The effect of noise when training a network.

Deliverables

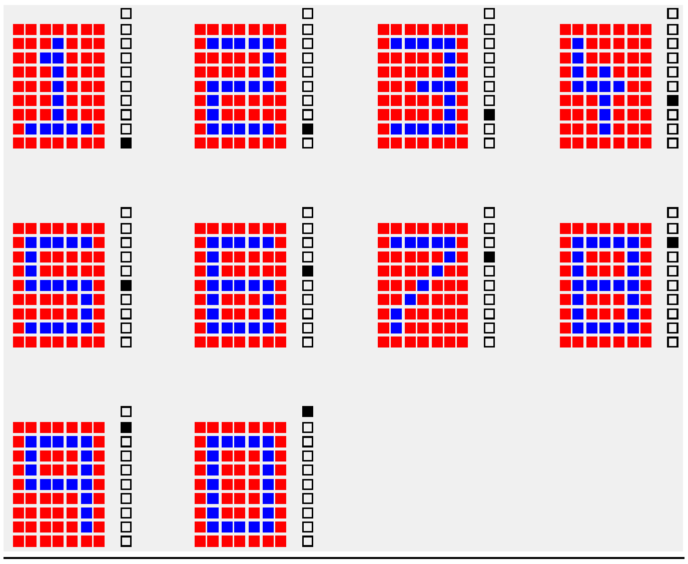
1. Report explaining what you did, your results, and what you believe the effect of noise is
(Both for testing and for training)

2. The code that trains and tests the network (Don't forget to comment so that I can understand what is what)

Task 1:

One of the most interesting capabilities of Artificial Neural Networks (ANNs) is their ability map complex linear or non-linear functions across vast multi-dimensional space. A useful feature of ANNs is their ability generate complicated classification decision boundaries when exposed to incomprehensibly large amount of data. Due to this ANNs in recent years have been applied extensively to image recognition classification tasks.

The Current task requires the creation of single layer of neurons capable of recognising a given series of images. The model will be then trained and the effects of noise on the input data will be explored.



Visualising the data

The image to the left represents the target data we hope to replicate in our model. As you can see the trained model is able to represent each digit using a one hoc (binary sequence activation,[00001] = 5) encoding method, where by only a single neuron activates relative to a certain input. Overall We can visualise the data in terms of a matrix with each cell of the matrix corresponding to a pixel intensity value. Considering the image to the left we can view this as a greyscale image with the background being uniformly red we can say its pixel value = 0, whilst the pixels that encode the contrasting information or useful image data will have a pixel intensity of 1. Abstractly to the human each the display pattern of 1s and 0s in the image matrix will form a shape similar to the digit shape when the image is shown normally. The aim is to connect each pixel data point to each of the ten neurons in the single layer. As Shown Below:

Each Connection will be ‘Weighted’ with randomly unutilised value, that will abstractly represent the synaptic strength of the data point to the specific neuron. Each neuron will represent a stream of input data respective to its weights by calculating its internal state via a summation and activation function. Summation take in the input value for a given pixel and then scale intensity up by the factor of its associated weight. The sum of every input pixel and weight will then be fed through an activation function which will determine whether a specific neuron should activate or not. In this case the activation of the model layers is the output, but in other use cases one could feed the activations of the network forward to several further layers to increase the level of abstractions the model can perform on the data. The output of our network will represent probabilities, in this case its simple a neuron is either certain a specific digit is there or not (1, 0). We can then calculate the layers error by calculating the difference between. Finally we will calculate the error of our model compared to the above target data, and update the networks weights accordingly. Over enough iterations the network will establish the correct functional parameters to map the input to the correct digit prediction.

Please see below:

Programmatic implementation

```
% DEFINE DATA & TARGET

%Flatten Input From Matrix To Vector Sequence
input = x;
target = t;
noise_store = [];
TotalBatch_Error = [];
```

Batch_size as the images are processed in one matrix, we are iterating through batches of ten images per epoch. Save_mode allows the weights of the last run to be saved, if set to 0 the weights will not be overwritten. Init_mode = unitise weights from saved variable

```
%parameters

batch_size = 35; %10 epochs neural net computes 10 examples 10x10 matrix  per batch
learnRate = 0.1;

save_mode = 1; %0 = no save to disk, 1: save weights, 2: save to disk
data_type = 0; % 0 = No Noisey training data // 1 = Noisey Training Data
init_mode = 0; %initlise weights with stored weights = 1

noise_switch = 0;
```

Use rand to generate the neuronal layer weights, rows represent weight connects, columns represents the 10 neurons

```
%Define Weights 40

if (init_mode == 1)

    disp('***TRAINED: NO NOISE: INITILISE SAVED WEIGHTS: ENABLED***')
    w1 = no_noise_save_store;

elseif (init_mode == 2)
    w1 = Noisy_save_store;
    disp('***TRAINED: NOISEY: INITILISE SAVED WEIGHTS: ENABLED***')
else
     disp('***INITILISE SAVED WEIGHTS: DISABLED'
     w1 = randn(63, 10);
     save = 0;
end
```

For loop to iterate through a specified training phase. N(i) generates a random noise percentage to prohibit overfitting to data at specific noise percentage.

```
%Define Netwrok Layer and Calculations

for i = 1:batch_size


 n(i) = randi([0 0], 1, 1);
 noise_store(i) = n(i);
 NewInput = addNoiseBinary(input,n(i));
```

Performs matrix multiplication on data and weights = summation, this result is feed through the step function which returns a 1 result is greater than or equal to 1, else a 0 is returned.

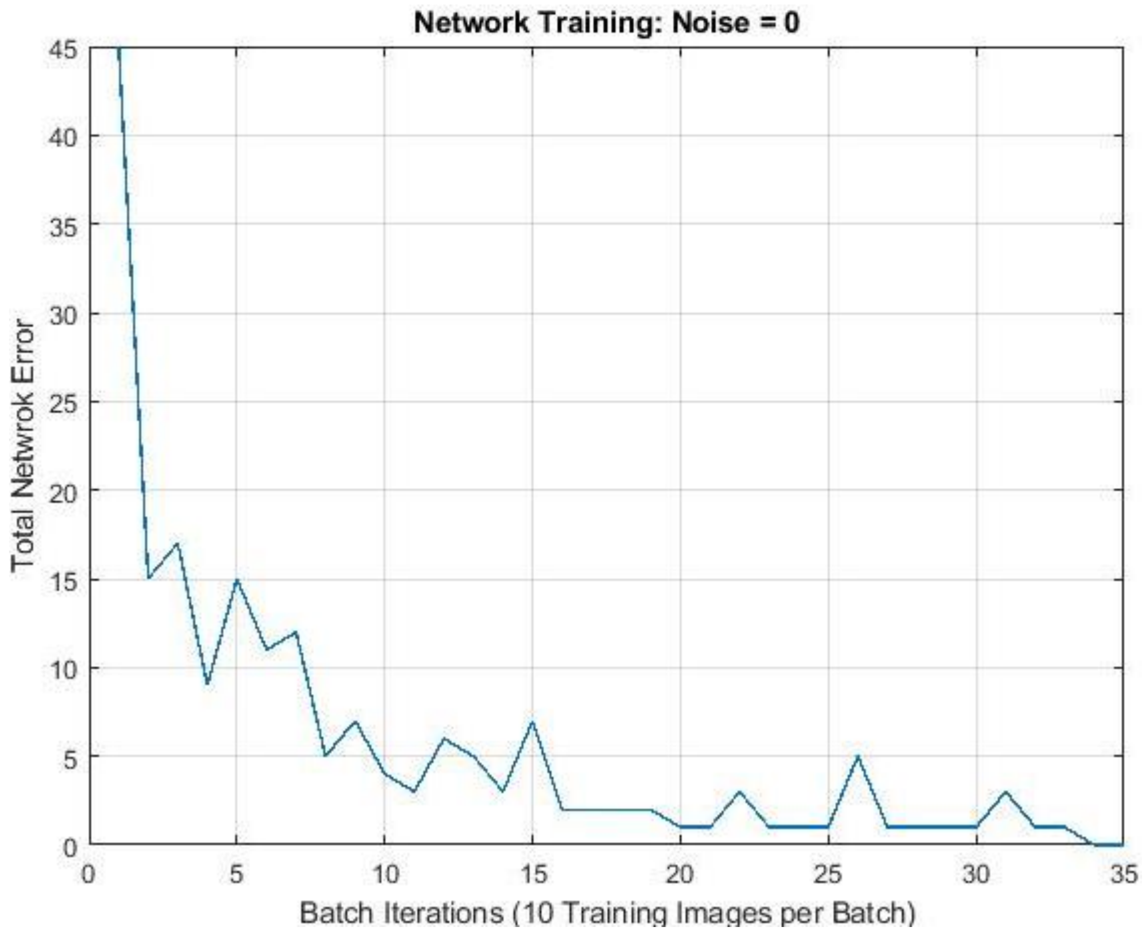
```
 neuron_layer_sum =  NewInput* w1; %Summation
 outputs = StepFunction(neuron_layer_sum);
 %activation function Sigmoid
 % Calculate Error
```

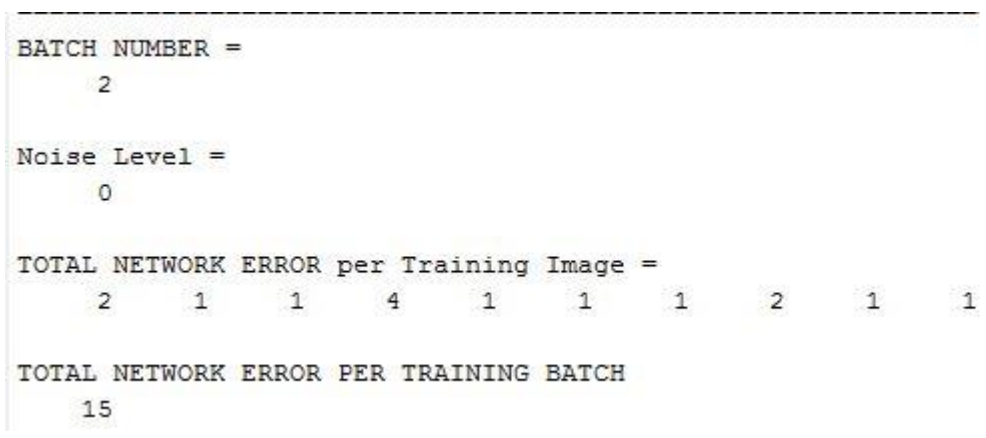
Calculates The Error between the network output and the target labels. The delta is calculated which informs us on how we should update the network weights to produce a result closure to the target

```
  Error = target - outputs;
  delta = learnRate*NewInput'*Error;

  %Error Measure

  for e = 1:length(Error)
     TotalError_perImage(e) = sum(abs(Error(e, :))
  end
  TotalBatch_Error(i) = sum(TotalError_perImage);
  %update weights
```

