Project: Follow Me

In this project, I've trained a model for a dataset of images using Neural Networks. I've learned Deep Neural Networks, Convolutional Neural Networks, Fully Convolutional Neural Networks and Semantic Segmentation over the Lab Exercises provided. Each of these individual techniques with their results and project explanations are clearly defined with pictures and code samples in this write up below. I've explained all the concepts separately before starting with the project code explanations, so that the terms used in Project explanation section are known beforehand.

NOTE: **Please jump into page 11, if only Project Implementation part is required to be verified.**

- **Lab 1 RoboND-NN-Lab**

Lab 1: **Tensor Flow Neural Network Lab with notMNIST data of English alphabets from A to J in different fonts.**

Criteria: **To train the network, and verify against the test dataset to acheive an 80% or more accuracy.**

Solution:

1. The first step for performing a data classification task using Neural Networks is to achieve a Dataset over which we want to train our model with a set of verified labels and do the predictions over a different test Dataset. A good Dataset often is a crucial parameter impacting the accuracy of predictions made over classifications. In this Lab, I was given 5,000 images for each label (A-J).

Each image in the dataset is comprised of Pixels. For a image in Grayscale, the individual pixel holds its corresponding Grayscale value usually between 0-1, after normalization. So for a 28X28 image, there are 784 pixels called **Inputs** which can be written as x_1 to x_n where n is the total number of inputs in a dataset. Each of these inputs holds its normalized Grayscale value called its corresponding **Activation**.

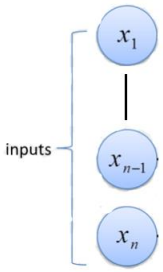


Fig 1 : Input Layer in a Neural Network

2. **Weight & Bias allocations:** After having the input layer we need to allocate weights for each of the inputs. The weights are the values that classifies a certain dataset in different required categories (or labels) throughout the inputs for all the layers. Weighted sum of all the inputs from one layer are passed to the inputs of the next layer, which then becomes the activation of all the inputs in that next layer.

If a certain input(or neuron) has higher weight than another input (or neuron) in the same layer then it depicts that, the input with higher weight is having higher influence than the input with lower weight.
A bias is a value that's added with the weighted sum of all the inputs in a layer, to specify how high the weighted sum needs to be before it gets active. The weights used are randomly normalized for initialization with zero mean and equal variance with a tensor flow variable tf.random_normal.

Numerically the weighted sum and bias, looks like: $Wx + b = \sum_{i=1}^{n} W_i x_i + b$ where w_i and x_i are the weights and inputs in one layer, n is the total number of inputs in the same layer and b is the bias.

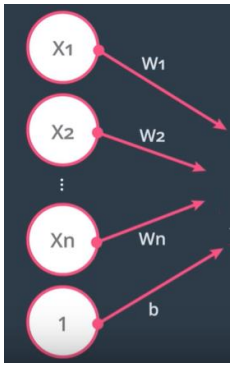


Fig 2 : Weighted Inputs and Bias for the input Layer

In code, with Tensor flow this is shown as below, where features_count is the count for number of inputs or the pixels for the first layer and labels_count is number of labels or the outputs for classification. Here biases are taken as zeros initially to set the activation outputs set high or low around zero. For this Exercise there are 10 labels required for letters A to J and 784 features_count for the 28X28 pixels.

```
weights = tf.Variable(tf.random_normal([features_count, labels_count]))
biases = tf.Variable(tf.zeros([labels_count]))
```

The below diagram shows the layers of a basic Neural Network:

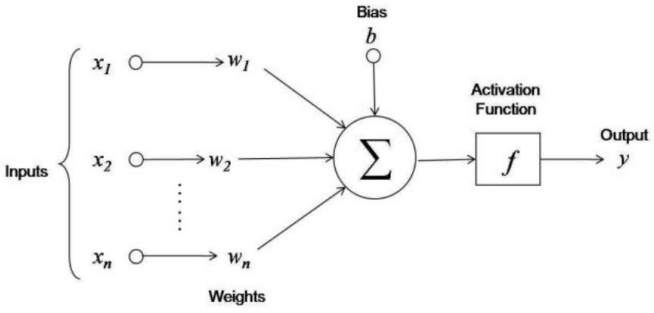


Fig 3 : Structure of a basic Neural Network [source: [1]]

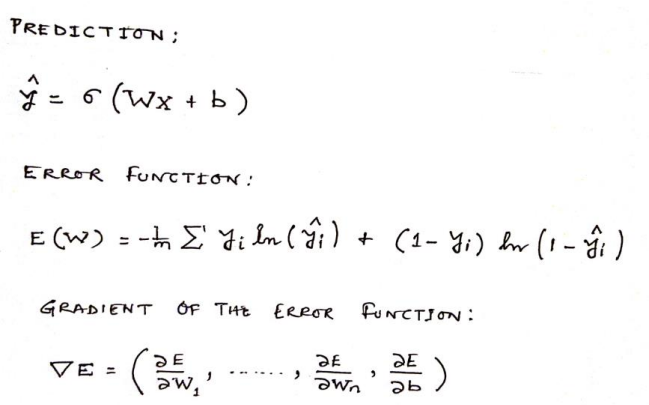
3. **Predictions & Loss Function:** The weighted sum of inputs and bias are then fed to softmax function to get the outputs for the next layer neuron. The softmax function predicts the outputs or logits from all the inputs with the given weights and biases in the range 0-1. These predictions are the likelihood of a certain image to have a certain Letter(label)

expressed in probability. The probability values are compared with the encoded labels of the corresponding images which then allows for better weight computation through a process of Gradient Descent which computes the loss function. The aim is to reduce the loss till it reaches its minimum value. This increases the accuracy of model to predict the images corresponding to their labelled categories.

The Prediction and Error Loss Function computation for Deep Neural Network or Convoluted Neural Network or Fully Convolutional Neural Networks follows the same concept with exceptions in usage.

There are various optimizers like **SGD (Stochastic gradient descent optimizer), Adagrad optimizer, Adadelta optimizer, Adam optimizer** etc. All performs the same function of reducing Error Function with different methods. In the next sections, I've used a few of these different techniques, the underlying concept behind which I'm explaining here.

Prediction, Error function and Gradient Descent as described in the class is as follows:



In the lab for Tensor Flow with Neural Networks, predictions and Loss functions are calculated in code as shown below:

```
# Linear Function WX + b
logits = tf.matmul(features, weights) + biases

prediction = tf.nn.softmax(logits)

# Cross entropy
cross_entropy = -tf.reduce_sum(labels * tf.log(prediction), axis=1)

# Training Loss
loss = tf.reduce_mean(cross_entropy)
```

4. **Setting Hyperparameters:** After computing the Predictions and Loss functions, the weights are updated at a learning rate, to find the minimum of the Error function (or Loss Function) after a certain iterations. The lower the learning Rate and higher the number of cycles or epochs, the better is the model trained. Here I've chosen the parameters as shown below:

```
epochs = 5
batch_size = 100
learning_rate = 0.2
```

Here the batch_size is number of inputs allowed per epoch. This depends on the memory and resource allocation of one's system. More about hyperparameter is also explained in the project explanation under the section **"Hyperparameter Selection" .**

5. **Feedforward and Backpropagation:** Using the above techniques of weight and bias computation, in every iteration the weighted sum of inputs are calculated using Feedforward for all the layers to receive the prediction values for the labels. With this training of model, the encoded label comparison with predictions, calculates the Cross Entropy. The Loss value or the Error function is then reduced at a learning rate towards the local minima of Error function with the technique of Backpropagation. These updated weights are again passed through all the layers to receive the updated predictions through Feedforward. This loop is repeated through iterations till the required accuracy is obtained.

