Convolution Network ?

In convolutional neural networks (CNN) every network layer acts as a detection filter for the presence of specific features or patterns present in the original data. The first layers in a CNN detect (large) features that can be recognized very easy. The layers after that detect more (smaller) features that are more important (and are usually present in many of the larger features detected by earlier layers). The last layer of the CNN is able to make an special classification by combining all the specific features detected by the previous layers in the input data.

Fully Convolution Network ?

- A Fully Convolution Network (FCN) is a network architecture that allows preserving the spatial information throughout the network, which is very important to object detection and recognition in an image. also FCN can receive an input of any dimension.
- Also CNNs and FCNs both have an encoder section consist of regular convolutions. But Fully Convolutional Networks have a 1x1 convolution layer and a decoder part made of reversed convolution layers. Instead of a final fully connected layer like a Convolutional Neural Network.
- Also a FCN tries to learn representations and make decisions based on local spatial input. But appending a fully connected layer enables the network to learn something using global information where the spatial arrangement of the input falls away and need not apply.

Encoder?

- The encoder section consists of one or more encoder blocks with each including a separable convolution layer.
- Separable convolution is a convolution technique for increasing model performance by reducing the number of parameters in each convolution taking less time and save memory.
- A spatial convolution is performed first followed by depthwise convolution.
- The depthwise separable convolution is so named because it deals not just with the spatial dimensions, but with the depth dimension

```
def encoder_block(input_layer, filters, strides):
    # Create a separable convolution layer using the separable_conv2d_batchnorm() function.
    output_layer = separable_conv2d_batchnorm(input_layer, filters, strides=strides)
    return output_layer
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
def conv2d_batchnorm(input_layer, filters, kernel_size=3, 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
```

1X1 Convolution?

- The problem appears when convolutional layers are fed into a fully connected layer , the data is flattened which leads to loss of spatial information
- The 1X1 Convolution helps as to preserve spatial information while flattening the data at the same time which is a big advantage over the fully connected layer
- This allow us to feed images of any size into our trained network

Skip connections?

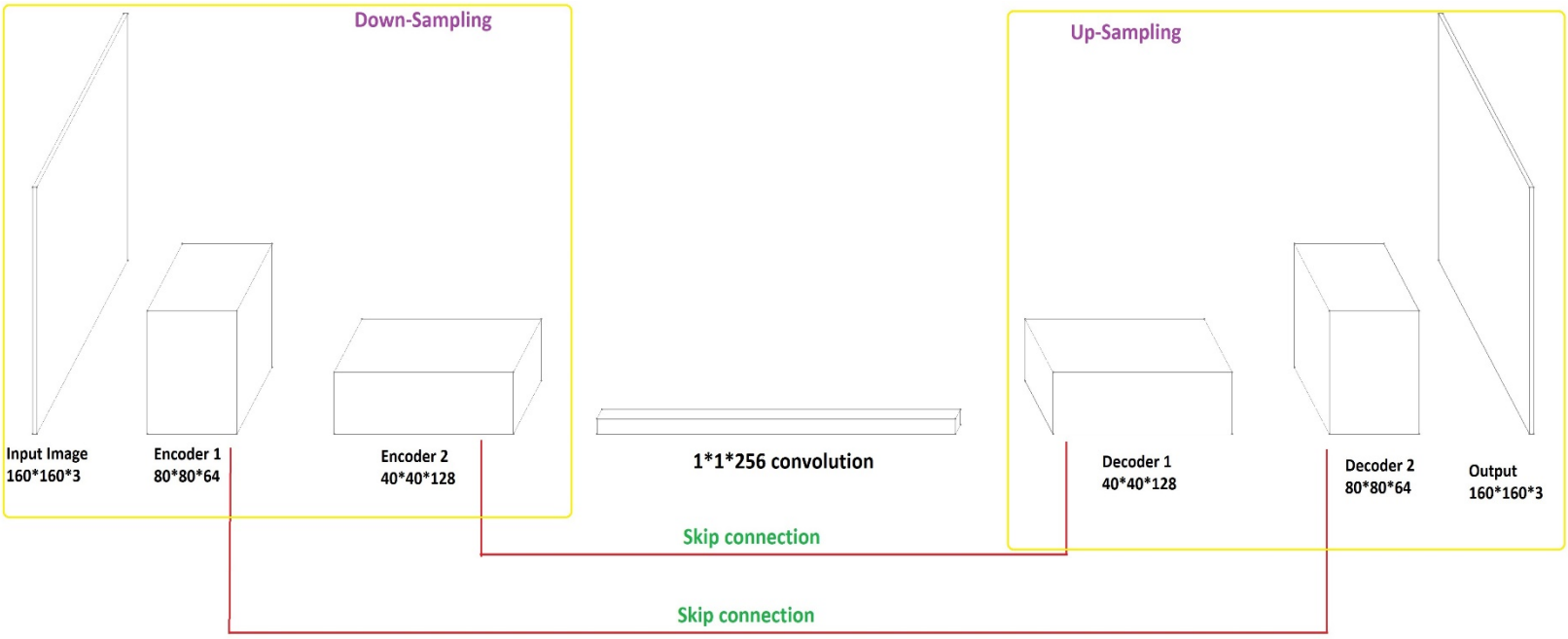
- One disadvantage of convolutions is that when we convolute more and narrow down the scope to look for more features
- What happens is the skipped layers from the encoders is combined with the decoder layers to get more spatial information that is lost during convolution this method is used during up sampling