Finally we update the weights using the delta value

```
  w1 = w1 + delta;


  disp("==================================================================")
  disp("BATCH NUMBER =")
  disp(i)
  disp("Noise Level = ")
  disp(n(i))
  disp("TOTAL NETWORK ERROR per Training Image =  ")
  disp(TotalError_perImage)
  disp("TOTAL NETWORK ERROR PER TRAINING BATCH")
  disp(TotalBatch_Error(i))
  disp("==================================================================")
end

if (save_mode == 1)
    if (data_type == 0)
        %Save No noisey
        no_noise_save_store = w1;
    elseif(data_type == 1)
         %Save to Noisey Save store
         Noisy_save_store = w1;
```

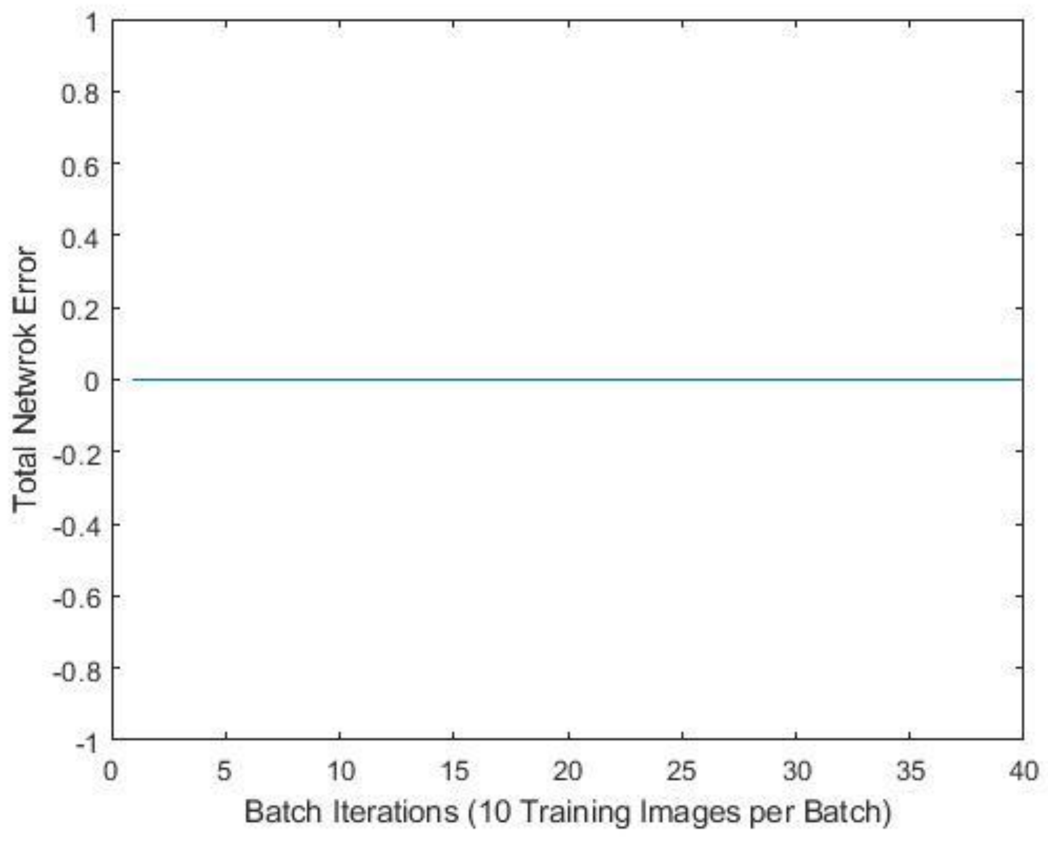




```
----------------------------------------------------------
BATCH NUMBER =
     2

Noise Level =
     0

TOTAL NETWORK ERROR per Training Image =
     2     1     1     4     1     1     1     2     1     1

TOTAL NETWORK ERROR PER TRAINING BATCH
    15
```

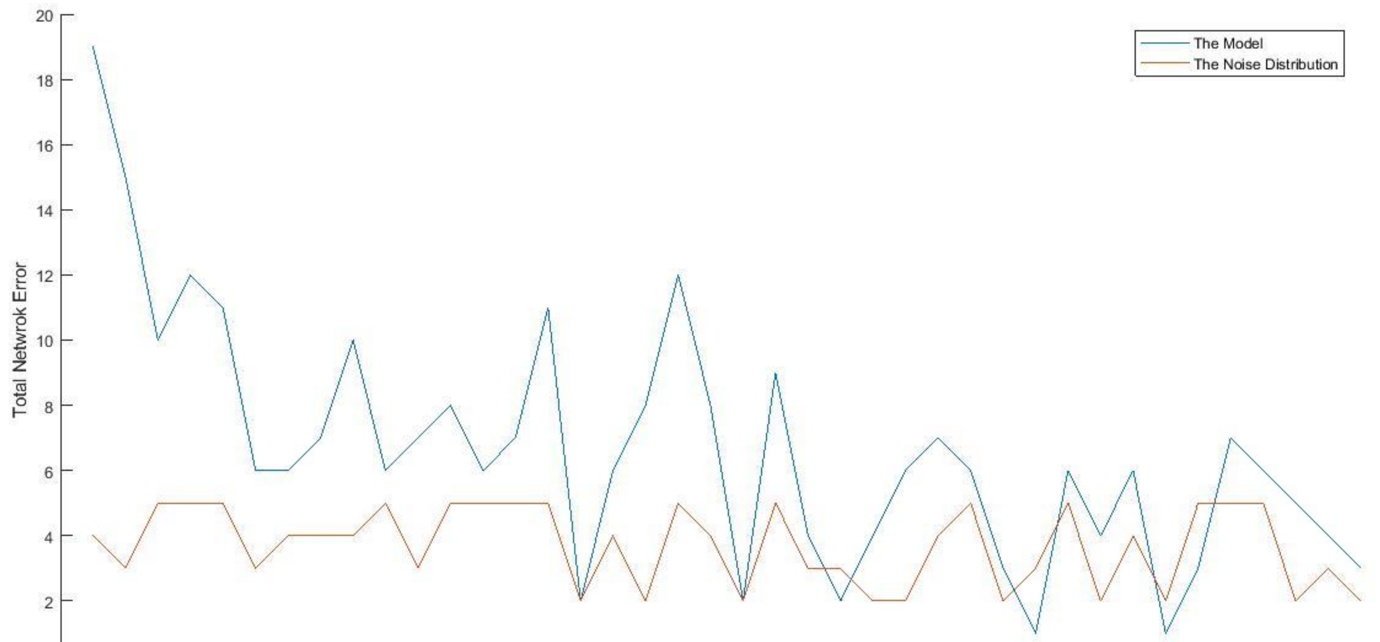
The Above graph Shows the learning progression of the network. Error as expected begins high due to weight parameters being random, as the number of iterations increase the network is able to more accuratley map between the input data and its output relative to the target data. Learning occurs rapidly in the first 10 iterations and then slowly decends for another 24 iterations until converging to an error value of 0. This indicates that the current model is trained and will produced the currect label for each digit image presented.

For clarity Error was measured by calulating the differences between the binary matrix of targets and the binary matrix of outputs. This results in 3 possible differences the output can have from the target [1, 0, -1], whereby 0 indicates spesfic neuron fired correctly, 1 the neuron fired when it should not have, -1 the neuron didn’t fire when it should have. Because training occurs in batches of 10 the TOTAL NETWORK ERROR respresents the cumplitive error of the network per image, per batch. The individual errors of each neuron was summated using the absolute values to allow for the minus sign of indivisual neuron error to not effect the true error result. For example given the above image it shows that for the first image (far left digit), 2 neurons in the network responded incorrectly.

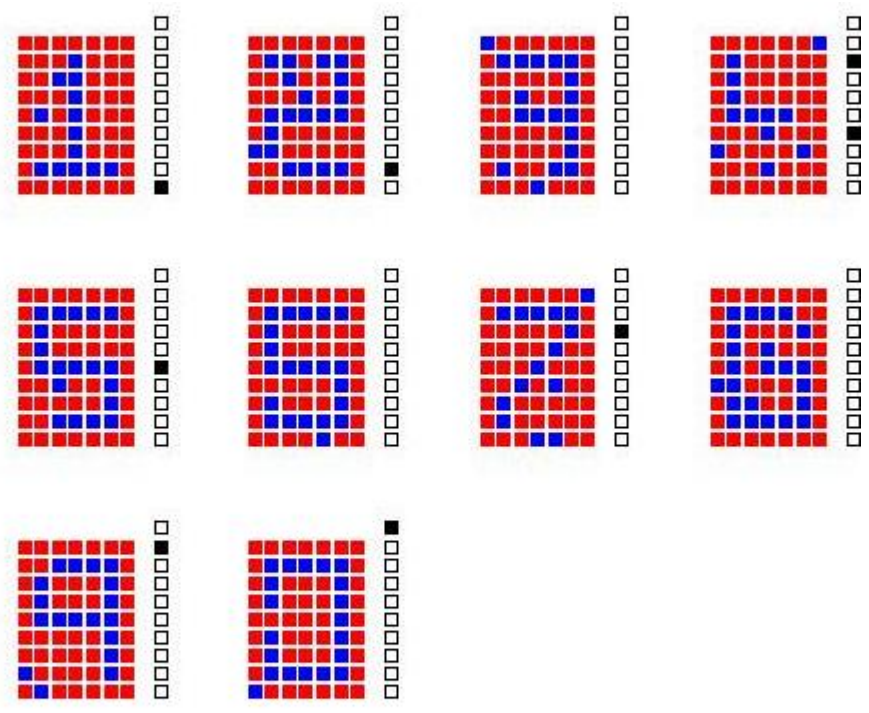


The Above graph provides evidence that the network is indeed trained, as it outputs 100% accurancy for all iterations with 0 error from the target output.

<u>The Effects of noise Exposure on a trained model</u>

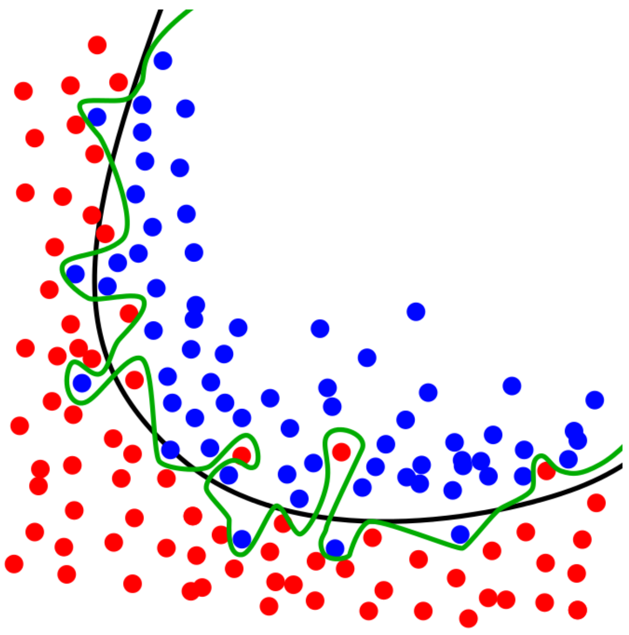


In contrast we see that when the model is exposed distribution of noise between an inesity of 2-5%, we see that the amount of error in the model increased drastically compared to the clean data run it was trained on.

