figure 12 : convolution layer 1 features for a sample input

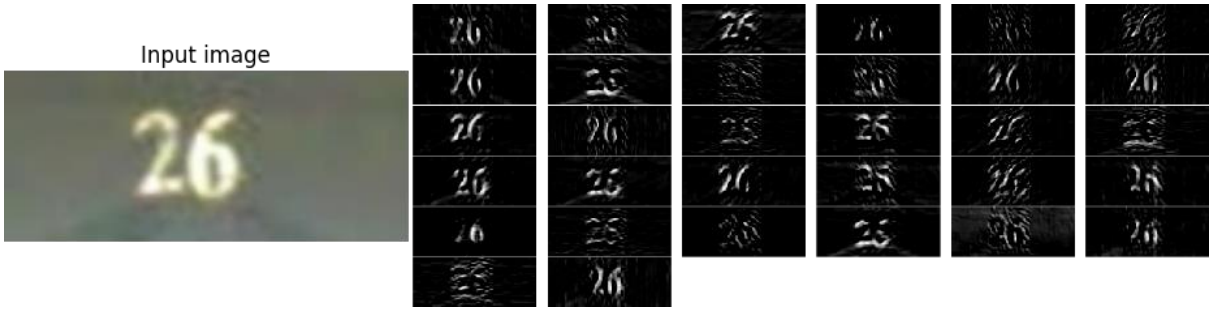


Figure 13 : convolution layer 2 features for a sample input

Interestingly, the convolution features appear to be results of edge detection filters from different directions. This tallies with literature explaining internals of CNNs. We can also see that in layer two, the CNN was able to reject most of the noise and bias and had prominent digit edges, which would in the end be combined together to affirm presence of a particular digit.

Reflection

To summarize, a convolutional neural network based solution was designed to solve the publicly available SVHN dataset. The solution we designed comprised of three macro models, a shared vision model, a counter and a label detector. During training, the counter and label detector were trained with independently true data, which is a flattened version of SVHN training dataset. But during prediction, the label detector depends on counter output.

The architecture of this solution was a challenge to implement because of slightly different training and testing graphs. It was tried in both Tensorflow and Keras. Saving and loading the model weights and graphs was especially challenging in Tensorflow so Keras was used in the end because of its high level functional api and ability to directly use numpy arrays between models.

The biggest challenge we encountered was overfitting. Dropout layers were of very little help. It was eventually figured out that image augmentation does exceptionally well against overfitting. In our final model, we encountered little to no overfitting. It is perhaps the biggest lesson learnt from this project.

We also learnt that batch normalization significantly boosts the performance of deep learning networks. Even though training with batch normalization is much slower per epoch, we were able to achieve better results in much fewer epochs than without it.

The final solution that we developed did meet our expectations. There can definitely be improvements, but the solution developed can in fact be used in the same general setting. For example, we can also develop an English word reader using the same methodology.

Improvement

The rescaling step that is performed as part of preprocessing does result in loss of detail. If the digits are quite small in size compared to the image dimensions, then rescaling the image can render the digits completely unrecognizable. If we develop a localizer that detects and crops the digit sequence in an image, then we can pass sequences to our models in much more detail. For example, see the results of an original image from test set and a cropped section of it passed to our solution below.

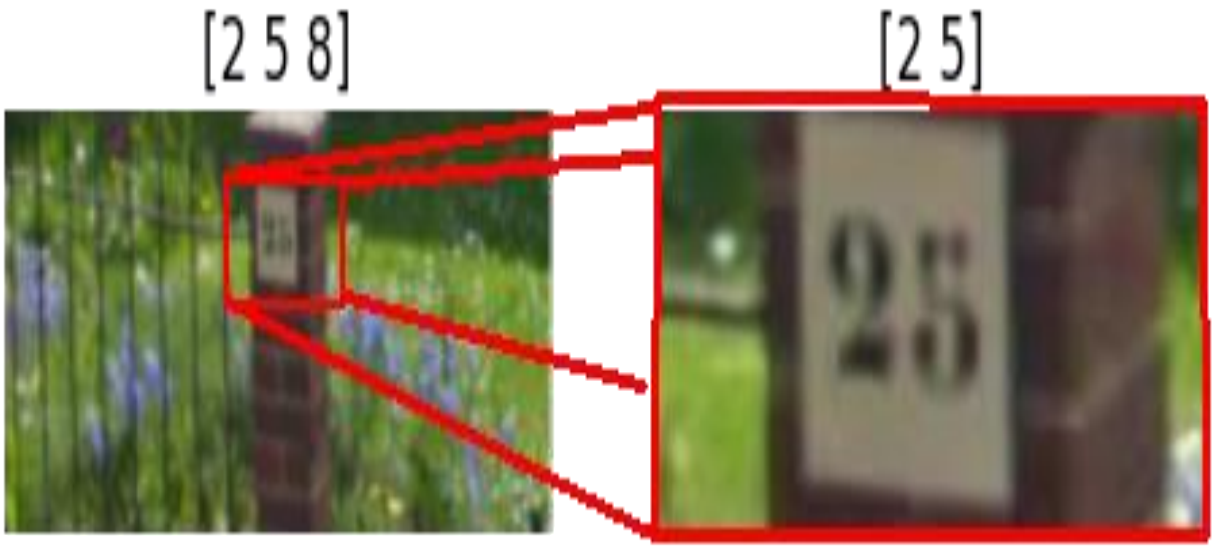


Figure 14 : original vs cropped image passed to prediction network

We can also try increasing the number of layers in our models, as suggested in the reference. If the network is trained for more time, we can expect the losses to further go down.

References

[1] Y. Netzer and T. Wang, "Reading digits in natural images with unsupervised feature learning," *Nips*, pp. 1–9, 2011.

[2] T. Wang, D. J. Wu, A. Coates, and A. Y. Ng, "End-to-end text recognition with convolutional neural networks," *ICPR, Int. Conf. Pattern Recognit.*, pp. 3304–3308, 2012.

[3] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. Shet, "Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks," *arXiv Prepr. arXiv …*, pp. 1–13, 2013.

[4] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *Arxiv*, pp. 1–11, 2015.

[5] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, 2014.

[6] S. C. Wong, A. Gatt, V. Stamatescu, and M. D. McDonnell, "Understanding data augmentation for classification: when to warp?," *DICTA*, Sep. 2016.

[7] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *Int. Conf. Learn. Represent.*, pp. 1–13, 2014.

[8] Q. Guo, D. Tu, J. Lei, and G. Li, "Hybrid CNN-HMM Model for Street View House Number Recognition."

[9] D. Stutz, "Understanding Convolutional Neural Networks," *Nips 2016*, no. 3, pp. 1–23, 2014.