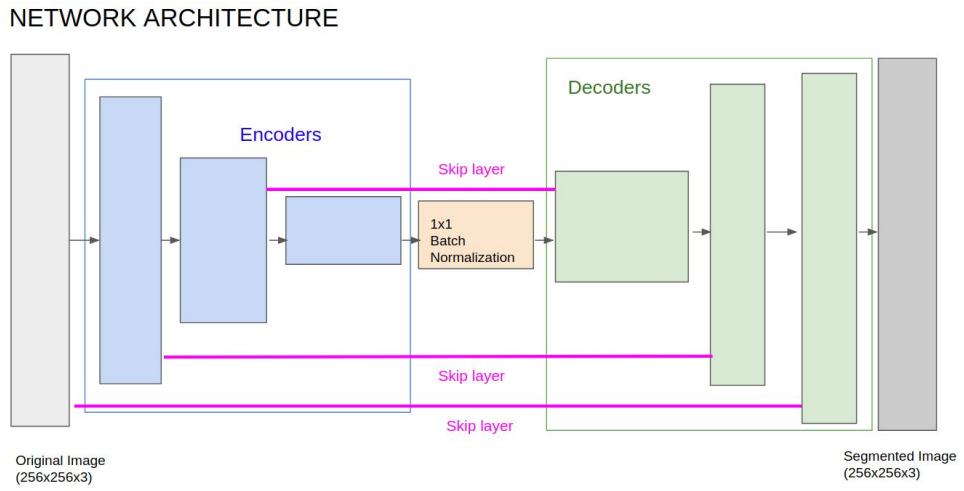
Udacity-Robotics NanoDegree
Project 4: Deep Learning - Follow Me Project
By: Stacey Ng

---

1. Network Architecture

The network architecture i used in this project now is a 7 layer network. I've added another encoder and decoder layer. So it has 3 encoding layers , 1 1x1 convolution layer and 3 decoding layers. Compared to my 5 layer network used previously.



The implementation is found in the function fcn_model.

```
def fcn_model(inputs, num_classes):

    # Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of your model (the number of filters)
    # The shrinking of the layers ia achieved by changing stride
    # We also use more depth as illustrated by 32 and then 64
    # Initially I had 16 and 32 and the higher values perform much better without incurring t
    l1 = encoder_block(inputs, 32, 2)
    l2 = encoder_block(l1, 64, 2)
    l3 = encoder_block(l2, 128, 2)

    # Add 1x1 Convolution layer using conv2d_batchnorm().
    # This is implemented as regular convo so set stride and kernel to 1 for it to be 1x1
    l4 = conv2d_batchnorm(l3, 128, kernel_size=1, strides=1)

    # Add the same number of Decoder Blocks as the number of Encoder Blocks
    # Add decoder blocks and the skip layer connections
    # If you look at the implementation of the decoder the elementwise addition is happening
    # after the upsample. So we have to connect the layers of correct size.
    l5 = decoder_block(l4, l2, 128)
    l6 = decoder_block(l5, l1, 64)
    l7 = decoder_block(l6, inputs, 32)

    x = l7
    # The function returns the output layer of your model. "x" is the final layer obtained fr
    output = layers.Conv2D(num_classes, 1, activation='softmax', padding='same')(x)

    # Print network shape
    print (inputs)
    print (l1)
    print (l2)
    print (l3)
    print (l4)
    print (l5)
    print (l6)
    print (l7)
    print (output)

    return output
```

Encoder Block

As there are 3 layers of encoder blocks, each block is meant to compress the output from the previous later to produce certain features. The compression is done by a convolution layer with stride > 1. Convolution helps preserve spatial information which is helpful in detecting the features we desire when "following" the person. I used the same padding in all cases so as to simplify the process across the dimensions of the layers. The encoder block implementation is as follows, in the encoder_block function :

```
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
```

1x1, Batch Normalization

The middle layer is the 1x1 and batch normalization layer. The batch normalization helps in better learning and also introduces some randomness to help in regularization and reduce biasness.

```
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
```

Decoder Blocks

The decoding layers are the opposites of the convolution layer and they perform the deconvolution step. This is done by:

- Upsampling :To revert the stride of convolution which has caused shrinkage
- Concatenating information from the skip step connections
    - Information from skip step connections give the model some context
- Adding a layer of separable convolutions to counter the convolution in the encoder block and batch normalization.