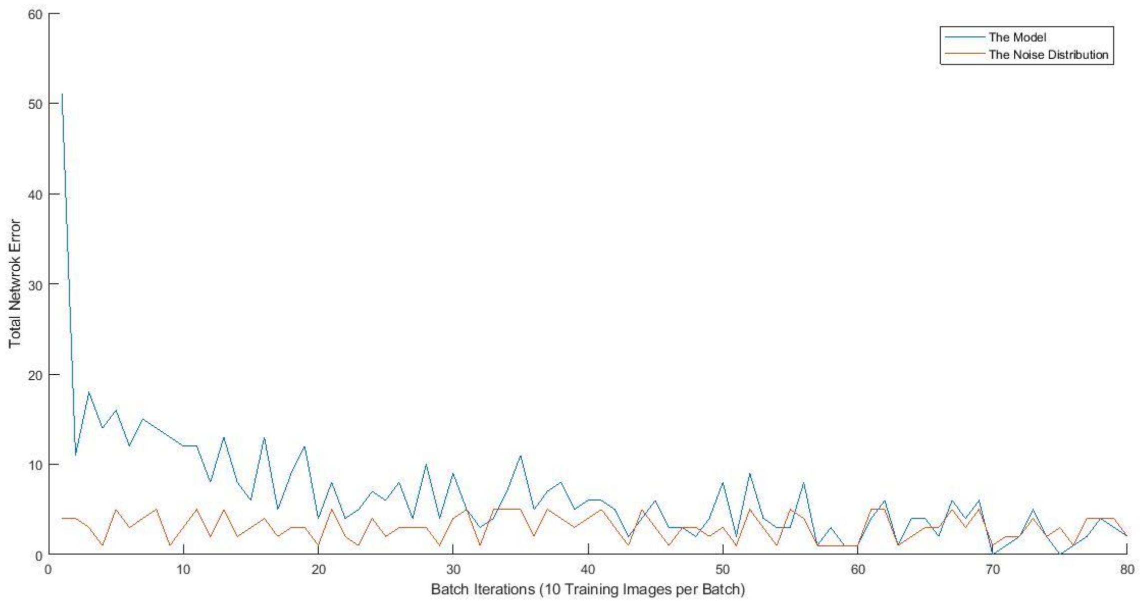


Inspecting a batch individually one can see that the network is finding it difficult to classify image labels. This raises the several questions upon what actually is noise? How does it effect neural works, are their any ways of handling noise?

We can think of noise as a stocastic process by in this case its percentage intensity range is randomly selected. As previously stated neural networks traverse data-space and try to obtain the best descion boundry in order to distiguish between clusters ofdata points. Each cluster of data points will have a high associative probability of belonging to a certain label E.g the digit ‘5’. The more randomability that can occur on the input stream increases the probability of more data-points becoming spread out across data space, forming fewer clustered structures. Therefore the probability of a datapoint that belongs to a 5 label now becomes close enough or cross the descion boundry and is now classfied as a diiferent digit. We can see this occuring in the above data output graph, the 4th image we see the 4th neuron firing correctly but we also see the 7th neuron firing, it two classifying this imagine to be 7 and not a 4. Consider graph below and the model that was just trained to produce 100% accurate results when exposed to a clean input stream.



The orginal model used in this report was very well trained to classifying a clean input stream, we saw on the previous page that it generated a 1 dimensional error space of 0 for each input image. Like the diagram to the left the model has ocerfitted to tis spesfic instance of the input stream, it may explain this spesfic data very well but is equally very rigid and senistive to deviations of the data points from its current formation. This due to the fact that more input streams a network samples from will contain a degree of randombability thus its very inprobable that an input stream will be the exact same orientation coninously. Therefore with a rigid decision boundry its more likely produce higher error values. The next logical question would be, how does one make the descion boundry more flexiable to prevent overfitting and promote higher genralisability of the data. One possible method when considering differentiable functions would be to add a bias node to each layer in order to allow the descion boundry to move more freely. For this given example one could simply train the network with added noise in order to model a sample estimate on the randombability the model may encounter. The results are as follows:

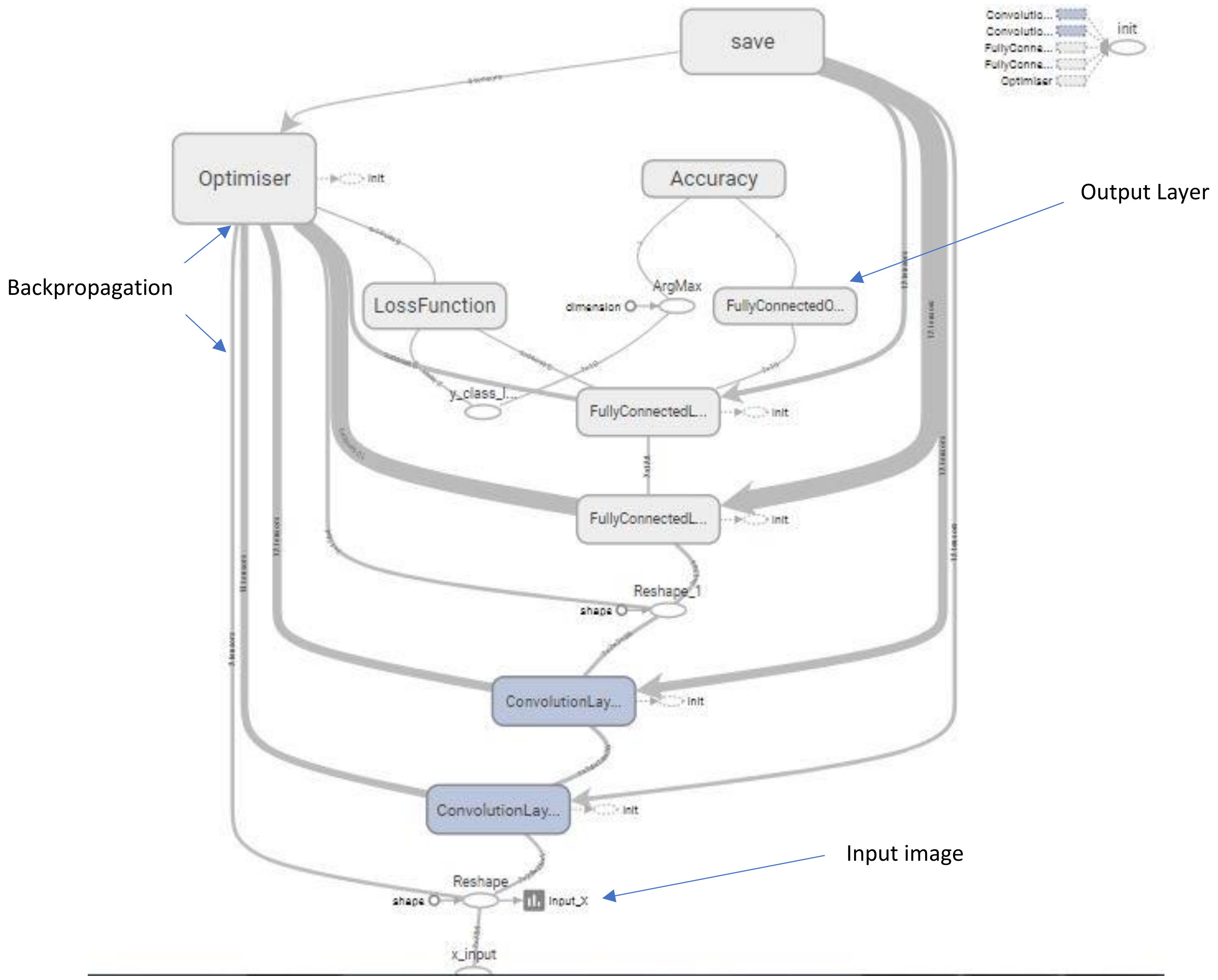


The yellow line indicates the level of noise whilst the blue shows the error of the model. It is clear that the training a model with a noise distribution does produce results in which the error is much lower given the noirse compared to when a clean trained network was gneralised to a noise input stream. However it does take many more iterations for the error to decrease to reltively tolarable value.

In conclusion noise can be useful in training the model to be more robust, however thjere are other possibilties that provide more robust approuches to handling the stocastic nature of noise. In the next section I will discuss my code implementation for a convolutional neural network that is capable of classifying the noise against the relevent information. In this way the model not only is creating a descion boundry for the input image data points but it is also creating decision boundry between the overall true input data stream as whole and the input stream of adversarieal noise.

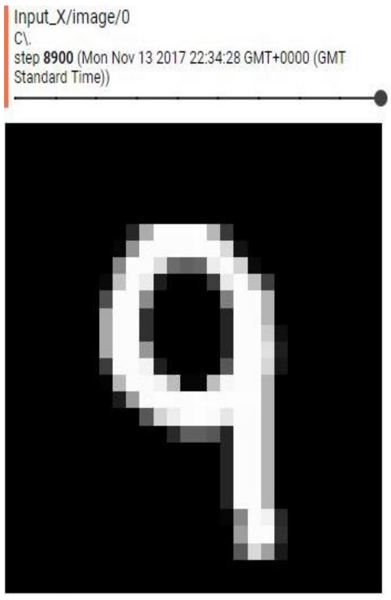
How to handle noise: Convolutional Neural Network.

