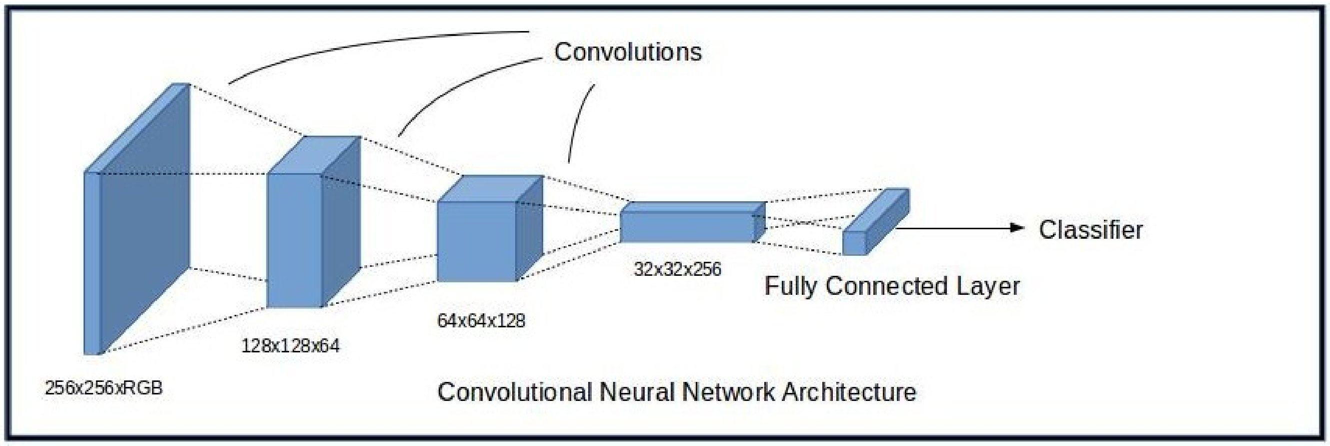
Follow-Me Project Write-up

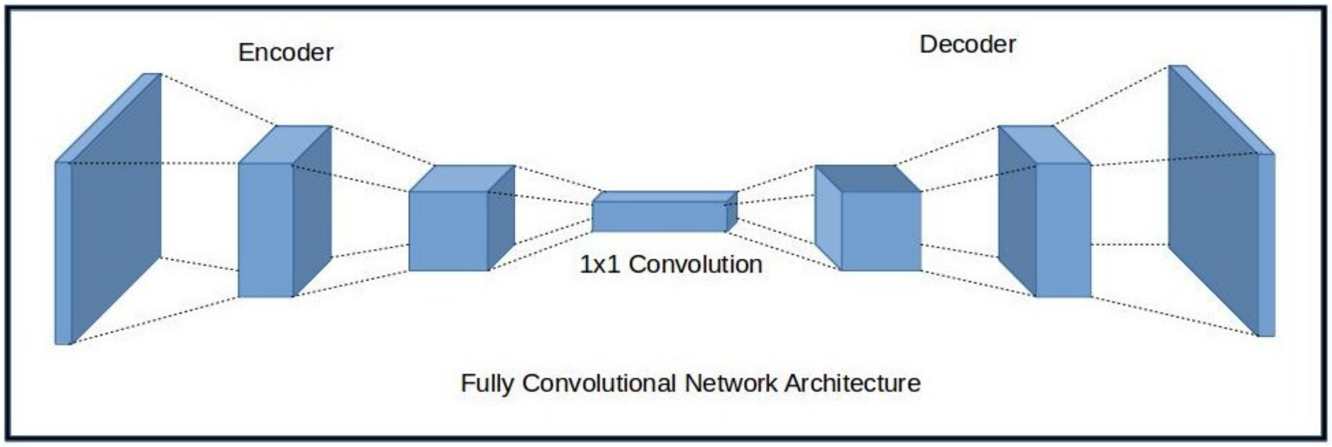
– Mohit Chaturvedi

➢ Network Architecture

To satisfy the objectives of this project, a Fully Convoluted Network (FCN) architecture is implemented with a final IoU accuracy score of 41.02%.
A typical Convolutional Neural Network (CNN) architecture consists of a series of convolutional layers followed by fully connected layers and a classification function. This type of architecture is helpful for basic classification of images but not for providing information about the location of targeted image in the scene. This is because the fully connected layers flatten the output of a convolutional layer into a 2D tensor resulting in loss of spatial information.