   The weights are updated in every epoch as follows:

$$W'^{(k)}_{ij} \leftarrow W^{(k)}_{ij} - \alpha \frac{\partial E}{\partial W^{(k)}_{ij}}$$

Here α is the learning Rate, E is the Error function, $W_{ij}^{(k)}$ is the previous weight and $W'_{ij}^{(k)}$ is the updated weight.

The diagrammatic representation of feedforward and backpropagation flow looks like this:

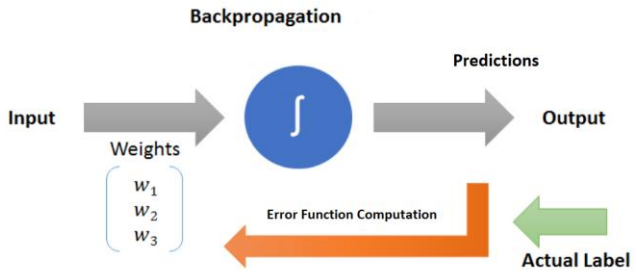


Fig 3 : Feedforward and Backpropagation flow in Neural Networks [source: [2]]

Results & Analysis: With the concepts discussed in the steps above and the tuned Hyperparameters, I obtained an accuracy of 0.832 for the model trained for English Alphabets from A to J.

```
Epoch  1/5: 100%|██████████| 2850/2850 [00:01<00:00, 1681.79batches/s]
Epoch  2/5: 100%|██████████| 2850/2850 [00:01<00:00, 1651.05batches/s]
Epoch  3/5: 100%|██████████| 2850/2850 [00:01<00:00, 1614.89batches/s]
Epoch  4/5: 100%|██████████| 2850/2850 [00:03<00:00, 875.77batches/s]
Epoch  5/5: 100%|██████████| 2850/2850 [00:03<00:00, 877.58batches/s]
Nice Job! Test Accuracy is 0.8320000171661377
```

➢ Lab 2 RoboND-DNN-Lab

Lab 2: **Tensor Flow Neural Network Lab with notMNIST data of English alphabets from A to J in different fonts.**

TODO: **To train the network, and verify against the test dataset with *Deep Neural Networks***

Solution:

This Lab work follows from the previous Lab, with the additional implementations illustrated below:

The model outputs after computation from WX+b gives a linear function, while in real world data classification, based on the image type in the dataset, the model needs to be non-linear. For this introduction of non-linearity, **ReLUs** are used in this Lab.

6. **ReLU Implementation:** ReLUs are functions that gives the output y as y = x (where x is the input) when x>0 and y=0 for x<=0. This function is applied to the outputs of all the layers before its sent to the next layer inputs. This allows the model to be non-linear with much ease.

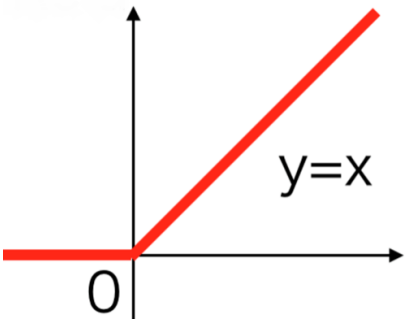


Fig 4: ReLU function [source: [3]]

With ReLUs , the implementation of weights and biases changes with respect to the Hidden Layer Width. A Hidden Layer is layer or layers between the input data pixels and the output labels. As Hidden layer works all by itself, we only see the results by tweaking a few parameters, the actual function inside it is not visible and hence Hidden. With respect to this the Input layer is called the Visible Layer as the Input data is nothing but the dataset fed by us.

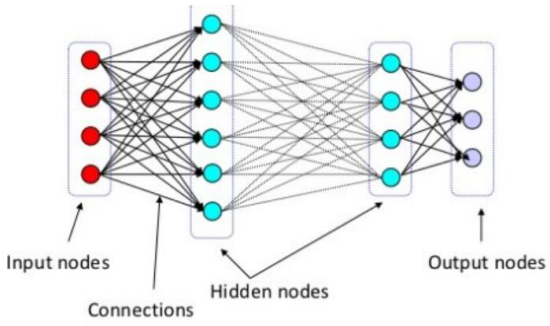


Fig 5: Neural Network with Hidden Layer Structure [source: [4]]

So once we calculate the Linear models from the input layer, its passed to a ReLU function. The outputs of this enters as inputs in the hidden layer. This depending on the hidden layer width, calculates the weighted sum of all the inputs and allows the resultant weighted sums to pass through a ReLU again, the outputs of which are sent to the final Output Layer in case the number of Hidden Layers used are 2. For the outputs, the predictions are calculated using softmax function for all the Labels.

7. **Regularization:** Regularization is technique to prevent Overfitting of Model, which is highly required for a model to train correctly. This is prevented here using Dropout with a tensor flow variable named tf.nn.dropout(layer, keep_prob), where keep_prob is the variable that determines the number of units to keep (not to drop). With a few inputs dropped randomly in every layer, the outputs are never dependent on any particular activation and the network learns from a redundant representation of every data.
For this Lab, I've taken a value of 0.75 in keep_prob which means I'm dropping ¼ of units in every layer.

This then follows the same process of Backpropagation and Feedforward with tensor flow tf.train.GradientDescentOptimizer and hyper parameters, as described in step 4 and 5 above to minimize the Loss function in every subsequent epoch:

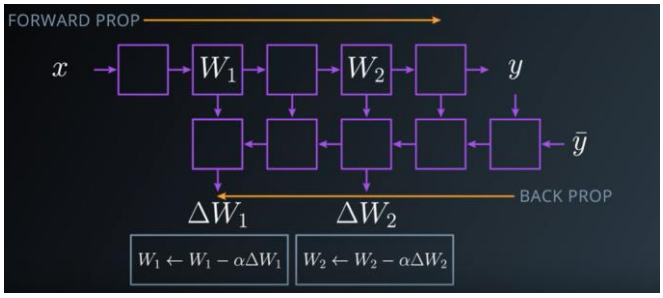


Fig 6: Model Update Flow in Neural Networks with Feedforward and Backpropagation

In this lab, the hyper parameters used are similar to the ones used in the previous lab with values of num_epochs as 5, learning_rate as 0.2, batch_size as 100 and Keep_probability as 0.75.
Initial Weights and biases are computed based on normalized values at 0 mean and 0.01 standard deviation:

```
features_count = 784
labels_count = 10

# TODO: Set the hidden layer width. You can try different widths for different layers and experiment.
hidden_layer_width = 64

# TODO: Set the features, labels, and keep_prob tensors
features = tf.placeholder(tf.float32, shape = (None, features_count))
labels = tf.placeholder(tf.float32, shape = (None, labels_count))
keep_prob = tf.placeholder(tf.float32)


# TODO: Set the list of weights and biases tensors based on number of layers
weights = [tf.Variable(tf.truncated_normal([features_count, hidden_layer_width], mean = 0.0, stddev = 0.01)),
           tf.Variable(tf.truncated_normal([hidden_layer_width, hidden_layer_width], mean = 0.0, stddev = 0.01)),
           tf.Variable(tf.truncated_normal([hidden_layer_width, labels_count], mean = 0.0, stddev = 0.01))]

biases = [tf.Variable(tf.zeros([hidden_layer_width])),
          tf.Variable(tf.zeros([hidden_layer_width])),
          tf.Variable(tf.zeros([labels_count]))]
```

Here I've created a 2 layer Neural Structure with hidden layer width of 64:

```
# TODO: Hidden Layers with ReLU Activation and dropouts. "features" would be the input to the first layer.
hidden_layer_1 = tf.add(tf.matmul(features, weights[0]), biases[0])
hidden_layer_1 = tf.nn.relu(hidden_layer_1)
hidden_layer_1 = tf.nn.dropout(hidden_layer_1, keep_prob)

hidden_layer_2 = tf.add(tf.matmul(hidden_layer_1, weights[1]), biases[1])
hidden_layer_2 = tf.nn.relu(hidden_layer_2)
hidden_layer_2 = tf.nn.dropout(hidden_layer_2, keep_prob)

# TODO: Output Layer
logits = tf.add(tf.matmul(hidden_layer_2, weights[2]), biases[2])
```

Adding more hidden layers or hidden layer width, impacts the predicted labels, as this adds on to the non-linearity of the overall model generated.