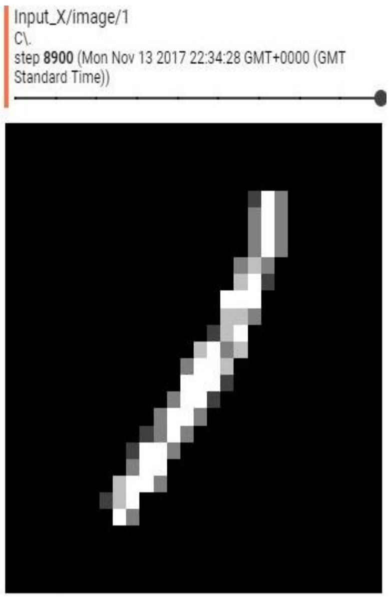
Now I would like to delve into the realm of non-linear multi-layered network architectures and explain how one can implement L2-loss regularization protocols that allow a network to become immune to noise, with minimal extra training and loss of cleanly trained weight settings. I developed a 4 layered convolutional network, with 2 convolutional layers and two fully connected layers using tensor flow(python package). The code itself is hundreds of lines long, therefore I will use crucial code snippets and the tensor-board(visualisation package) to explain the effects of Adversarial noise on convolutional layer using the mnist dataset: The Network Framework can be detailed below:



This section focuses on handling noise, using a model that is robust to the type of noise that is likely to occur for the given tasks, when it comes to image recognition convolutional neural networks (CNNs) are a sensible architecture. In the first one layered model we saw that although training with some noise can benefit a models adaptivity, there is always limitations to how much input noise a network can train with whilst still producing any meaningful results. The computational structure and processes of CNN provide some very useful process that allow the model to be more robust to increased input image complexity and thus noise.The following benefits are:

- The first convolutional layers do not require dimensionally flat image data, but in fact can receive multi-dimensional tensors as inputs, for example 32pixel * 32pixel * 3. The 3 represents the RGB value or colour value of an image for the red, green and blue light spectrums. Thus tensors can process through colour data, which in some image recognition cases could be in itself an input noise contribution.
- Low to high level abstraction through the convolution of multiple feature or activation maps. CNN's employ filters(kernel) that slide across the height and width of a pixel matrix of one of the RGB dimensions. The dot products are then computed between filter and the input matrix at a given position to create map of activation intensity response for the filter at every spatial location on the image. Consequently, each filter will learn to activate when a certain type of visual feature occurs in its given receptive field (relative the sliding position of the filter). Each filter layer contains many filters, and each produces a feature map that are then stacked to produce an output tensors that are then passed to the next convolutional layer. The first convolutional layer provides low level processing and will generate a feature map of edges. These maps of edges are then convoluted upon by a second layer, which is able to identify more complex feature patterns. It is at this layer that clusters of edges can become abstracted to shapes, clusters of shapes can then be abstracted towards the overall encapsulating pattern structure (the overall image) of image data sub-components. Long story short these Filters are spatially invariant via layered abstraction. This can be explained using the MNIST input images for training my CCN image classifier.


Input_X/image/0
C\.
step 8900 (Mon Nov 13 2017 22:34:28 GMT+0000 (GMT Standard Time))


Input_X/image/1
C\.
step 8900 (Mon Nov 13 2017 22:34:28 GMT+0000 (GMT Standard Time))

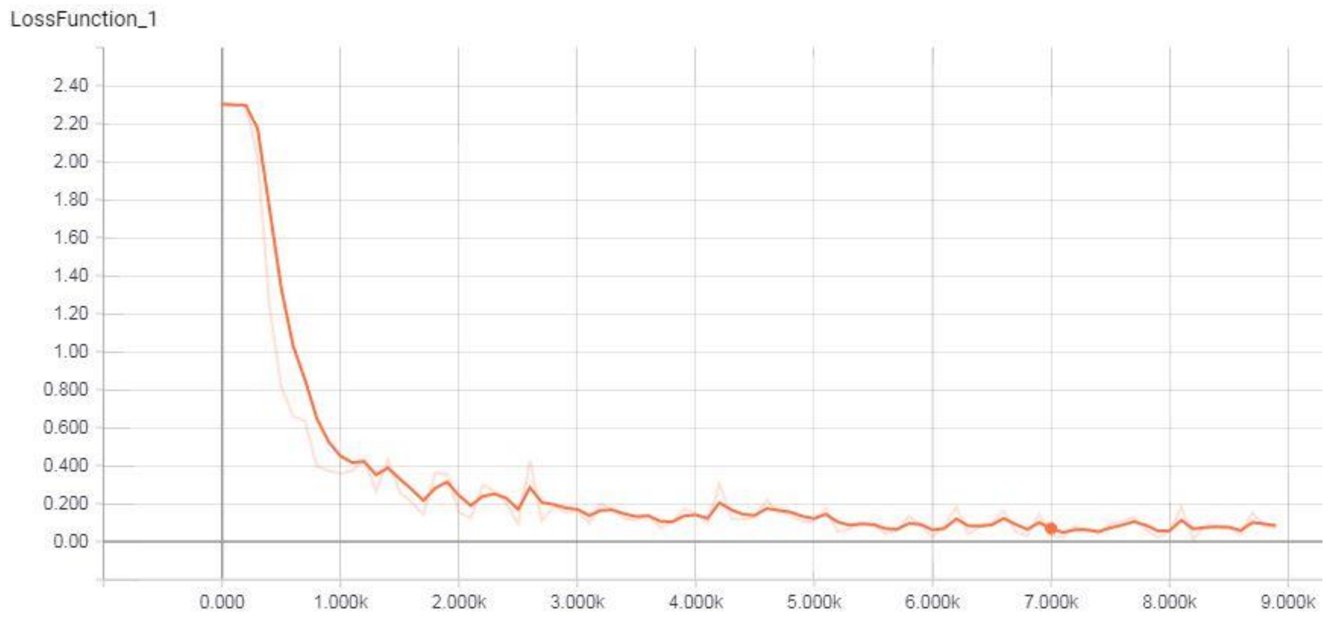
The used dataset was comprised of 5500 hand written images of numbers from 0 to 9. Like with the noise added to the simple neural network input image the depiction of a number can be written differently with thousands of possible orientations. A CNN is spatially invariant to the spatial variation of representing the same data label. This due to abstraction as the model learns to associate general structural patterns by combining decreasing levels of sub-components. For example, the number 9 in the left image will contain thousands of variations, but all 9s will follow the general pattern of being constructed by circle shape on the top, followed by a connecting vertical straight-line shape.

Finally it should be noted that Convolutional layers are only locally connected, thus we employ at the end of the a 2 layer fully connected network that takes all of the learned feature maps of the image space and begin to cluster associated data points relative to a functionally determined decision boundary. The final result runs through a SoftMax function to squash final values between 0-1, that represent, via one hoc encoding, the probability of the image belonging to specific digit label.

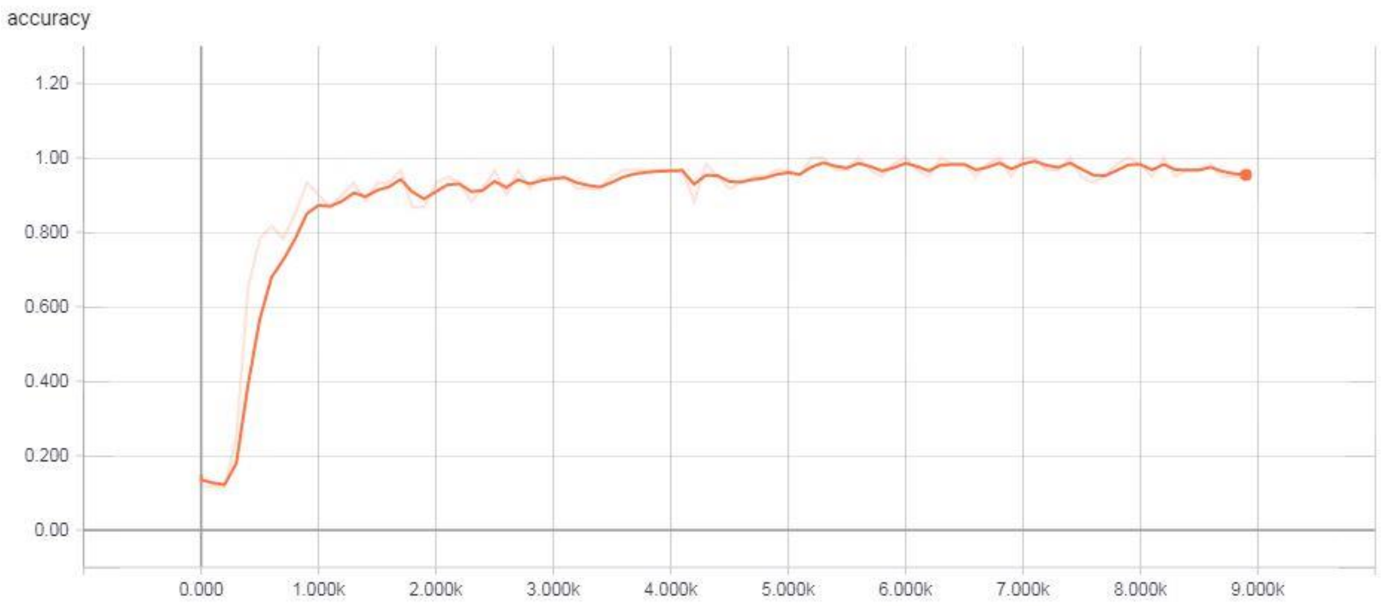
Results:

It can be evidenced that a CNN can perform extremely well when tasked with performing image classification given noise of spatial variance of the input images:

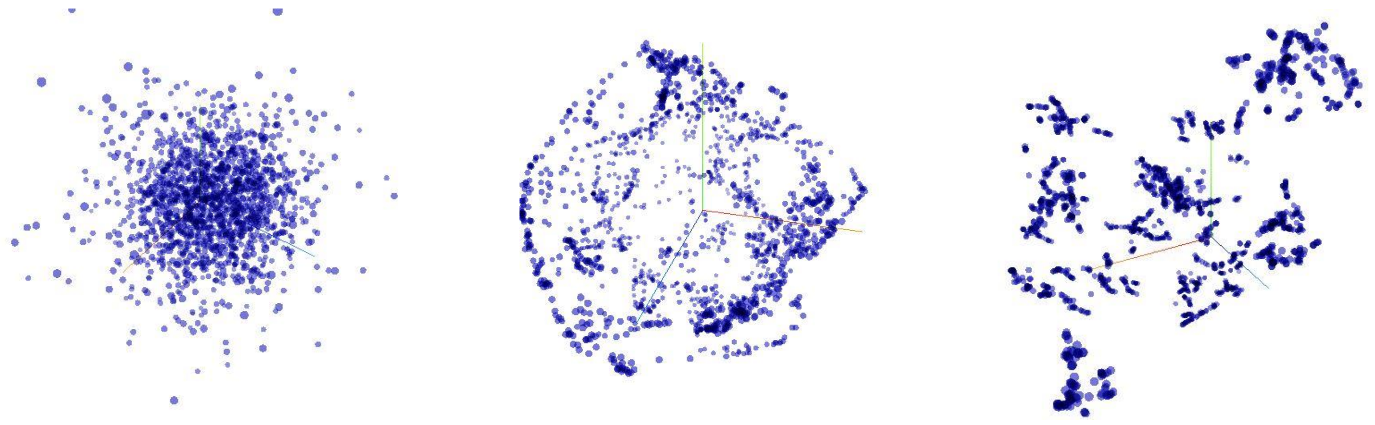
<u>Loss Function: [x = iterations, y=loss value]</u>



Model Accuracy: [x = Iterations, y=probability accuracy value]



The graphs show a steep learning curve in the first 1000 iterations, which eventually levels into low amplitude oscillations in performance. The model shows a general performance of 95-98% accuracy when performing image classification.