The implementation is as such, in the decoder_block function:

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

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

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

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

    return output_layer
```

Skip Connections Step

I used skip connections between the encoders and the decoders to provide the model with some context of the original image, regularization and prevent overfitting. It is important to have the layers to be of the same size as the outputs are to be added together.

2. Parameters Chosen for the Neural Network

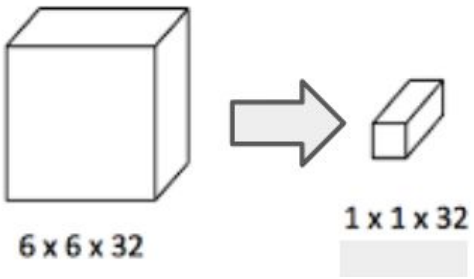
Parameters	Values Chosen	Remarks
Batch Size	40	How many images sent to the network in 1 step. Limit batch size to limit amount of memory used.
Steps per epoch (validation steps)	100	Full forward and backward iteration in one batch
Epochs	40	Full forward and backward pass
Learning Rate	0.003	Learning rate tunes the proportion of the weighted change during the backward propagation step
Workers	10	Workers control the number of parallel processes during training I previously did not adjust these values. However the change was not significant.
Validation Steps	30	Number of validation images over batch size. My aim is to test all data on every epoch

From my previous submission, the parameter i started tuning first after modifying the network design was the learning rate, i started again from a value of 0.001. Just to have a baseline measurement of what it was or was not learning. Next i decreased the epoch value as since i had an additional layer for encoding and decoding. I eventually stopped at 40 for both batch size and epoch as i felt the learning was happening more slowly and it won't result in much difference if i were to to increase it further.

Next i decreased the learning rate. I reduced it from 0.001 to 0.002 and I arrived at a final score of 0.38 which was better than the previous 0.001 model (final score 0.36). So i reduced it further and decided on 0.003 as the learning rate.

I would take approximately 8 hours to train the model on my own desktop PC and it was a huge improvement compared to my previous computation which took around 20 hours.

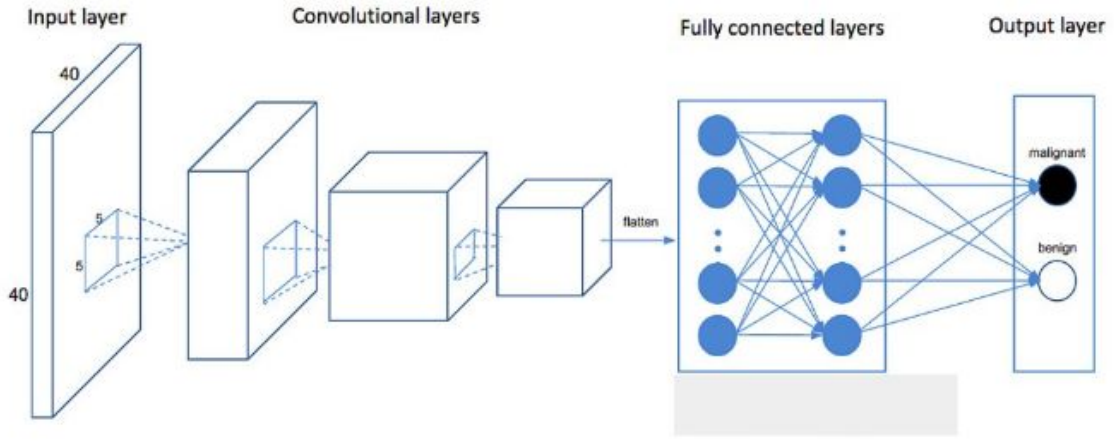
3. Understanding 1x1 Convolution



What a 1x1 convolution does to my understanding is that it collapses all channels in an input pixel to ONE pixel. This reduces the dimensionality in the filter space. Thus 1x1 convolution layers are used to compute reductions before the bigger 3x3 or greater convolutions. Reduction of dimensionality also results in faster computation as it can be implemented as matrix multiplication.

Also as mentioned above, the 1x1 kernel width and height is preserved , thus retaining the depth which contains the spatial information (w, h, retained). It is also a cheap and efficient way of making a network deeper.

4. Understanding a Fully Connected Layer



A fully connected layer is defined as a traditional multi-layer perceptron that uses softmax in the output layer. Fully connected means that every neuron in the previous layer is connected the every neuron in the next layer.

As output from multiple convolutional layers represent high-level feature extractions in the data of the input image. Adding a Fully connected layer is a cheap and efficient way of learning or using these features for classifying the input image to various classes or even discovering non-linear combinations.