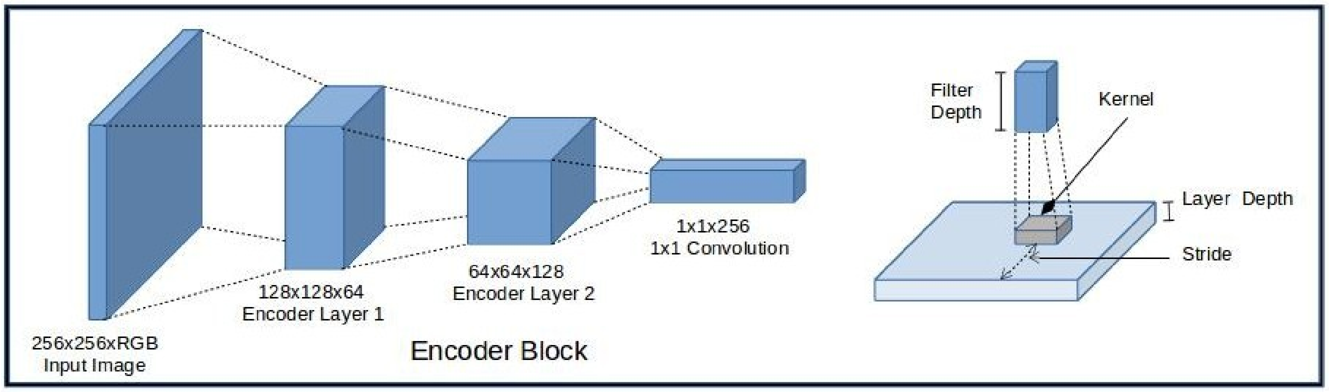
For the purposes of this project, it is important to not only identify the target but also to figure out where in the whole scene is the target present. In an FCN architecture, the fully connected layers of CNN is replaced by 1x1 convolutions followed by a series of convolutional layers (decoder block). This kind of architecture, while doing the convolution, keeps the output shape of a convolutional layer as a 4D tensor. This preserves the information about the location of the pixels throughout the entire network and hence becomes the primary choice for this project. Additionally, an FCN works on images of any size since the convolutional operation does not care about the size of input and the output images are upscaled to the same size as that of input images in the decoder block. However, in a CNN with fully connected final layers, the size of the input is constrained by the size of the fully connected layer. Passing different sized images through the same sequence of convolutional layers in this architecture results in output images of different sizes.



The FCN architecture used for this project consists of two encoder layers, one 1x1 convolutional layer and two decoder layers.

Encoder Block:

Encoder block in a typical FCN is a pre-trained classification network preceding the 1x1 convolutional layers in the architecture. The first layer in the network takes the input image with specific width, height and depth (RGB channel) and runs convolution operations on it resulting in an output image with reduced width and height but increased depth i.e. multiple color channels. Adding subsequent convolutional layers on top forms a pyramid structure. At the bottom is the input image which is bigger in size but shallower in depth. In the middle, a stack of convolutional layers progressively reduce the spatial dimensions while increasing the depth of the input image. Finally, at the top the output has only the parameters mapping to the content of the image with all the spatial information squeezed out. The dimensionality reduction is a resultant of choosing stride while the number of layers dictate the depth of the network.



Here, two convolutional layers are used in the encoder block that includes separable convolutions. Separable convolutions reduce the number of parameters needed; resulting in improved run time performance and reduced overfitting. They work by performing convolution over each channel of input layer followed by a 1x1 convolution that takes the output channels from previous step and combines them into an output layer. The separable_conv2d_batchnorm() function creates the encoder block taking input layer, filters and strides as arguments. Processed images of size 256x256x3 are passed on as input layer.

```python
def separable_conv2d_batchnorm(input_layer, filters, strides=1):
    output_layer = SeparableConv2DKeras(filters=filters,kernel_size=3, strides=strides,
                             padding='same', activation='relu')(input_layer)

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

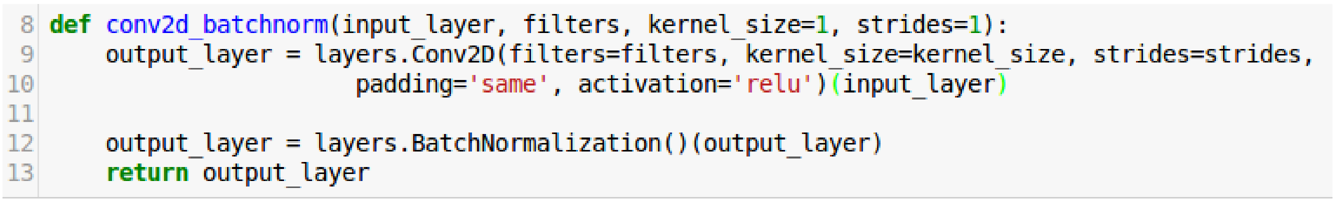
```python
def encoder_block(input_layer, filters, strides):