Using t-distributed stochastic neighbour embedding it is possible to visualise the inflow of flattened tensors through the fully connected layers of the network as the number of iterations increases. With starting iterations being the left image we can see how the model becomes increasingly better at classifying the input images over time, with associated data points becoming increasing clustered to their respective dataspace coordinates, that in turn corresponds to a digit label. Despite their being noise variance in the spatial depiction of each digit label. Furthermore it should be noted that the CNN is cable of dealing with blurred edge noise, considering the input data images it is clear that unlike the images in simple 1 layered network, the data here does not have definitive edges, but rather has a decreasing pixel intensity. The CNN based on the results is able to reason through this noise via the use of max pooling, which entails only the highest activation value of a filter is transferred to the overall feature map. This ensures that the network only captures the most important information of an image. Thus edge blur equals lower pixel intensity which equals lower filter activations, which finally means, via max pooling, that edge some blur will be discarded from filters.

The Code snippets are below. The Next section will focus on accounting for the effects of adversarial Noise.

Python Tensor flow code: Hyperparameters

```
filter_size1 = 5                #16 : 5 * 5 pixel filters
number_of_filters1 = 16
#conv2
filter_size2 = 5
number_of_filters2 = 36
#final Fully Connected layer
fully_connected_layer_size = 128 #128 neurons
#Input Data
#data = input_data.read_data_sets('data/MNIST/', one_hot=True)
data = tf.contrib.learn.datasets.mnist.read_data_sets(train_dir=LOGDIR + 'data', one_hot=True)
#converting Class numbersto integers
data.test.cls = np.argmax(data.test.labels, axis=1)
#Defining Data Dimensionaility
image_size = 28
#flatten images into 1 dimensional array of 764 pixels
Image_flatten = image_size * image_size
#create a tuple that holds image height and weidth for reshaping
image_shape = (image_size, image_size)
#number of colour channels
num_channels = 1
#amount of Classes, one hot thus one for each digit image thus 10 classes
classes = 10
```

Python Tensor flow code: Builder functions for weights, biases, convolutional and fully connected layers

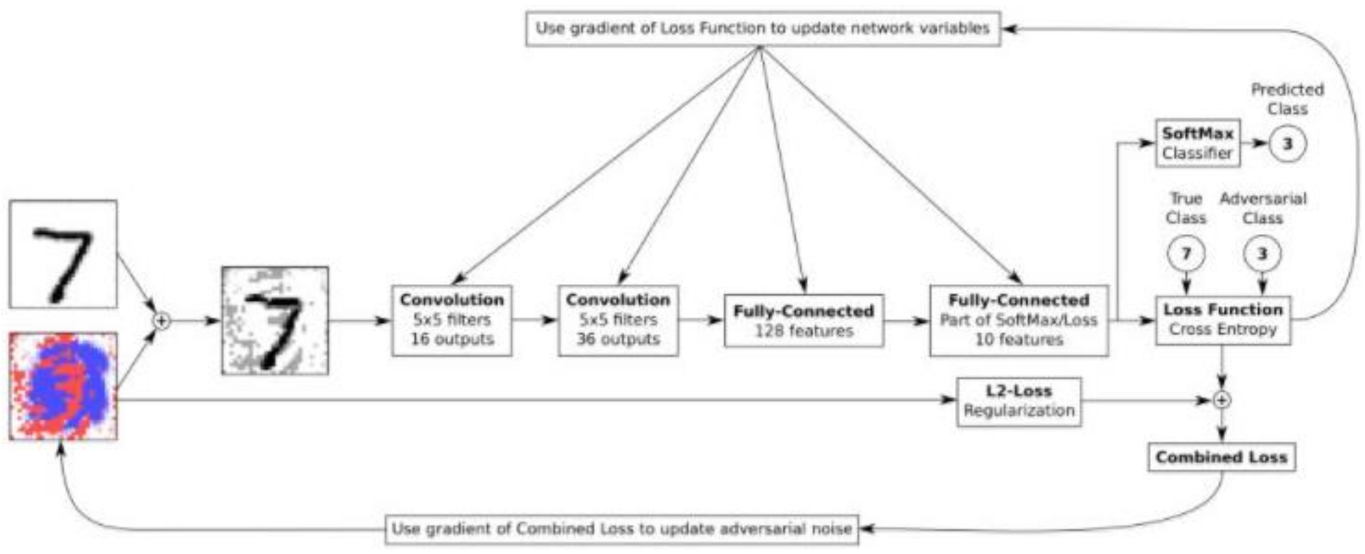
```
def build_new_weights(what_shape):

        return tf.Variable(tf.truncated_normal(what_shape, stddev=0.005)) #Creates random weights with a
spesfied shape = parameter

def build_new_biases(how_long):
    return tf.Variable(tf.constant(0.05, shape=[how_long]))

#Builder Function for convolutional layers

def build_convo_layer(input,
                      number_of_channels,
                      filter_number,
                      filter_size,
                      enable_pooling=True):

    #input = the input can be from the data images or another layer
    #number of channels from previous layers could be filters or from colour streams from input
    #use 2x2 max pooling
    #The below paramets will be optimised

    #Define Shape of filter weights
    filter_weights_shape = [filter_size, filter_size, number_of_channels, filter_number]

    #Init the filter weights with the defined shape
    filter_weights = build_new_weights(what_shape=filter_weights_shape)#matrix

    filter_biases = build_new_biases(how_long=filter_number)#vector

    #Define convolution operation
    layerComputataion = tf.nn.conv2d(input=input,
                                     filter=filter_weights,
                                     strides=[1, 1, 1, 1],
                                     padding='SAME')

    layerComputataion = layerComputataion + filter_biases

#dimesnionalility of input =
# Image number.
# Y-axis of each image.
# X-axis of each image.
# Channels of each image.

#Strides spesficies how our mouch our filter moves accross each dimension of the image
#First and last dimensions are always set to 1 because there is no direction to actuall move in, middle
two are the x, yplane which we can change how many pixels to move by

        #Account for 2x2 pooling Max pooling
        #max pooling means = take the highest value from each 2x2 from each 2x2 window in the filter
        #this way only the most important features are captured by the network
    if enable_pooling == True:
        layerComputataion = tf.nn.max_pool(value=layerComputataion,
                                           ksize=[1, 2, 2, 1],
                                           strides=[1, 2, 2, 1],
                                           padding='SAME')

        #Add Final Actiavtion function for layer output also adds non-linerality to
        #Allow for the network to learn more complex function behaviours
        layerComputataion = tf.nn.relu(layerComputataion)

        tf.summary.histogram("filter_weights", filter_weights)
        tf.summary.histogram("Filter Biases", filter_biases)
        return layerComputataion, filter_weights

        #Convolutional Layers output 4D tensors thus needs flattening to feed into
        #A fully connected layer

        #Define Flattening function

def layer_Flattener(layer_input):
    layer_input_shape = layer_input.get_shape()

    number_features = layer_input_shape[1:4].num_elements()

    flat_layer = tf.reshape(layer_input, [-1, number_features])
```

Python tensor flow code: Optimizer Function

```python
batch_size = 60# chunks 60 images into optimiser at a time so my CPU doesnt Die A Computationaly
painful Death
total_epochs = 0 # This will increase over time

def batch_optimiser(num_iterations):
    global total_epochs # make sure globabl variable is updated and not local one

    start_stopwatch = time.time()#Shows Time usage a new cool function

    #Batch iterations
    for i in range(total_epochs, total_epochs + num_iterations):
        x_batch, y_labels_batch = data.train.next_batch(batch_size)

        #Feed the batches into a dictionary that is feed through the placeholders into the
network

        feed_dict_train = {x_input: x_batch,
                           y_class_labels: y_labels_batch}

        #Tensorflow will auto assign variables for dictionary feed, to placeholders
        #and run the optimiser function

        session.run(optimiser, feed_dict=feed_dict_train)

        #Print Progess every 100 iterations
        if i % 100 == 0:

                summary,accuracy = session.run([summ, accuracy_model],
feed_dict=feed_dict_train)

                writer.add_summary(summary, i)

                message_printer = "Optimization Iteration: {0:>6}, Training Accuracy: {1:>6.1%}"

                saver.save(sess=session, save_path=save_path)
                print(message_printer.format(i + 1, accuracy))

    total_epochs += num_iterations

    #Stop the stop watch
    end_stopwatch = time.time()

    #Caluclate time usage by doing difference between starting time and end time
    time_usage = end_stopwatch - start_stopwatch

    print( print("Time usage: " + str(timedelta(seconds=int(round(end_stopwatch))))))

batch_optimiser(num_iterations=1000)
```

The Effects and protocols for Adversarial Noise

Although there are many initial advantages of using a CNN for image recognition, CNNs like the single layered network are still susceptible to noise. Consider the following diagram:



Use gradient of Loss Function to update network variables
Predicted Class
SoftMax Classifier
3
True Class
Adversarial Class
7
3
Convolution 5x5 filters 16 outputs
Convolution 5x5 filters 36 outputs
Fully-Connected 128 features
Fully-Connected Part of SoftMax/Loss 10 features
Loss Function Cross Entropy
L2-Loss Regularization
Combined Loss
Use gradient of Combined Loss to update adversarial noise