➢ Lab 3 RoboND-CNN-Lab

Lab 3: **Classify images from the Fashion-MNIST dataset for different types of clothing items such as shirts, trousers, sneakers etc.**

TODO: **To train the network, and verify against the test dataset with *Convolutional Neural Networks***

Solution:

In this Lab, the network model is trained for Fashion-MNIST dataset for 60,000 examples of 28X28 grayscale image and 10 labels for different types of cloth wear like Trouser, Pullover, Dress etc.

8. **Convolutional Neural Networks:** Convolutional Neural Network extracts features from the input image. Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data. Out of an image of 28X28 pixels and depth of 3 for RGB, a kernel of a certain width, height and depth is taken out, which is then allowed to swipe over the input image with a certain stride value. This kernel is having equal weights through out the image area, which ensures the fact that a particular data is independent of its pixel location in a given image and therefore carries equal weight throughout the image. The output of this results in a layer of new width and height depending on strides, kernel size and Padding. Padding can be of 2 types namely 'SAME' & 'VALID' padding. A 'SAME' padding is for the output image of the same size as input image, where the patches are allowed to go off the edges by padding with zeros whereas a 'VALID' padding is for the size of output image where the kernels are not allowed to go pass the edges.
   In most of the lab notebooks for this project I've used 'SAME' padding so that edge pixels are not left out from the available information passed on to the next layer.
   These kind of convolutional networks are applied to the input image through some layers to get an output image of the corresponding next layers which results in a reduced width and height (in case of stride value more than 1) and increased depth in every subsequent layer. Increase in depth in every layer increase the spatial information of the image with every layer resulting in growing non linearity and hence improved model.

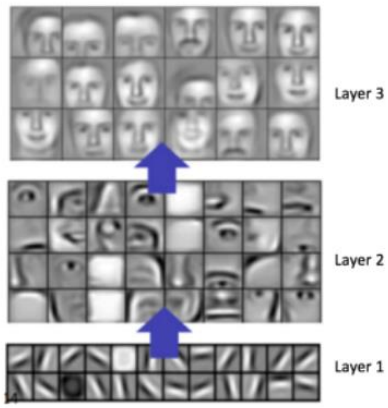


Fig 7 : Learned features from a Convolutional Neural Network [source: [5]]

In the starting layers, the models are linear in structure which are then merged over every next layer to make more and more non-linear functions. So for E.g., if the first layer depicts a few lines, the next layer will have the outputs for curves, the even higher layers will be combination of curves till the final layer which will be some feature outputs of things that are required for classification like human faces or birds etc.

Hence, increase in number of layers enhances the model training specially for dataset having rich spatial information like detecting facial features of a human being. Similar to this has been implemented for extracting features from the target and non-target objects in the "Follow Me" Project.

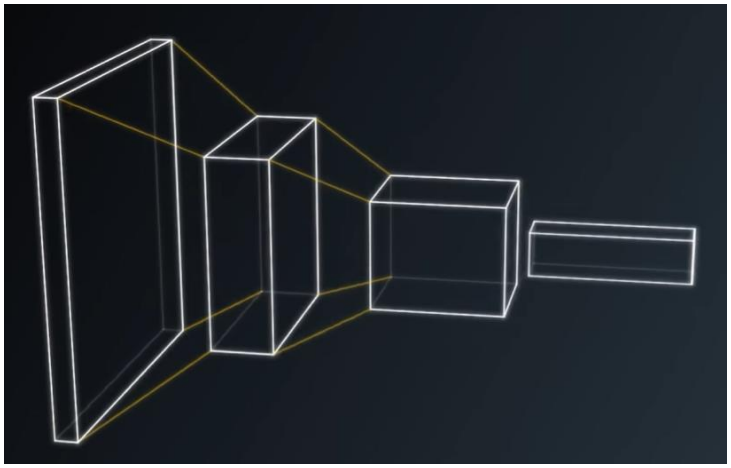


Fig 8 : Diagrammatic Representation of Convolutional Neural Networks

9. **Pooling:** Max Pooling is a technique used with the layer of convolutional networks. Max Pooling basically extracts out the highest pixel value or the most important pixel in a Pooling Kernel size. For every Pooling, there is a pooling stride value and Pooling kernel size similar to Convolutional Layer. Convolutional Network with a Max Pool Layer in Jupyter Notebook for this lab looks like:

```
filter_size = [conv_ksize[0], conv_ksize[1], x_tensor.get_shape().as_list()[3], conv_num_outputs]
weight = tf.Variable(tf.truncated_normal(filter_size, stddev = 0.01))
bias = tf.Variable(tf.zeros([conv_num_outputs]))

conv1 = tf.nn.conv2d(x_tensor, weight, [1, conv_strides[0], conv_strides[1], 1], padding='SAME')
conv1 = tf.nn.bias_add(conv1,bias)
conv1 = tf.nn.relu(conv1)

conv2 = tf.nn.max_pool(conv1,[1, pool_ksize[0], pool_ksize[1], 1],[1, pool_strides[0], pool_strides[1], 1],padding='SAME')
```

In this Lab, I've used 2 Convolutional and Max Pool Layers, followed by a Flatten Layer, followed by 2 Fully connected Layers. A complete CNN structure in code looks like this:

```
def conv_net(x, keep_prob):
    """
    Create a convolutional neural network model
    : x: Placeholder tensor that holds image data.
    : keep_prob: Placeholder tensor that hold dropout keep probability.
    : return: Tensor that represents logits
    """
    # TODO: Apply 1, 2, or 3 Convolution and Max Pool Layers
    conv_layer1 = conv2d_maxpool(x, conv_num_outputs=64, conv_ksize=(5,5), conv_strides=(1,1),
                                 pool_ksize=(2,2), pool_strides=(1,1))
    conv_layer1 = tf.nn.dropout(conv_layer1, keep_prob)

    conv_layer2 = conv2d_maxpool(conv_layer1, conv_num_outputs=128, conv_ksize=(3,3), conv_strides=(1,1),
                                 pool_ksize=(2,2), pool_strides=(1,1))
    conv_layer2 = tf.nn.dropout(conv_layer2, keep_prob)

    # TODO: Apply a Flatten Layer
    flat_layer = flatten(conv_layer2)

    # TODO: Apply 1, 2, or 3 Fully Connected Layers
    fc_layer1 = fully_conn(flat_layer, 128)
    fc_layer2 = fully_conn(fc_layer1, 64)

    output_layer = output(fc_layer2, 10)
    return output_layer
```

These combinations of convolutional layers followed by Max Pooling are done quite iteratively one after another through some layers making the network denser.
These layers together generates the output layer for predicted labels. In this lab, these predictions for me have resulted in an **accuracy of 0.88**, below is a screenshot:

```
Testing Accuracy: 0.8846914556962026
```

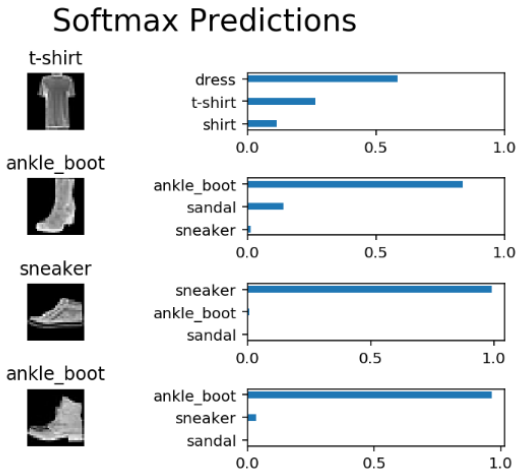


Fig 9 : Predictions of Fashion-MNIST dataset

➢ Lab 4 Semantic Segmentation Lab

Lab 4: **Build a deep learning network that locates a particular human target within an image**

TODO: **Building a semantic Segmentation network so that the target can be specifically located within the image.**

Solution:

This Lab is a pre-requisite towards the main project, I'll discuss next. Many of the concepts used in this Lab are introduced before and would be revised to focus on its application, for the required classification of this project. Here a quadcopter searches for a particular human target with a given image. The model needs to identify the target from the surroundings and other people to follow it thereafter.