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

    return output_layer
```

A Filter is a set of learn-able weights with which we convolve the input image and detect feature maps. The number of filters in a convolutional layer is called filter depth and define the size of the output layer. A higher number of filters increase the chances of detecting required features but also increases computations. With trial and error method focusing on a higher IoU score, 64 number of filters is chosen for the first encoder layer.

A Stride is the number of pixels that shift each time that the filter is moved over the input image. A higher number of strides reduce the dimensionality of output image by reducing the total number of patches each layer observes but at the cost of accuracy.

1x1 Convolution Layer:

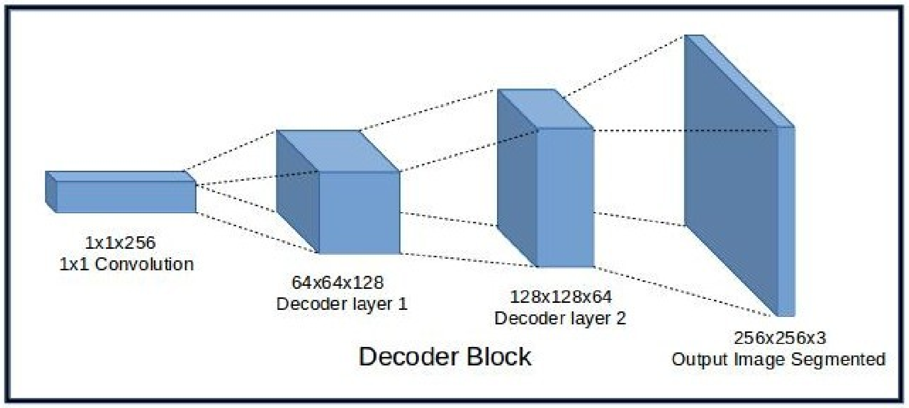
Following the encoder black, a 1x1 convolution layer is added to the FCN architecture.1x1 convolutions are an inexpensive way to make the network deeper providing more trainable parameters helping to increase the accuracy of the model. In a typical neural network, there's usually a small linear classifier running over the patch of the image. By adding a 1x1 convolution layer in the middle, we can introduce a mini neural network running over the patch instead of just a classifier. Replacing fully connected layers by 1x1 convolutions, this technique preserves spatial information while reducing dimensionality of the output layer. It is also cheaper since it turns convolutions into matrix multiplications operation and during inference any sized images can be fed to the model. As opposed to encoder layers utilizing the separable convolutions technique, the 1x1 convolution layer here is a regular convolution as implemented by the conv2d_batchnorm() function. This layer has kernel of size 1 with a stride of 1. The number of filters is chosen to be 256 with the input layer as the last layer of encoder block.

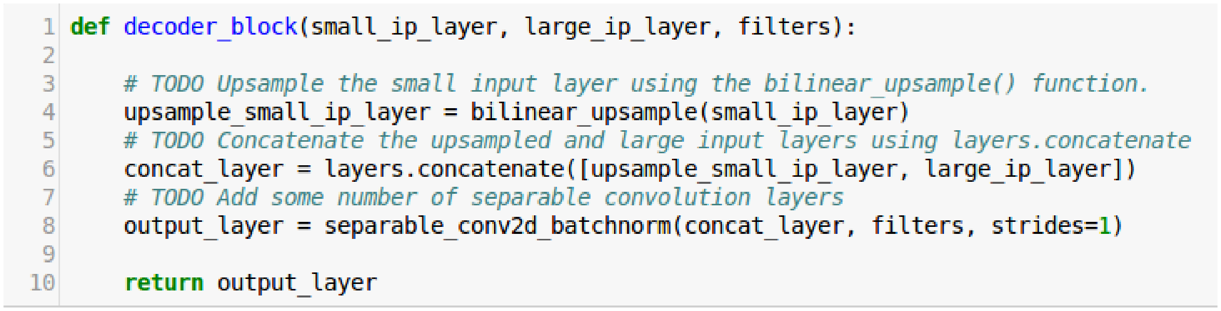