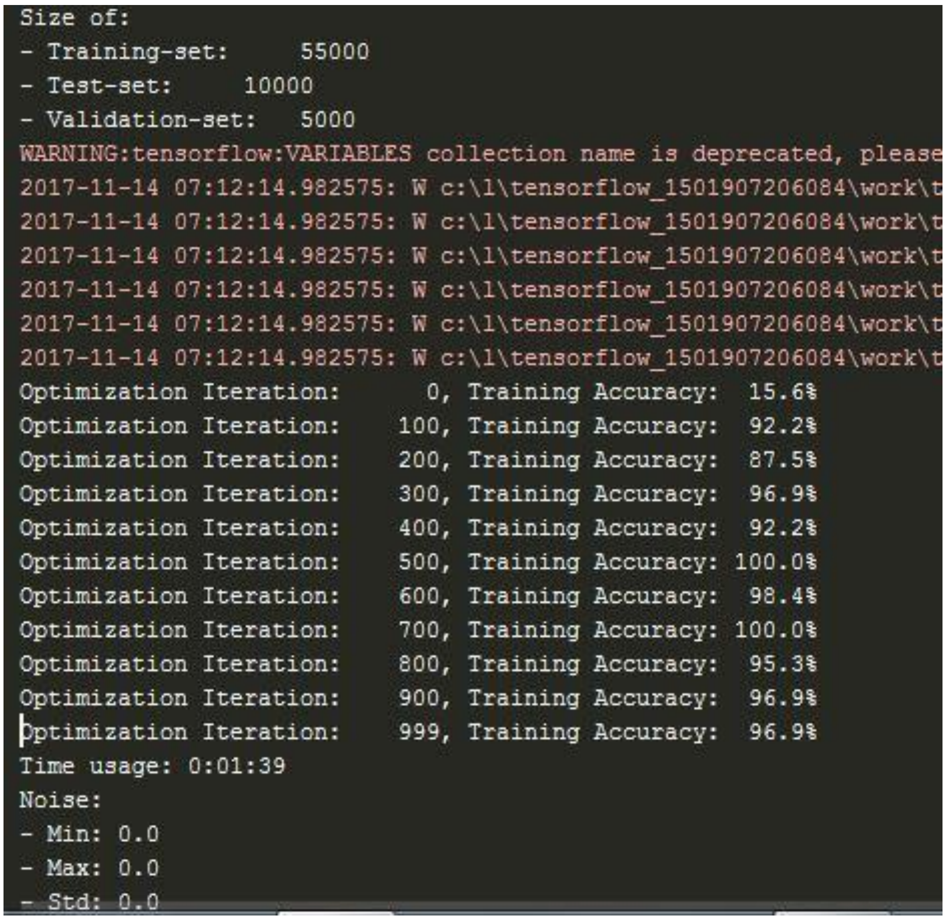
One can add adversarial noise of a specific label class to the MNIST data to generate a noisy input images. As the diagram shows we can add very subtle label class 3 noise patterns to a class 7 input image. Although to the human eye the added noise is noticeable, one can quickly ignore the added noise and still identify the image label as 7. However for a CNN as shown in the diagram such noise can cause the network to misclassify nearly all of the inputted images when fed through the network.:

```
#ADD NOISE
Noise_limit = 0.35
L2_loss_processing_weight = 0.02 # Determines how important L2 regulisation compared to
standard optimisation

#Define noise to tensorflow so noise can be distinguished between optimisers
ADVERSARY_VARIABLES = 'adversary_Variables'
collections = [tf.GraphKeys.VARIABLES, ADVERSARY_VARIABLES]
#Define Noise variable
input_noise = tf.Variable(tf.zeros([image_size, image_size,
num_channels]),name='input_noise', trainable=False, collections=collections)
#Add noise clipping to limit the noise
x_noise_clip = tf.assign(input_noise, tf.clip_by_value(input_noise,-
Noise_limit,Noise_limit))
#Define noisey Image by summing noise and image data
noisey_input_image = x_image + input_noise
x_noisy_image = tf.clip_by_value(noisey_input_image, 0.0, 1.0)
```

After researching methods for adding custom noise to a data set, I was able to find that the following parameters produce the intended noise signal on the input image, in order to reproduce the noisy images produced in the above diagram.

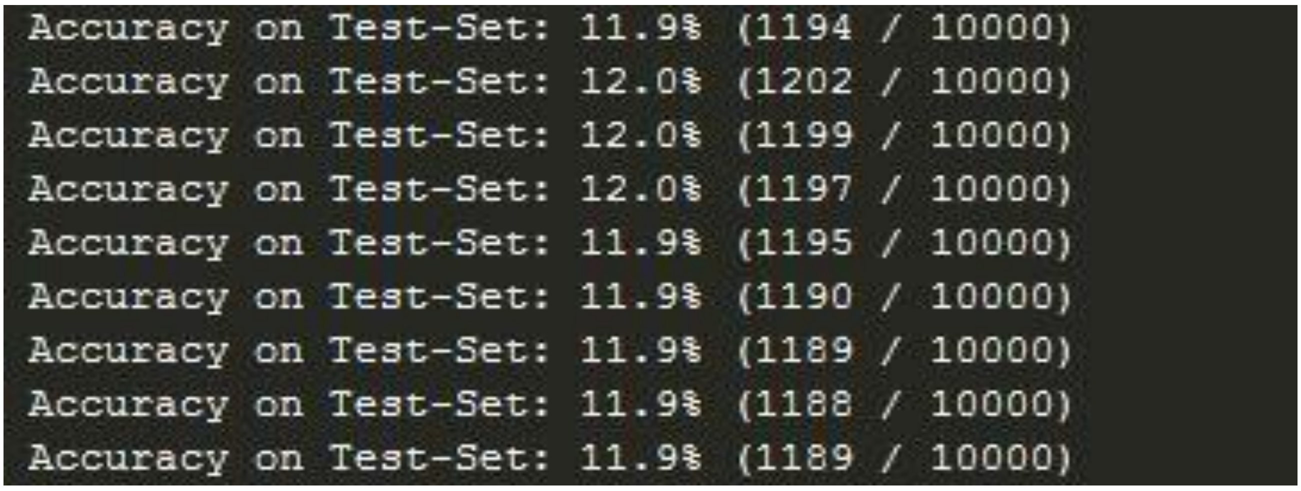
Results: Comparing CNN performance with and without adversarial noise

No Noise training Set



```
Size of:
- Training-set:     55000
- Test-set:     10000
- Validation-set:   5000
WARNING:tensorflow:VARIABLES collection name is deprecated, please
2017-11-14 07:12:14.982575: W c:\l\tensorflow_1501907206084\work\t
2017-11-14 07:12:14.982575: W c:\l\tensorflow_1501907206084\work\t
2017-11-14 07:12:14.982575: W c:\l\tensorflow_1501907206084\work\t
2017-11-14 07:12:14.982575: W c:\l\tensorflow_1501907206084\work\t
2017-11-14 07:12:14.982575: W c:\l\tensorflow_1501907206084\work\t
2017-11-14 07:12:14.982575: W c:\l\tensorflow_1501907206084\work\t
Optimization Iteration:      0, Training Accuracy:  15.6%
Optimization Iteration:    100, Training Accuracy:  92.2%
Optimization Iteration:    200, Training Accuracy:  87.5%
Optimization Iteration:    300, Training Accuracy:  96.9%
Optimization Iteration:    400, Training Accuracy:  92.2%
Optimization Iteration:    500, Training Accuracy: 100.0%
Optimization Iteration:    600, Training Accuracy:  98.4%
Optimization Iteration:    700, Training Accuracy: 100.0%
Optimization Iteration:    800, Training Accuracy:  95.3%
Optimization Iteration:    900, Training Accuracy:  96.9%
Optimization Iteration:    999, Training Accuracy:  96.9%
Time usage: 0:01:39
Noise:
- Min: 0.0
- Max: 0.0
- Std: 0.0
```

Again as Above we see the CNN performs excellently when not exposed to the additive misclassification noise, with most 100 iterations resulting in a 95%-98% classification accuracy. Interestingly when exposed to a max limit of 0.35 adversarial noise one finds that the Network is unable to exceed a performance accuracy of 12%, with almost all input-images being predicted incorrectly due to the additive noise, as shown below:

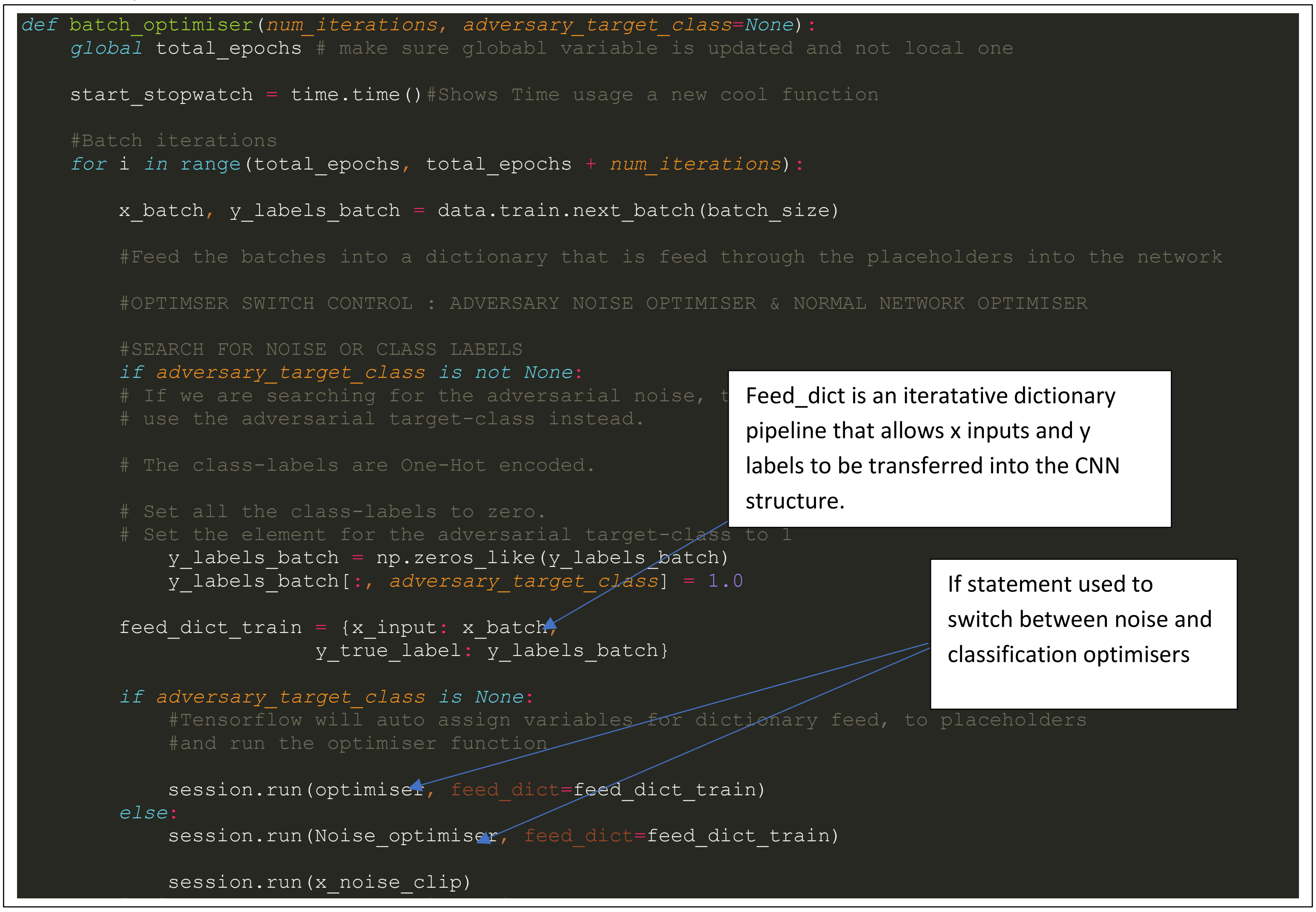


```
Accuracy on Test-Set: 11.9% (1194 / 10000)
Accuracy on Test-Set: 12.0% (1202 / 10000)
Accuracy on Test-Set: 12.0% (1199 / 10000)
Accuracy on Test-Set: 12.0% (1197 / 10000)
Accuracy on Test-Set: 11.9% (1195 / 10000)
Accuracy on Test-Set: 11.9% (1190 / 10000)
Accuracy on Test-Set: 11.9% (1189 / 10000)
Accuracy on Test-Set: 11.9% (1188 / 10000)
Accuracy on Test-Set: 11.9% (1189 / 10000)
```

Solutions:

After further research and experimentation, I was able to conclude upon a method to make the CNN robust to the adversarial noise. Unlike the single layered model, the CNN differentiates the overall network function to minimize the cross entropy loss function via backpropagating the respective weighted error contribution of each layer back through the network, we can use this (delta) to calculate how the weight parameters need to be updated in order to minimize the gradient slope of a loss function and thus optimise to a global or local minima when computationally traversing through error space. As a result of this learning procedure We can add a second optimisation function termed an L2 loss function that will specifically attempt to identify/classify the single pattern of adversarial noise that is causing the main network predictions to be incorrect. As shown below:

Duel optimiser Function:



```
def batch_optimiser(num_iterations, adversary_target_class=None):
    global total_epochs # make sure globabl variable is updated and not local one

    start_stopwatch = time.time()#Shows Time usage a new cool function

    #Batch iterations
    for i in range(total_epochs, total_epochs + num_iterations):

        x_batch, y_labels_batch = data.train.next_batch(batch_size)

        #Feed the batches into a dictionary that is feed through the placeholders into the network

        #OPTIMSER SWITCH CONTROL : ADVERSARY NOISE OPTIMISER & NORMAL NETWORK OPTIMISER

        #SEARCH FOR NOISE OR CLASS LABELS
        if adversary_target_class is not None:
        # If we are searching for the adversarial noise, t
        # use the adversarial target-class instead.

        # The class-labels are One-Hot encoded.

        # Set all the class-labels to zero.
        # Set the element for the adversarial target-class to 1
            y_labels_batch = np.zeros_like(y_labels_batch)
            y_labels_batch[:, adversary_target_class] = 1.0

        feed_dict_train = {x_input: x_batch,
                        y_true_label: y_labels_batch}

        if adversary_target_class is None:
            #Tensorflow will auto assign variables for dictionary feed, to placeholders
            #and run the optimiser function

            session.run(optimiser, feed_dict=feed_dict_train)
        else:
            session.run(Noise_optimiser, feed_dict=feed_dict_train)

            session.run(x_noise_clip)
```

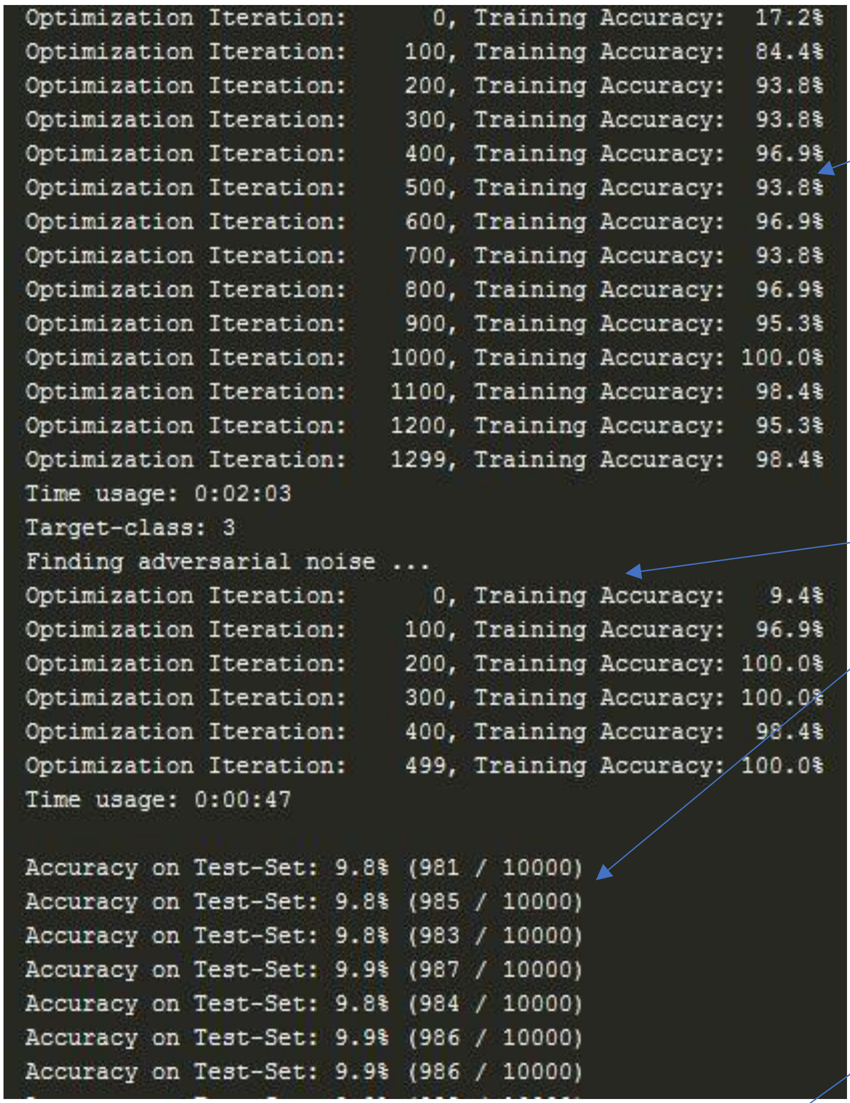
Feed_dict is an iteratative dictionary pipeline that allows x inputs and y labels to be transferred into the CNN structure.

If statement used to switch between noise and classification optimisers

With a dual optimization control function the network is able to optimise the parameters for standard classification whilst also being able to switch to an L2 function when exposure to adversarial noise causes performance accuracy to decrease significantly. The L2 function will the map the specific noise pattern to its respective noise class label E.g. noise class = 3, which is overlaid on the true class = 8. This noise loss function can then be minimised and in doing so will allow the network to ignore the presence of overlaid noise pattern. The code implementation can be shown below:

```
adversary_Variables = tf.get_collection(ADVERSARY_VARIABLES)
#We will combine the loss-function for the normal optimization with a so-called L2-loss for the
noise-variable.
# This should result in the minimum values for the adversarial noise along with the best
classification accuracy.
L2_loss_noise = L2_loss_processing_weight * tf.nn.l2_loss(input_noise)
#Combine Classifier - entropy loss function - optimiser with new l2 looss function
L2_lossFunction = loss_function + L2_loss_noise
#Define an optmiser for the Noise, so his adam opt will spesfically noise variables to
compensate
Noise_optimiser = tf.train.AdamOptimizer(learning_rate=1e-
2).minimize(L2_lossFunction,var_list=adversary_Variables)
#One optimiser will optimse the variable sof the network and the other will opt the variables of
the noise.
#So network learns how to classify why having a mechanism for ignoring noise
#=============================================================================
```

Firstly the loss function of the L2 noise optimiser is scaled to its optimization weight relative to the normal classifier optimizer. The loss of the L2 is then combined with the normal network cross entropy classifier loss. It is this combined loss that we feed to Noise optimiser that will update the noise parameters ultimately allowing the network to isolate the noise, and iteratively allow the network to ignore the noise, as evidenced below.

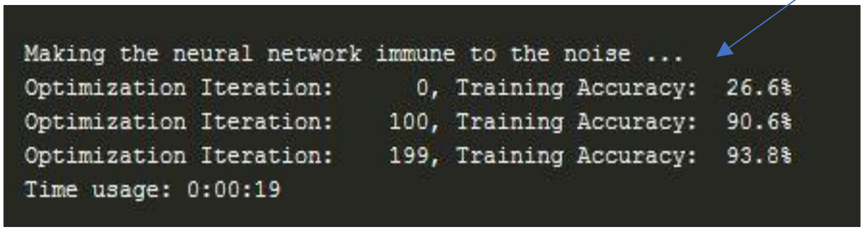


```
Optimization Iteration:      0, Training Accuracy:  17.2%
Optimization Iteration:    100, Training Accuracy:  84.4%
Optimization Iteration:    200, Training Accuracy:  93.8%
Optimization Iteration:    300, Training Accuracy:  93.8%
Optimization Iteration:    400, Training Accuracy:  96.9%
Optimization Iteration:    500, Training Accuracy:  93.8%
Optimization Iteration:    600, Training Accuracy:  96.9%
Optimization Iteration:    700, Training Accuracy:  93.8%
Optimization Iteration:    800, Training Accuracy:  96.9%
Optimization Iteration:    900, Training Accuracy:  95.3%
Optimization Iteration:   1000, Training Accuracy: 100.0%
Optimization Iteration:   1100, Training Accuracy:  98.4%
Optimization Iteration:   1200, Training Accuracy:  95.3%
Optimization Iteration:   1299, Training Accuracy:  98.4%
Time usage: 0:02:03
Target-class: 3
Finding adversarial noise ...
Optimization Iteration:      0, Training Accuracy:   9.4%
Optimization Iteration:    100, Training Accuracy:  96.9%
Optimization Iteration:    200, Training Accuracy: 100.0%
Optimization Iteration:    300, Training Accuracy: 100.0%
Optimization Iteration:    400, Training Accuracy:  98.4%
Optimization Iteration:    499, Training Accuracy: 100.0%
Time usage: 0:00:47

Accuracy on Test-Set: 9.8% (981 / 10000)
Accuracy on Test-Set: 9.8% (985 / 10000)
Accuracy on Test-Set: 9.8% (983 / 10000)
Accuracy on Test-Set: 9.9% (987 / 10000)
Accuracy on Test-Set: 9.8% (984 / 10000)
Accuracy on Test-Set: 9.9% (986 / 10000)
Accuracy on Test-Set: 9.9% (986 / 10000)
```

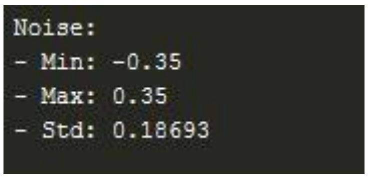
Begin With training the model to become a good image classifier, accuracy levels reach expected percentage levels

The network is then exposed to a noise pattern of image class label '3'. The network defaults to the L2 optimiser that begins to classify the noise pattern over 500 iterations with high accuracy. The effects of noise can also be seen as network classification performance has dropped to 9.8% accuracy

The network then becomes immune to the noise exposure by training the classification network to ignore the identified noise signal. Over now only 200 iterations the network returns to normal accuracy percentage values of 90+%. Final image evidences presence of adversarial noise in the system, during training.



```
Making the neural network immune to the noise ...
Optimization Iteration:      0, Training Accuracy:  26.6%
Optimization Iteration:    100, Training Accuracy:  90.6%
Optimization Iteration:    199, Training Accuracy:  93.8%
Time usage: 0:00:19
```



```
Noise:
- Min: -0.35
- Max: 0.35
- Std: 0.18693
```