10. **1x1 Convolutional Networks:** 1x1 convolutions are introduced for this and the following lab. A 1X1 Convolutional Network preserves the spatial information of the output image without changing anything in width and height thereby causing zero loss. This is being used instead of a flatten network after the convolutional layers from previous labs. With the help of a 1x1 convolutional network, the model is made more dense in structure due to the added depth of layers. The parameters for a certain 1x1 Convolutional Network is " HXWXD = 1x1x(filter size or depth)", "stride = 1", "SAME PADDING" .

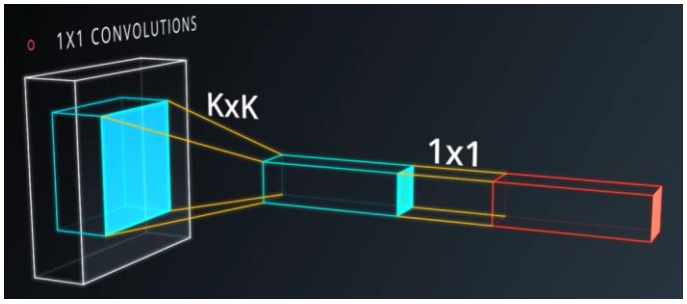


Fig 10 : Structure of 1X1 Convoluted Network

11. **Skip Connections:** Once the images are narrowed down, from its original size after the convolutional layers, decoding the output of the image to its original size is the most important requirement for Semantic Segmentation. This although is very difficult to achieve as some information lost, in the process of Encoded Layers when a higher width and height value of input image are brought down to lower width and height value, cannot be retrieved. Skip connections are a way to handle this for retaining the information by connecting the output of one decoder layer to a non-adjacent layer from the Encoder by concatenation or addition. The result is then passed to the next decoder layer. As a result Network is able to make more precise Segmentation decisions.

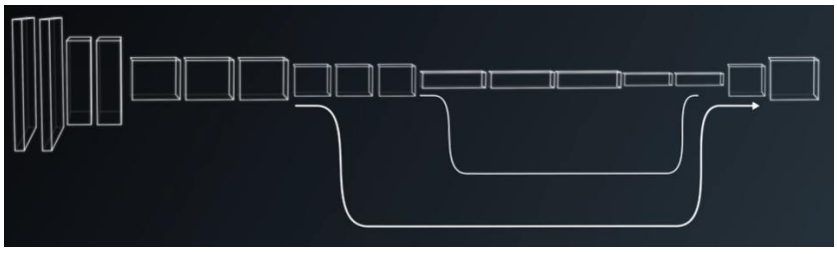


Fig 11 : Skip Connections Architecture

As the code in this Lab are similar to the project code, more about this lab and other concepts, I've explained together in the Project Implementation part.

➢ Project Implementation: Training Model

So far I've discussed about various concepts in Neural Networks, my understanding on it and how several parameters or layers influences the training of a model. Now for this phase of Project Implementation and training the model, I'll explain the techniques used here with every detail of hyperparameters and Network design.

Criteria: In this Project, the model is trained for a human target, which a Quadcopter in Unity Simulator follows. The aim towards the end of the project is to achieve an IOU final grade score of more than 0.4 to pass the submission criteria.

Solution: The workspace Jupyter Notebook in this project, begins with a few imports from Keras, Tensorflow and other python files placed in the Project Directory.

- **Data Collection:** A good dataset is a must requirement for a good model. I've used a dataset of 12014 images and Masks to train my model. This helps in better training to the model throughout the iterations.

- **FCN Layers:** FCN or Fully convolutional Neural Networks are comprised of Encoders and Decoders. Encoders extracts features from pixel level in the inputs which after a few layers are allowed to pass through a 1x1 Convolutional Network, this is then used to up sample the heights and widths in the corresponding layers of Decoder. This along with the technique of Skip Connections, gives the Final output layer of Semantic Segmentation for a certain image in a dataset. Here instead of a flatten network after convolutional layers, use of 1X1 convolutional layer helps in preserving the spatial information.

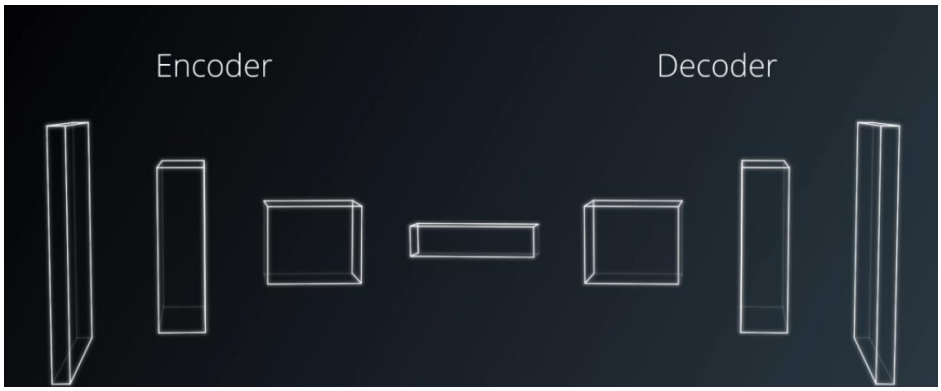


Fig 12 : FCN Structure

- **Encoders:** In the Encoder layer Separable convolutions are used for increasing the efficiency of network by reducing the number of Parameters. This is different to the regular Convolutional Layers I've discussed before. Here I've used a Kernel size of 2. The Kernel size is the Height and Width of the sliding window which shares the weights throughout the image region. Having a lower Kernel Value allocates more weights to the input image. If an

image is having highly diverse pixels of different categories of Data, then allocating a higher Kernel value may lead to assigning of similar weights throughout the image which may downgrade the performance of the model training. This although is good for memory allocation, as the requirements of number of parameters (weights) will be less in case of higher Kernel value. After optimizing the value for Kernel in previous Labs, I've chosen a value of 2 over 3 as it increases the model accuracy. Other than Kernel size for other inputs in separable convolutions I've used "SAME PADDING" and "ReLU" activation as it was provided in the notebook.
Along with the convolutional layer, a batch normalization is done. During training we normalize each layer's inputs by using the mean and variance of the values in the current mini-batch using batch normalization. Batch normalization provides the advantages like training the network faster, allowing higher learning rates and in regularization to prevent overfitting.

```
def separable_conv2d_batchnorm(input_layer, filters, strides=1):
    output_layer = SeparableConv2DKeras(filters=filters,kernel_size=2, strides=strides,
                                padding='same', activation='relu')(input_layer)

    output_layer = layers.BatchNormalization()(output_layer)
    return output_layer

def conv2d_batchnorm(input_layer, filters, kernel_size=2, strides=1):
    output_layer = layers.Conv2D(filters=filters, kernel_size=kernel_size, strides=strides,
                      padding='same', activation='relu')(input_layer)

    output_layer = layers.BatchNormalization()(output_layer)
    return output_layer
```

In encoder block, this is implemented as:

```
def encoder_block(input_layer, filters, strides):

    # TODO Create a separable convolution layer using the separable_conv2d_batchnorm() function.
    output_layer = separable_conv2d_batchnorm(input_layer, filters, strides)
    return output_layer
```