```python
def conv2d_batchnorm(input_layer, filters, kernel_size=1, strides=1):
    output_layer = layers.Conv2D(filters=filters, kernel_size=kernel_size, strides=strides,
                          padding='same', activation='relu')(input_layer)

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

Decoder Block:

The next part of FCN architecture is the decoder block which performs the task of semantic segmentation. It requires discrimination of features at pixel level with semantic projection of the these features learned by the encoder onto the pixel space. The decoder block consists of a sequence of layers implementing transpose convolution technique. A transpose convolution is a reverse convolution where the forward and backward passes are swapped and thus training the network remains the same as in case of regular convolutional network. This decoder block utilizes the features extracted by the encoder block and upsamples the output from the 1x1 convolution layer to the required higher dimensions.



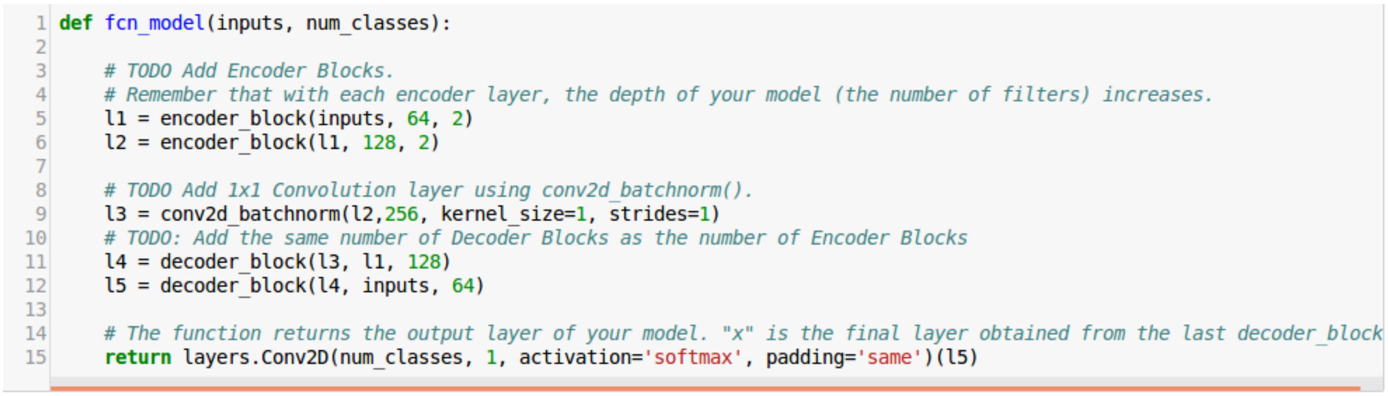


```python
def decoder_block(small_ip_layer, large_ip_layer, filters):

    # TODO Upsample the small input layer using the bilinear_upsample() function.
    upsample_small_ip_layer = bilinear_upsample(small_ip_layer)
    # TODO Concatenate the upsampled and large input layers using layers.concatenate
    concat_layer = layers.concatenate([upsample_small_ip_layer, large_ip_layer])
    # TODO Add some number of separable convolution layers
    output_layer = separable_conv2d_batchnorm(concat_layer, filters, strides=1)

    return output_layer
```

Constructing the decoder block consists of three parts namely,

1. Upsampling: The upsample_bilinear() function upsamples the input layer by a factor of 2. The bilinear upsampling method is prone to lose some finer details but it speeds up performance.
2. Concatenation: This step concatenates the small upsampled layer and the large input layer with more spatial information than the upsampled one. The idea is to utilize the advantages of skip connections technique to retain some of the finer details from the previous layers. As compared to adding layers, concatenating layers have the advantage that the depth of input layers need not match up which in turn simplifies the implementation.
3. Adding separable convolutions: To extract some finer details from previous layers and to let the model learn back from relevant features that were lost during encoding, its often advantageous to add some separable convolutional layers after concatenation. The number of layers are the same as the number of encoder layers i.e. 2.