5. Understanding Image Manipulation

Encoder is a convolution network which takes the input and outputs features. They can be in the form of vectors, maps or tensors. They hold the (extracted features) which represent the input. This is done through applying multiple layers of filters. Through several layers of encoding, simple features can be used in combinations to learn mre about complex features. For example an edge to a corner then to a jawline of a face etc.

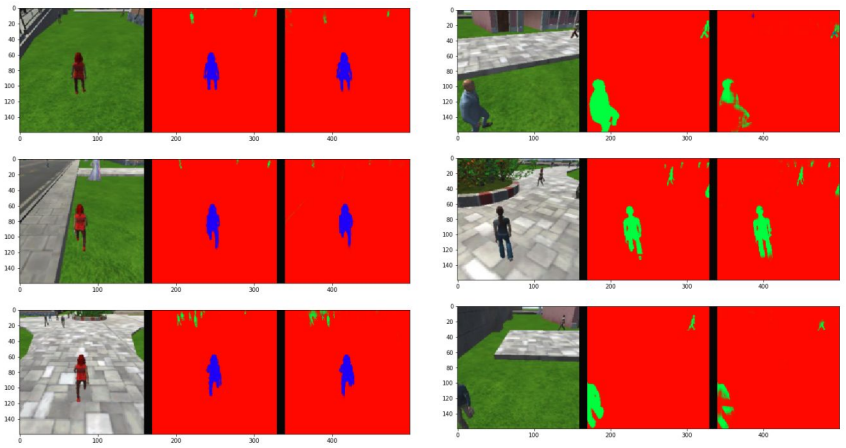
Decoders on the other hand performs a deconvolution. From the features extracted in the encoders, it is then used back to reconstruct the original image (thus returning the image to its original size) .

Possible problems which may arise when designing a network is that convolution performs reduction. It works on a particular area, and the result is smaller. We have to deal with this reduction by applying some padding. Another problem is the computational effort of convolution. Generating multiple feature layers in multi-dimensional convolutions is very heavy on computation. One work around is to apply depth separable convolutions and also skip step layer connections. Skip step layer connections are done from encoder to decoder. It provides the layer with more context to the image, thus preserving spatial information.

6. Discussion and Results

Overall, the quad rotor took a while to locate the target but once their paths coincided, i was rather impressed with the results and it was highly satisfactory to know that it worked. Each training took about 8 hours now and i was not sure about how to use the GPU computing of CUDA and CUDNN even though i installed both. Maybe more instructions on how we could utilise them would be helpful.

I've got my results and values of my parameters largely through trial and error from my previous submission and i added an additional encoder and decoder layer.



Overall my results from the training was

- Final score: 0.419

Limitations of this network :

In my opinion, I do not think that this model would work well if animal images were introduced. This is because a bulk of the images from its “library” consists of images containing humans and it has not been fed any animal images during the training phase. Furthermore, the extracted features in this current model are those of a human being, thus making it recognise people more than animals.

However, I think the network architecture can be kept the same, just that the data which is being fed into it matters. We would just have to feed it correctly labelled images of animals or whatever desired object which is to be detected, so that the network can properly tune itself by repeating the training phase.

Suggestions for Improvements :

- Custom data gathering
- Further reducing the learning rate (although it involves more training time, unless GPU computing)

I’ve begun some work on customising my own data gathering paths so as to see if i can get a better score. This revolves making the drone fly to capture all angles of the target. (by making the target zigzag) . However, this also means that i need to have datasets to include other humans so that the network will be able to differentiate them as well.

Below are some custom paths :