Decoders: In Decoders block, the output from 1x1 convolutional layer, is upsampled using a Bilinear Upsampling technique, where the output will be upsampled by a row and column Upsampling Factor. Here a value of (2,2) is chosen in one upsampling layer. A higher value of this may lead to a faster reproduction of the final output layer but at a lower resolution. This is followed by a concatenation of upsampled layer with an input layer. This is similar to skip connections discussed previously. Here the depth of the 2 layers to be concatenated need not be equal. Following this, I've added 2 separable convolution layers for my model to be able to learn the finer spatial details from the previous layers better. The strides for this I've kept as 1, as I don't want to loose any information from an already low resolution image. This helps in training the upsampled layer and improves the model accuracy.

```
def bilinear_upsample(input_layer):
    output_layer = BilinearUpSampling2D((2,2))(input_layer)
    return output_layer
```

```
def decoder_block(small_ip_layer, large_ip_layer, filters):

    # TODO Upsample the small input layer using the bilinear_upsample() function.
    upsample_layer = bilinear_upsample(small_ip_layer)

    # TODO Concatenate the upsampled and large input layers using layers.concatenate
    input_layer = layers.concatenate([large_ip_layer, upsample_layer])

    # TODO Add some number of separable convolution layers
    output_layer1 = separable_conv2d_batchnorm(input_layer, filters, strides=1)

    output_layer = separable_conv2d_batchnorm(output_layer1, filters, strides=1)

    return output_layer
```

- **Building Model:** With the Encoder and Decoder block functions created, a final model is required to be built for the aim of human target classification. For this I've chosen 4 Encoder layers as shown in the below diagram, Fig 13. Increasing the number of layers enhances the model training as the depth keeps on increasing with every separable convolutional layer thereby making the model more dense until the point, when increase in more layers no more increases the accuracy of trained model or worse can decrease the computational speed of model training. With this in mind, I've optimized the number of Encoder Layers with different values before finally fixing it to 4.

    - **Encoder Layers in Model**: The incoming input images are having dimensions of (160X160X3). These inputs are allowed over the first Encoder layer with the filter depth of 32 and stride of 1. This encoder layer hence comes with an output dimension of (160X160X32). These inputs are then passed to the next Encoder Layer where stride value is 2 and filter depth further increases to 64. With every passing Encoder Layer, the filter depth keeps on increasing. I've kept the value of strides around a lower range to have all the pixel information passed to the next layers. These values are optimized through multiple runs to have the best parameter selection.
      The output of second Encoder Layer with dimensions of (80X80X64) is allowed as an input to third Encoder Layer for which strides are equal to 2 and filter depth further increases to 128 making the Output Layer from this to have dimensions of (40X40X128). This is allowed over the final Encoder Layer of strides value of 1 and filter depth of 256, making the outputs with dimensions of (40X40X256). The Kernel size for all these Encoder Layers are 2, with padding as "SAME" and activation of outputs with ReLUs.

    - **1X1 Convolutional Layer:** The final Output of the Encoder Layer is then allowed to pass through a 1X1 convolutional Layer with strides=1 and kernel size =1 which generates an output layer of dimensions (40X40X256).

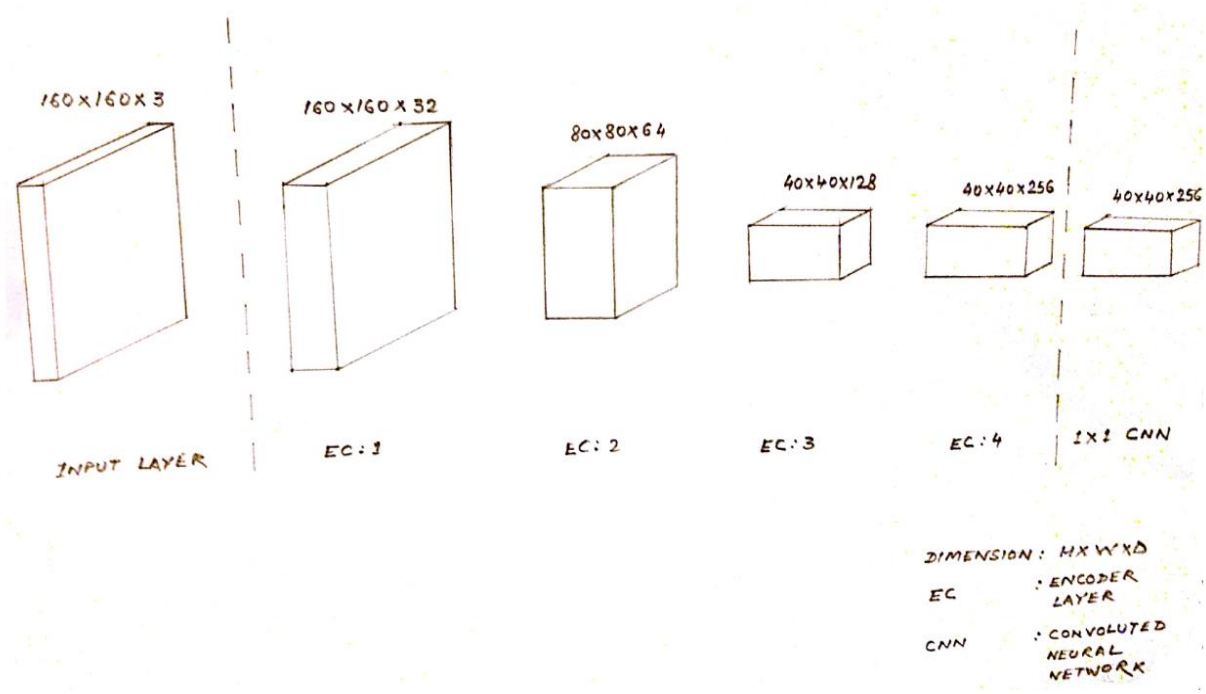


Fig 13 : Schematic Representation of Encoder Block followed by 1X1 convoluted network used in my Training model

- **Decoder Layers in Model:** The output from the 1X1 Convolutional layer is upsampled using bilinear upsample technique in the first decoder layer which gives the output dimensions as (80X80X64). This upsampled layer is concatenated with the 2nd Encoder Layer, which then undergoes 2 Layers of Separable Convolutions to have better training. Hence each of the Encoder block consists of 3 layers (1 Upsample Layer and 2 convolutional layers). There are 2 such Decoder blocks used in the algorithm. The first is having a filter depth of 64. The 2nd Decoder block is upsampled layer from the 1st Decoder Block which is then concatenated with the 1st Encoder Layer for model training following the Skip connections rule. This layer is then passed through 2 separable convolutional neural network again with filter depth of 32. The output of this final Decoder layer is having dimensions of (160X160X32).

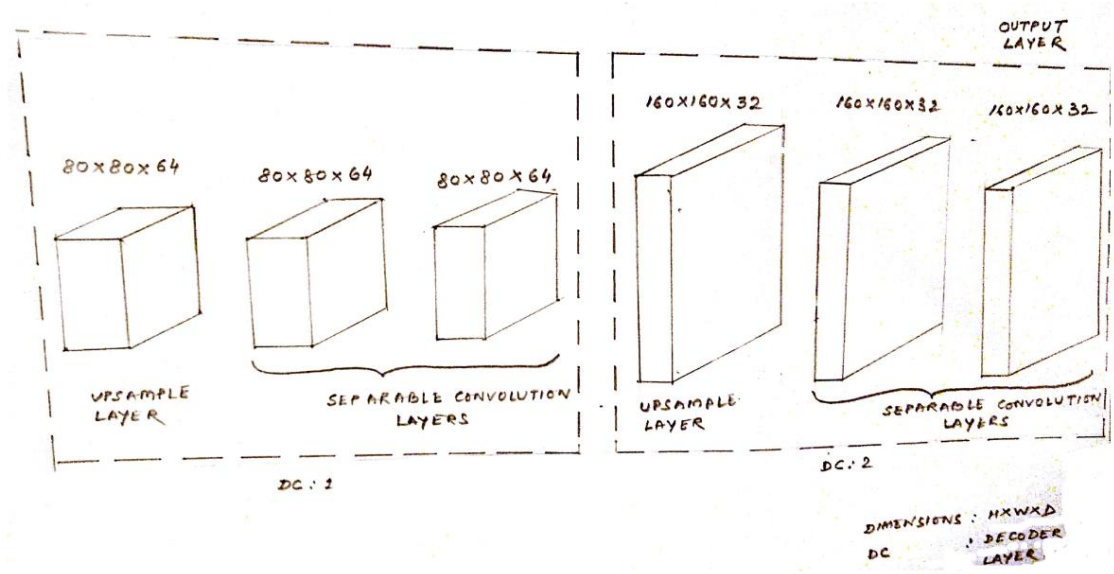


Fig 14: Schematic Representation of Decoder Block followed by the Output Layer used in my Training model