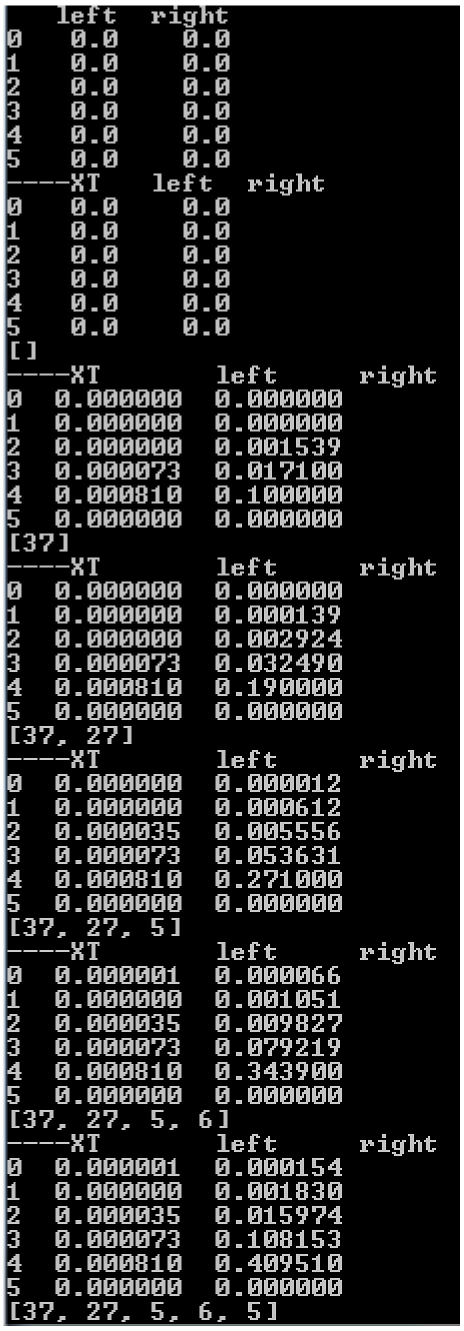
The following results demonstrate that models can be equipped with computational protocols that allow for the network to be dynamically robust to the exposure overwhelming outs adversarial noise, as shown with simplistic one layered model if noise was high enough the network had no protocol response to clean up the noise to allow for further progression. The CNN described above does employ such responses, whereby a high error rate triggers an “immune” response in the network, thus allowing for in this case the CNN to become increasingly generalisable during image classification.

Going Beyond

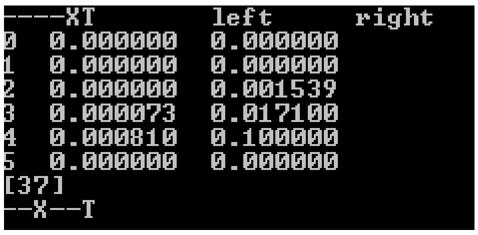
So far the report has demonstrated that neural network performance can benefit from training with a distribution of noise or by implementing a dual L2 optimizer controller can respond dynamically to the exposure of noise. However, we can think of noise more generally as an unexpected input-stream. Given the one-layered network, we had a noise level that could take on the value of a distribution of possibilities, but the process by which this range of noise that can occur for a specific input stream is a stochastic process and inherently random. As stated we can train with a model on a possible distribution of the variability of an input stream, but even still there may come a time when the model faces a new noise distribution with an even greater range than previous now trained noise distribution. As stated above we can generated models that are able to dissociate between noise and the relevant information, but this again is limited to the fact that one has to know what the prior range of labels to looks allowing the network to know what base error from, this maybe somewhat more challenging when during unsupervised learning when there are no predefined labels. A possible solution to creating more generalised robust models that can shift their decision boundaries more freely may lie in reinforcement learning. The start up company Deep mind demonstrated that a neural network was able to learn and out perform humans on more than 40 Atari games with only the pixel data and score provided as an input stream. This now has been coined q learning which are implemented by deep q networks. The deep Q network utilised by deepmind had the ability generalise its decision boundary in terms of which, action to take, by learning the action to value function that ultimately results in a world state in which the receives reward. The AI will then learn over time the optimal policy for navigating from its current state to the state that yields the most immediate reward. By implementing this approach computationally the AI at deep mind was able to generalise to variety of games and thus demonstrating its ability to reason through noise and randomability. This approach may yield a more productive approach to handling noise data as decisional boundries can be shifted based on the environmental rewards available.

I implemented a short program to illustrate the point. The program begins at state x and has to navigate to state T if it does it receives a reward. The Ai updates its Q matrix that describes the relationship between state, action and probability of a reward. The Ai will learn to the best policy to navigate to a state in with the value of Q is highest. We can consider that all other paths that do not lead to the T as distractors, thus a noise variable that the Ai must reason through. As you can see below the Ai is able to learn the optimal policy extremely quickly. Given the CNN described above, implementing Q learning algorithms with the network structure may enable the network to display a more flexible decision boundary that is sensitive to environmental reward.

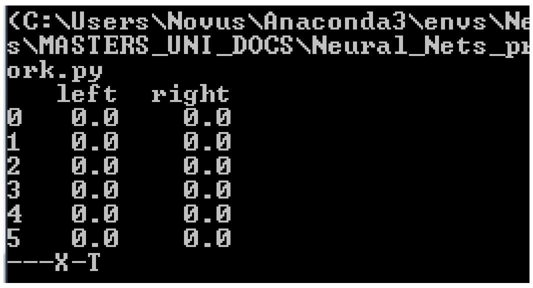
See below:



```
   left  right
0   0.0    0.0
1   0.0    0.0
2   0.0    0.0
3   0.0    0.0
4   0.0    0.0
5   0.0    0.0
----XT   left  right
0   0.0    0.0
1   0.0    0.0
2   0.0    0.0
3   0.0    0.0
4   0.0    0.0
5   0.0    0.0
[]
----XT        left      right
0  0.000000  0.000000
1  0.000000  0.000000
2  0.000000  0.001539
3  0.000073  0.017100
4  0.000810  0.100000
5  0.000000  0.000000
[37]
----XT        left      right
0  0.000000  0.000000
1  0.000000  0.000139
2  0.000000  0.002924
3  0.000073  0.032490
4  0.000810  0.190000
5  0.000000  0.000000
[37, 27]
----XT        left      right
0  0.000000  0.000012
1  0.000000  0.000612
2  0.000035  0.005556
3  0.000073  0.053631
4  0.000810  0.271000
5  0.000000  0.000000
[37, 27, 5]
----XT        left      right
0  0.000001  0.000066
1  0.000000  0.001051
2  0.000035  0.009827
3  0.000073  0.079219
4  0.000810  0.343900
5  0.000000  0.000000
[37, 27, 5, 6]
----XT        left      right
0  0.000001  0.000154
1  0.000000  0.001830
2  0.000035  0.015974
3  0.000073  0.108153
4  0.000810  0.409510
5  0.000000  0.000000
[37, 27, 5, 6, 5]
```



```
----XT        left      right
0  0.000000  0.000000
1  0.000000  0.000000
2  0.000000  0.001539
3  0.000073  0.017100
4  0.000810  0.100000
5  0.000000  0.000000
[37]
--X--T
```



```
(C:\Users\Novus\Anaconda3\envs\Ne
s\MASTERS_UNI_DOCS\Neural_Nets_pr
ork.py
   left  right
0   0.0    0.0
1   0.0    0.0
2   0.0    0.0
3   0.0    0.0
4   0.0    0.0
5   0.0    0.0
---X-T
```

Code implementation: Python.

```
mport numpy as np
import pandas as pd
import time

np.random.seed(2)

#World Parameters

NUMBER_OF_STATES = 6 #Lenegth of the 1D world, possible state position the agent can be in
ACTIONS = ['left', 'right']  #what actions can the agent

#Q Hyper Parameters
epsilon = 0.9 # Greedy plociy
alpha = 0.1 # learning Rate
gamma = 0.9 #Discount factor = used to cause prob of reward based on current action to exp
decreasre the further in the future

Exploration_limit = 13
FPS = 0.3
performce = []
#Q table = agent attempts to learn from environment and puts its learned behviour in the
#The Q table- policies are generated from the q table

def build_q_table(number_of_states, actions):
    table = pd.DataFrame(
        np.zeros((number_of_states, len(actions))),columns=actions,)

    print(table)
    return table
# build_q_table(4, ['a', 'b'])
# exit()

#Action or decision making Function
def decision_making_function(current_agent_state, q_table):
    current_state_actions = q_table.iloc[current_agent_state, :] # iloc indexes the axis label,
for this indexing, grabbing row of the states = numbers of the q matric
    #Set randomability is action picking = This causes a non greedy action: causes novel actions
to be taken
    if (np.random.uniform() > epsilon) or (current_state_actions.all() == 0):
        action_select = np.random.choice(ACTIONS)
    else:
        #act Greedy pick action with highest reward which wil mostly likely be the  actions
temporally close to the current state
        action_select = current_state_actions.idxmax() # argmax = highest value in q-table

    return action_select

def game_engine(current_state, game_action):
    if game_action == 'right':
        if current_state == NUMBER_OF_STATES -2:
            next_state ='TERMINATE'
            reward = 1
        else:
            next_state = current_state + 1
            reward = 0
    else:
        reward = 0
        if current_state == 0:
            next_state = current_state #Hit the boundries of 1D space
        else:
            next_state = current_state -1 # Move left

    return next_state, reward


def update_environment_controller(current_state, episode, iter_counter):
    oneD_World = ['-'] * (NUMBER_OF_STATES - 1) + ['T']

    if current_state == 'TERMINATE':
        performance = 'episode %s: how_many_steps = %s' %(episode+1,iter_counter)
        print('\r{}'.format(performance), end='')
        performce.append(iter_counter)
        time.sleep(10)

        print('\r                                        ', end='')


    else:
        oneD_World[current_state] = 'X'
        make_world = ''.join(oneD_World)
        print('\r{}'.format(make_world), end='')

        time.sleep(0.5)

#Build Reinforcement Learning Alogriythm

def RL_AlGO():
    q_matrix = build_q_table(NUMBER_OF_STATES, ACTIONS)
    for explore in range(Exploration_limit):
        iter_counter = 0
        current_state = 0
        end_game = False
        update_environment_controller(current_state, explore, iter_counter)

        while not end_game:
            game_action = decision_making_function(current_agent_state=current_state,
                                                   q_table=q_matrix)
            next_state, reward = game_engine(current_state=current_state,
                                             game_action=game_action)
            prediction = q_matrix.ix[current_state, game_action]
            if next_state != 'TERMINATE':
                #reinforcement learning algo - calculate maximum reward given state and action#
                Q_val = reward + gamma * q_matrix.iloc[next_state,:].max()
                #print(q_matrix)
            else:
                Q_val = reward # = 1 game eneded
                print(q_matrix)
                print(performce)
                end_game = True

            #update Qtable
            q_matrix.ix[current_state, game_action] += alpha * (Q_val - prediction)
            current_state = next_state

            update_environment_controller(current_state, explore, iter_counter)
            iter_counter = iter_counter + 1

    return q_matrix

if __name__ == "__main__":
    q_matrix = RL_AlGO()
    print('\r\nQ-table:\n')
    print(q_matrix)
```