Decoder?

- The Decoder section is either made of transposed convolution layers or bilinear upsampling layers
- The transposed convolution layers is the reverse of convolution (multiplying each pixel of the input with the kernel)
- Bilinear upsampling is the opposite of max pooling, it estimates the new pixel intensity value based on averaging neighbouring pixels.

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

    # TODO Upsample the small input layer using the bilinear_upsample() function.
    upsample = bilinear_upsample(small_ip_layer)
    # TODO Concatenate the upsampled and large input layers using layers.concatenate
    conc_layer = layers.concatenate([upsample, large_ip_layer])

    # TODO Add some number of separable convolution layers
    hidden_layer = separable_conv2d_batchnorm(conc_layer, filters)
    output_layer = separable_conv2d_batchnorm(hidden_layer, filters)

    return output_layer

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

My model?



- The FCN model used for the project contains two encoder block layers, a 1x1 convolution layer, and two decoder block layers.
- The first convolution uses a filter size of 64 and a stride of 2, while the second convolution uses a filter size of 128 and a stride of 2. Both convolutions used same padding. The padding and the stride of 2 cause each layer to halve the image size, while increasing the depth to match the filter size used, finally encoding the input within the 1x1 convolution layer uses a filter size of 256, with the standard kernel and stride size of 1.
- The first decoder block layer uses the output from the 1x1 convolution as the small input layer, and the first convolution layer as the large input layer, thus mimicking a skip connection. A filter size of 128 is used for this layer.
- The second decoder block layer uses the output from the first decoder block as the small input layer, and the original image as the large input layer, again mimicking the skip connection to retain information better through the network. This layer uses a filter size of 64

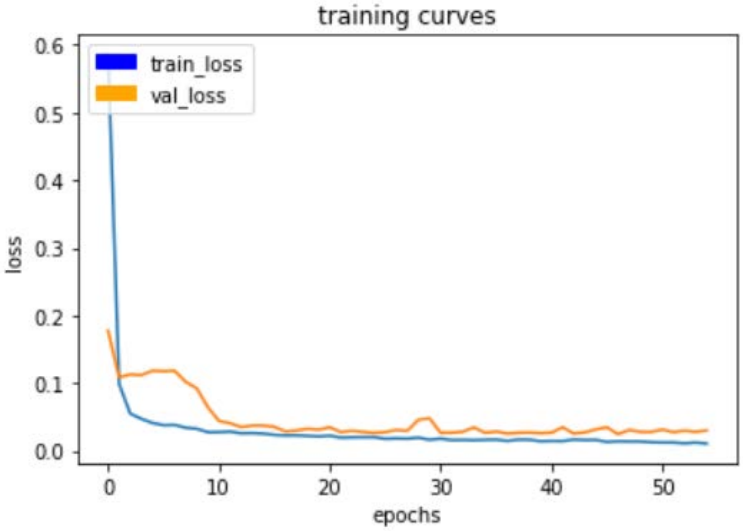
```
def fcn_model(inputs, num_classes):

    # TODO Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of your model (the number of filters) increases.
    input_layer0 = encoder_block(inputs, 64, 2)
    input_layer1 = encoder_block(input_layer0, 128, 2)
    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    input_layer2 = conv2d_batchnorm(input_layer1, 512, kernel_size=3, strides=1)
    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    output_layer3 = decoder_block(input_layer2, input_layer0, 128)
    output_layer4 = decoder_block(output_layer3, 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, 3, activation='softmax', padding='same')(output_layer6)
```

Hyperparameters

- learning_rate = 0.001
- batch_size = 64
- num_epochs = 50
- steps_per_epoch = 65
- validation_steps = 50
- workers = 120

Performance

The final score of my model is **0.42 and IOU 0.569**



As we notice after 50 epochs we started to reach overfitting