- **Output Layer:** The final output of Decoder Layer is having the Height and Width of the image similar to the Input Images. An activation of “softmax” is allowed over this final layer with “SAME” padding.

```
def fcn_model(inputs, num_classes):

    # TODO Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of your model (the number of filters) increases.
    filters1=32; filters2 = 64; filters3=128; filters4=256;

    output_layer1_en = encoder_block(inputs, filters1, strides=1)

    output_layer2_en = encoder_block(output_layer1_en, filters2, strides=2)

    output_layer3_en = encoder_block(output_layer2_en, filters3, strides=2)

    output_layer4_en = encoder_block(output_layer3_en, filters4, strides=1)

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    output_layer = conv2d_batchnorm(output_layer4_en , filters=256, kernel_size=1, strides=1)

    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    output_layer1_de = decoder_block(output_layer, output_layer2_en, filters=64)

    output_layer2_de = decoder_block(output_layer1_de, output_layer1_en, filters=32)

    x = output_layer2_de
    # The function returns the output layer of your model. "x" is the final layer obtained from the last decoder_block()
    return layers.Conv2D(num_classes, 1, activation='softmax', padding='same')(x)
```

The complete model with the Encoder, 1X1 Convoluted layer and Decoder block placed together gives an Output like this:

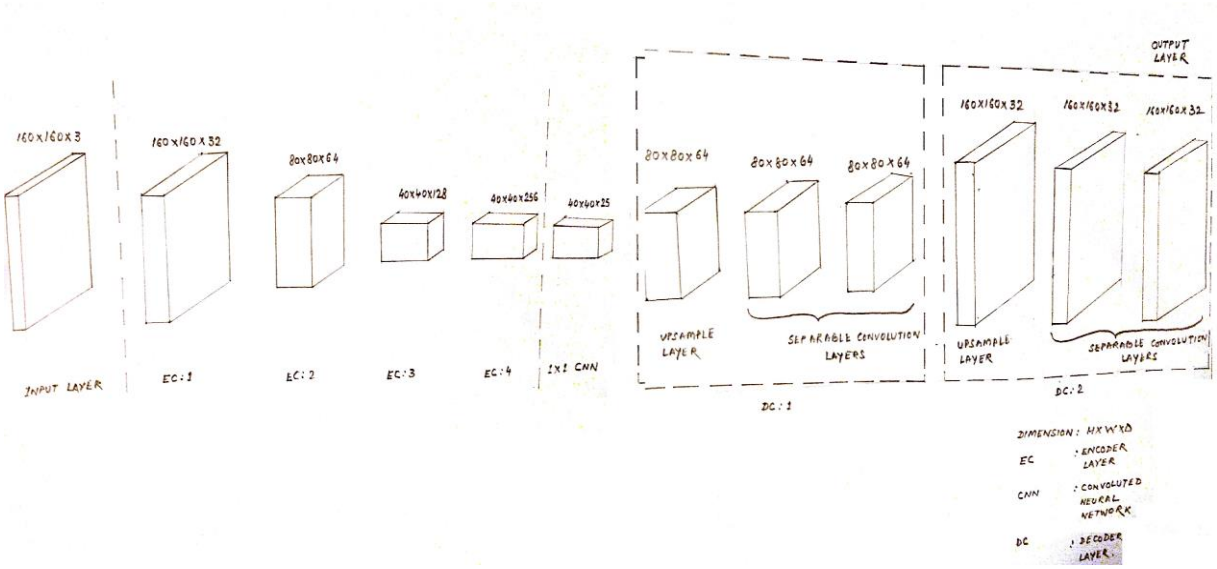


Fig 15: Schematic Representation of my Complete Training model

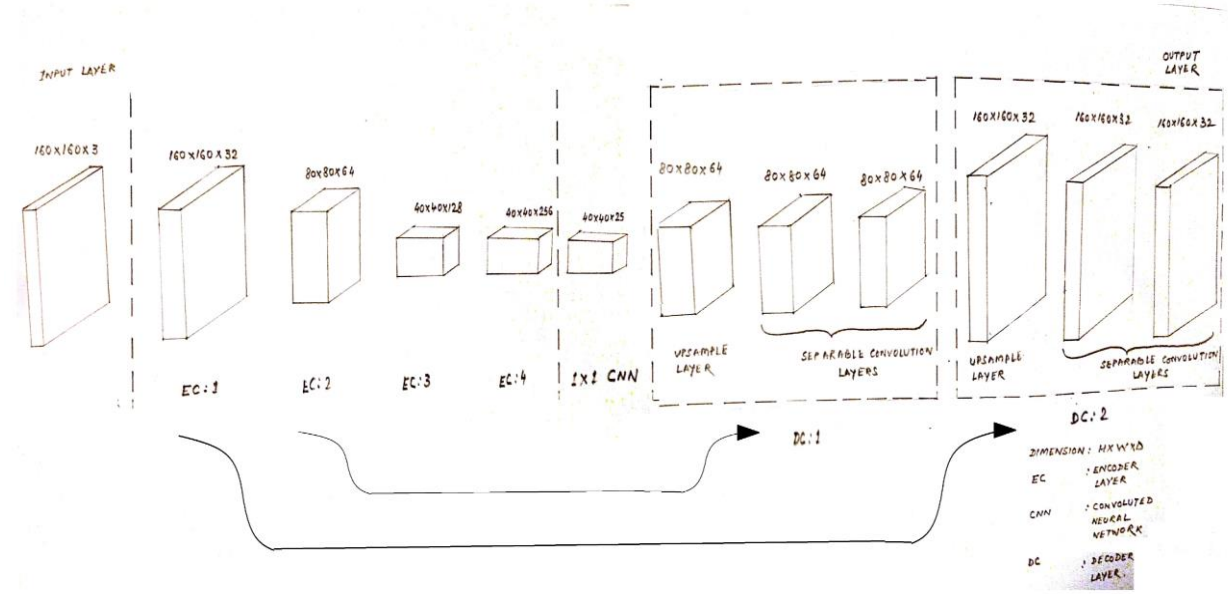


Fig 16: Skip Connections in Decoder Layers from Encoder Layers

- **Training:** The model is trained with the optimizer "optimizer=keras.optimizers.Adam(learning_rate)" over a 12014 images of training Dataset.