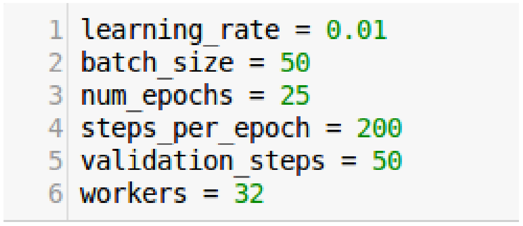
The following function implements the FCN architecture as discussed above.



```python
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.
    l1 = encoder_block(inputs, 64, 2)
    l2 = encoder_block(l1, 128, 2)

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    l3 = conv2d_batchnorm(l2,256, kernel_size=1, strides=1)
    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    l4 = decoder_block(l3, l1, 128)
    l5 = decoder_block(l4, inputs, 64)

    # 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')(l5)
```

➢ Model Training & Hyperparameters

Once the model is ready, the following hyperparameters are tuned to compile and train the model.



```python
learning_rate = 0.01
batch_size = 50
num_epochs = 25
steps_per_epoch = 200
validation_steps = 50
workers = 32
```

Learning Rate (0.01): In the context of optimization, learning rate determines how quickly or slowly the model parameters need to be updated in order to reach a function minimum. In the FCN architecture learning rate tuning is not straight forward. A higher learning rate can make the model learn fast initially but at the expense of getting plateaued later. Conversely, a lower learning rate starts slower but it can keep on going and give a better model. For this project learning rate was chosen based on trial and error while keeping other parameters constant. Initially with the learning rate of 0.001, the model compiled to a below 40% accuracy. That could have happened due to smaller steps to gradient descent making the algorithm converge to local minimum and resulting in data over-fitting. An increase of learning rate to 0.01 converged to the solution faster and gave a much better score of above 40% accuracy.

Batch Size (50): It is the number of training images that are going to be propagated through the network in single pass. A higher batch size requires more memory to train the model and takes longer to compute the gradient for each step. On the other hand, setting the batch size lower provides a less accurate estimate of the gradient as compared to higher batch size model. However, lower batch size drastically cuts down on training time and requires much less memory for computations. This FCN model was tested on batch size of 128 and 256 initially, which provided only marginally better and at times a lower accuracy score as compared to the batch size of 50. To take advantage of lower memory utilization and faster solution convergence with acceptable , batch size was set to 50.

Number of epoch (25): This parameter defines the number of forward and backward passes of the entire training dataset through the network. The idea is to increase the accuracy of the model without requiring more data. At first, this network was tested with 30 and 40 number of epochs resulting in long computation times and an increase in validation loss as well as training loss values after they hit respective minimum values around 22 epochs. This was a possible result of over-fitting of data giving a low overall accuracy score. Hence, the model was run again with 25 epochs which resulted in a better accuracy score and lesser training time.

Steps per epoch (200): It is the number of batches of training images that go through the network in 1 epoch. In order for the network to cover the whole dataset of images during training, a recommended value of this parameter is the number of images in training set divided by the batch size. With batch size set at 64 and total number of training images 4131, this parameter was set to a safe value of 200. Testing the network with a much lower steps per epoch like 80 and 100 did not improve the accuracy score.

Validation Steps (50): Similar to steps per epoch, this parameter defines the number of validation images that go through the network in 1 epoch. With 64 batch size and 1184 images in the validation dataset, the validation steps parameter was set to a safe default value of 50. The model was not tested with different values for this parameter.

Workers (4-32): This determines the maximum number of processes to spin up. While training the model on local machine with tensor flow CPU computation engine, the maximum number of workers was set at 4 providing a very minute performance increase in processing time as compared to a value of 2. However, on the AWS server with tensor flow utilizing the GPU, it was possible to increase the number of workers value which provided a boost in computation times.

➢ Improvements & Limitations

After running the model multiple times with numerous parameter tunings, the best accuracy score was reached at 41.02%. Increasing the number of filters (from 16 to 64 for the first encoder layer) was a big change that resulted in a boost to the final accuracy score. One major step towards increasing the accuracy further would be to get more trainable data. Collecting a set of good training data and training the model along with the provided one would make the solution more accurate. Alongside, training the network longer with an increased set of validation data would be helpful.

The current FCN architecture is designed to recognize and follow a human in the scene. For detecting other objects, additional classes or labels (dog, car, etc.) need to be added for semantic segmentation and model needs to be trained on images containing the desired object to be detected. If the different classes are highly unrelated, it would be better to have separate networks for each class. In case of different classes with some degree of relation, correlations between classes can be learned and a shared feature space can be represented by the same network leading to a speed-up performance.