  Hyperparameter Selection: To train a model for its best performance, the algorithm requires the tuning for Hyperparameters to their best achievable values. Here the hyperparameters which are tuned are as follows:

  - ***Batch_size:*** Batch_size is the number of training samples/images that get propagated through the network in a single pass. The batch_size depends on the memory allocation of a system. The higher the number of images, the more is the requirement of memory, which leads to slow performance or Resource exhaustion errors of the PC. To avoid this instead of passing the entire Data at one time, the data is subdivided into batch_size, this allows the model to get trained without the system hanging. In terms of model training performance, the greater the value of

batch_size accommodated at one run, the better is the calculations for model towards the minimum loss function. At times having lower batch_size may lead the model to go in wrong direction, which then can be improved through increase in the number of epochs and reduction in learning rate so that average minimum loss function comes down to its minimum value at the end of Model Training. Hence, batch_size needs to be optimized between the trade offs of system performance issue and number of epochs and learning rate the model is required to trained on. In this project I've chosen a value of 100 as batch_size. This is explained with the figures below:

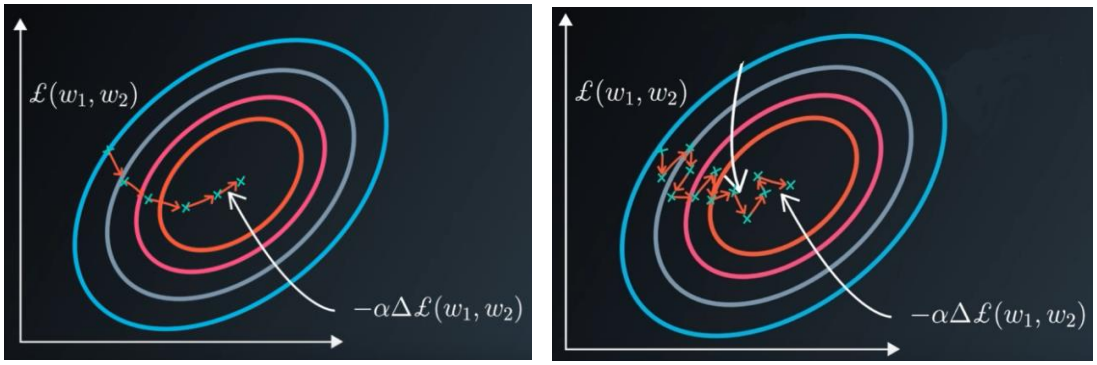


Fig 17 : Comparison on Higher(Left) Vs Lower Batch size (Right) in Loss Error function

- **learning_rate:** The learning_rate is the rate at which the model weights gets updated through backpropagation in every Error loss function computation. The lower the learning rate, the better the performance. But this value saturates at one point, where further reduction in learning rate results in no better improvement in the trained model. The value I've chosen for this project is 0.01.

- **num_epochs:** Number of times the entire training dataset gets propagated through the network. The higher the number of epochs the better the model is trained. This is because with every epoch, the Error function goes more closer to the local minima, updating the corresponding weights to achieve a better model. Here I've chosen an epoch value of 60 after testing with various other values.

- **steps_per_epoch**: steps_per_epoch is the number of batches of training images that go through the network in 1 epoch. The recommendation of value based on the total number of images in training dataset divided by the batch_size in my case is 12014/100 approximately 121. I've chosen 150 instead to let more images being allowed to train from training dataset per epoch.

- **validation_steps**: validation_steps is the number of batches of validation images that go through the network in 1 epoch. The recommendation of value based on the total number of images in validation dataset divided by the batch_size in my case is 1184/100 instead I've taken a value of 50 to let more images being allowed to validate from validation dataset per epoch.

- **workers**: Maximum number of processes to spin up. This I have kept the recommended value of 2 provided in the workspace.

These hyperparameters in the code is shown like this:

```
learning_rate = 0.01
batch_size = 100
num_epochs = 60
steps_per_epoch =150
validation_steps =50
workers = 2
```

Performance of this model on some different output classification like Dogs, Cats etc

This model is specifically trained according to the Input dataset provided and collected. The image pixels in the input layer are not from any other dataset of Dogs or Cats. Also the true labels are for only the datasets of this particular scenario of Project 4. This will not work for any other kind of classification, for which there is no images. Not only that even for human beings, there's only a certain kind of people that are present in the images that will be segmented properly.

For these kind of classification to work, there is requirement of corresponding pixels or images in the dataset, so that model could be trained by tuning the hyperparameters for these classifications.

IOU metric: The project submission criteria is dependent on a parameter called IOU or Intersection Over Union which is a score that determines the number of pixels in the predicted image matching with the number of pixels in the corresponding true labelled image divided by the total number of pixels in predicted class plus the total number of pixels in the true labelled class. This metric helps in finding out how accurate is our predicted model and thus also helps in improving the model further. The higher the value the better the model and vice-versa.

Results and Conclusions:

- The performance of the trained model for "Images while following the target" over 3 random images looks like:

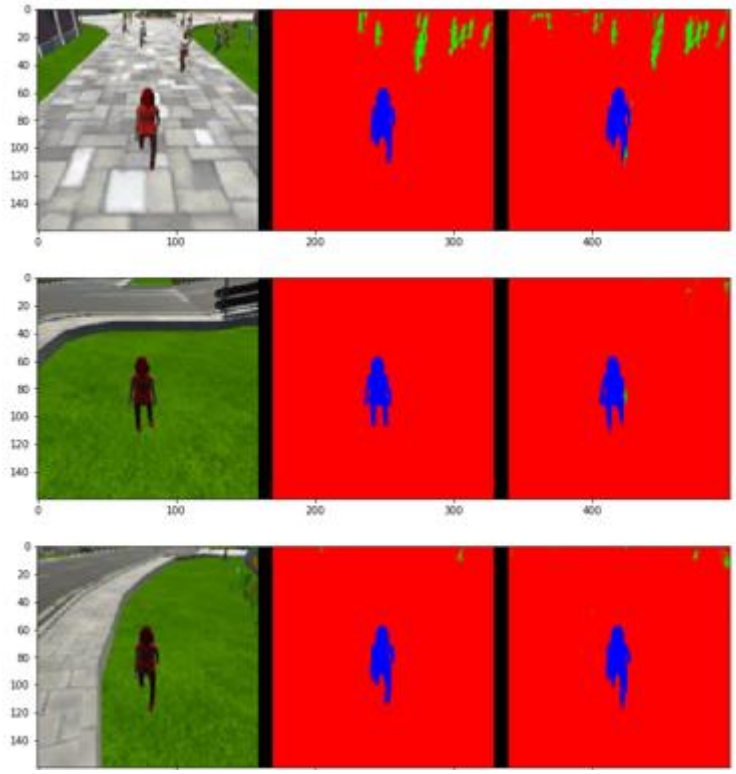


Fig 18 : Target Follow, 3rd Image: Output of my trained Model

- The performance of the trained model for “Images while at patrol without target” over 3 random images looks like:

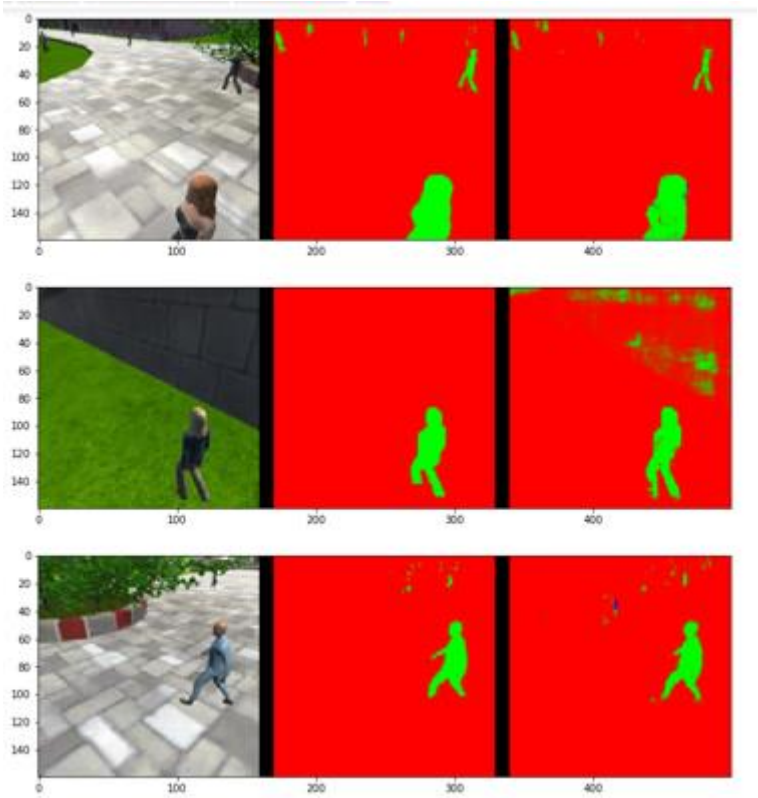


Fig 19 : Quad Patrols without Target, 3rd Image: Output of my trained Model

- The performance of the trained model for “Images while at patrol with target” over 3 random images looks like:

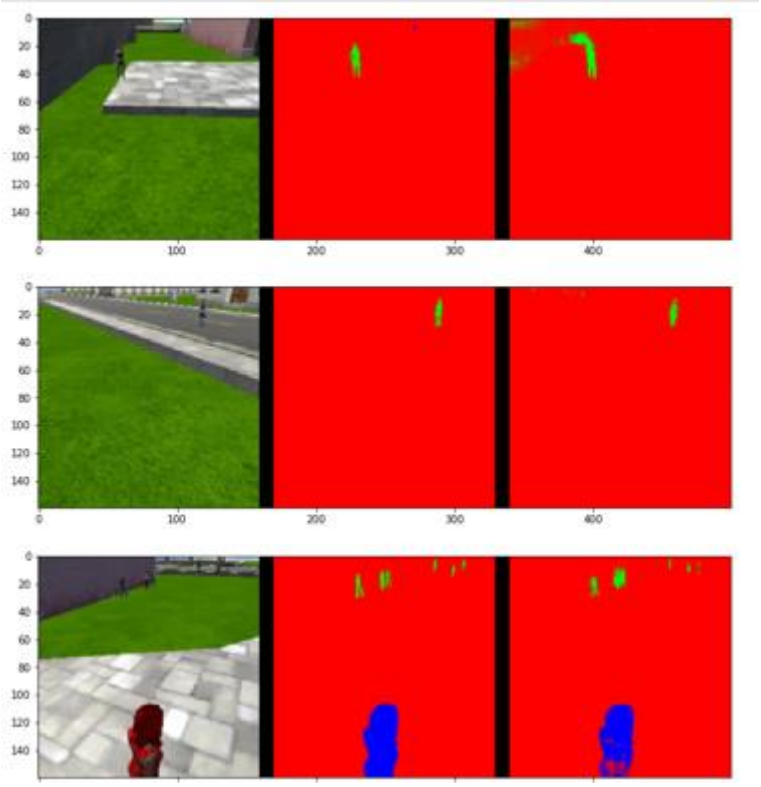


Fig 20 : Quad Patrols with Target, 3rd Image: Output of my trained Model

Fig 18, 19 and 20 depicts the model performance (3rd Image) with respect to the sample evaluation data (2nd Image). Below are the IOU scores of my model:

```
# Scores for while the quad is following behind the target.
true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_following, pred_following)
```

```
number of validation samples intersection over the union evaulated on 542
average intersection over union for background is 0.9937563451815151
average intersection over union for other people is 0.3386656646745068
average intersection over union for the hero is 0.8917546360255945
number true positives: 539, number false positives: 0, number false negatives: 0
```

```
# Scores for images while the quad is on patrol and the target is not visable
true_pos2, false_pos2, false_neg2, iou2 = scoring_utils.score_run_iou(val_no_targ, pred_no_targ)
```

```
number of validation samples intersection over the union evaulated on 270
average intersection over union for background is 0.9853519918758865
average intersection over union for other people is 0.7533047887105001
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 128, number false negatives: 0
```

```
# This score measures how well the neural network can detect the target from far away
true_pos3, false_pos3, false_neg3, iou3 = scoring_utils.score_run_iou(val_with_targ, pred_with_targ)
```

```
number of validation samples intersection over the union evaulated on 322
average intersection over union for background is 0.9956124516249008
average intersection over union for other people is 0.4586512181436584
average intersection over union for the hero is 0.3081232398866387
number true positives: 186, number false positives: 4, number false negatives: 115
```

The final Grade score is calculated with the product of weight and final_IOU:

```
# Sum all the true positives, etc from the three datasets to get a weight for the score
true_pos = true_pos1 + true_pos2 + true_pos3
false_pos = false_pos1 + false_pos2 + false_pos3
false_neg = false_neg1 + false_neg2 + false_neg3

weight = true_pos/(true_pos+false_neg+false_pos)

print(weight)
```

```
0.7458847736625515
```

```
# The IoU for the dataset that never includes the hero is excluded from grading
final_IoU = (iou1 + iou3)/2
print(iou1,iou3)
print(final_IoU)
```

```
0.891754636026 0.308123239887
0.599938937956
```

```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.447485318949
```

In the above screenshots it can be observed that, the weight computed is 0.746. The IOU scores are based on 3 types of classes:

a. following_images - Quad following the Target
b. patrol_non_targ - Patrolling with no Target
c. patrol_with_targ - Patrolling with Target

The final IOU is calculated with the average of IOU score wherever target is present with a weightage value.

The final grade score obtained from this is having a value of: 0.44748 (45% approx..)

This passes the submission criteria of the project to achieve a final grade score of 0.4 (40%).

How to run my model "model_weights" file to verify the results?

1. Please uncomment the below lines to load the model present in "\Project_4 Follow Me\Weights" directory.

```
# If you need to load a model which you previously trained you can uncomment the codeline that calls the function below.

#weight_file_name = 'model_weights.h5'
#restored_model = model_tools.load_network(weight_file_name)
```

2. After that, please change the variable named 'model' as shown in the below lines to 'restored_model'.

```
run_num = 'run_1'

val_with_targ, pred_with_targ = model_tools.write_predictions_grade_set(model,
                                        run_num,'patrol_with_targ', 'sample_evaluation_data')

val_no_targ, pred_no_targ = model_tools.write_predictions_grade_set(model,
                                        run_num,'patrol_non_targ', 'sample_evaluation_data')

val_following, pred_following = model_tools.write_predictions_grade_set(model,
                                        run_num,'following_images', 'sample_evaluation_data')
```

This will reproduce the results for grade score, I've explained here.

Future Enhancement:

I. Currently my model outputs a final grade score of 0.447 approx, this can be enhanced to higher value if we look through the reasons for this performance. For E.g., as we can see the prediction for following target is the best in my trained model which means for around 75% of images the segmentation for following images are good. Whereas for the images where target is present in the image but at a far off distance, the IOU calculated is not that good. The overall IOU score depends on an average which means both of these value, are equally important and hence are required to be trained properly.

II. In my training dataset, the input images that impacts IOU3 is very less and hence the model is not trained good enough for those test datasets. Hence for future implementation I would like to analyse the dataset more and more and collect the images that better implicates the output classes.

III. Increasing the weight could also be another solution which means reduction in False Positives. This again is dependent on the dataset. In following images the True positives are approximately 539 and target far away is having True positives of 186. This number for target far away can be improved with a better data collection which again relates to the point discussed above.

IV. Tuning of hyperparameters: Tweaking the value for hyperparameters can also result in better performance. Due to a limited GPU instance of 80 hours I couldn't try with many more variations in the hyperparameters.

V. Increasing in the Layers in model: Increase in layers in model makes the model more dense and hence more rich in spatial information. This may lead training enhancements in Model architecture.

References:

[1] https://naadispeaks.files.wordpress.com/2017/11/vqope.jpg

[2] http://dni-institute.in/blogs/wp-content/uploads/2015/02/Backpropagation-Learning.png

[3] http://img.thothchildren.com/ec6ef79b-788a-4e69-bffc-8fca03b38ed9.png

[4] https://www.kdnuggets.com/wp-content/uploads/neural-networks-layers.jpg

[5] Honglak Lee, *et al*, "Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations" (link)