Project: Deep Learning Follow Me

Introduction

In this project we train a fully-convolutional neural network to segment information out from images so that we can identify between our ‘hero’ target and everyone else.
This allows us to enable a drone to follow the hero even in a dense crowd.
The model is evaluated by an IOU score(Intersection Over Union) score, which takes the intersection between prediction pixels and ground truth pixels and divides it by the union of them both.

That gives us an idea of how close the models predictions are to absolute ground truth.
In this report I will be talking about Network Architecture, Design Choices, Experiments, and Conclusion.

Network Architecture

We need to use Fully Convolutional Networks for this architecture because the usual architecture of convolutional layers followed by fully connected layers won’t work in this scenario because we need to preserve spatial information in the model and fully connected layers lose that spatial information.
Therefore every layer in the model needs to be a convolutional one so that the spatial information isn’t lost.

The model includes encoder layers and decoder layers connected in between by a 1x1 convolutional layer.

The Encoder layers extracts useful information from the image for segmentation. It has multiple layers which allows it to first understand simple geometrical concepts in the image like lines, circles, dots and other rudimentary shapes whilst the subsequent layers understand more complex concepts like head, hands, legs etc.

The 1x1 Convolutional layer that connects Encoder layers to the Decoder layers acts as a fully-connected layer would in a generic conv net architecture except that it preserves spatial information.

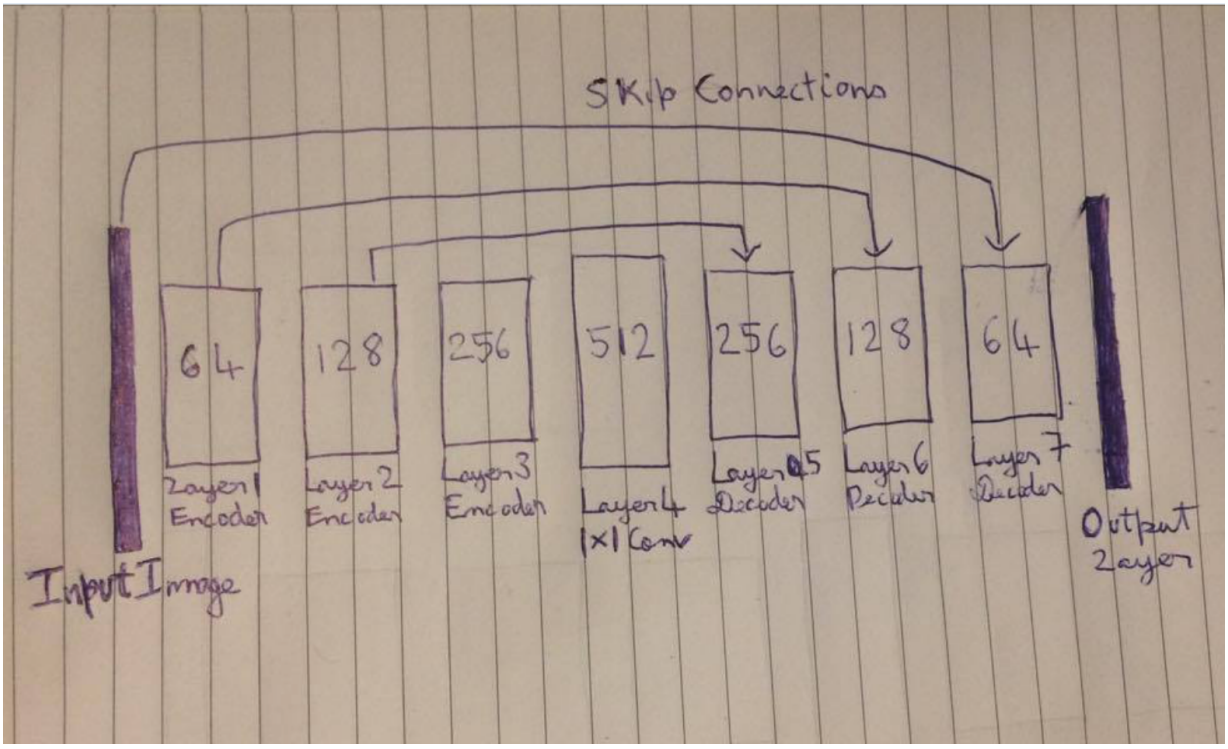
The Decoder layers upscale the encoder output back to the dimensions of the input image using techniques like bilinear upsampling. We also use skip connections so that

certain information lost in between the encoder and decoder layers are concatenated back.

The final output layer is a convolutional one with softmax activation to segment image data by classifying pixels into 3 classes of R, G and B.

As you can see from the image below, there are 3 encoder and decoder layers of depths 64, 128 and 256 connected by a 1x1 convolutional layer of depth 512.

The Decoder also has 1 extra separable convolution layer than usual after the up sampling and concatenation steps.



Network Architecture

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

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

    return output_layer
```

Decoder block with an extra separable convolutional layer

Design Choices

Optimizer

Chose **Adam** optimizer which was the default initially with a learning rate of 0.01 which is optimal for this case.
This makes sure that it neither overshoots convergence by taking too big of a step, nor takes a huge amount of time to converge by taking too small of a step. If the step is too small there's also a change it gets stuck at the local minima instead of converging to the global minima.

Momentum is a concept that makes sure the steps taken toward a minima(an optimal point) is more prominent.
This has the double benefit of converging faster as well as avoiding getting stuck in a local minima.
Nesterov Momentum goes a step forward by taking into account the momentum of the next step as opposed to the momentum of the current step. This allows it to take smarter decisions.

So I chose **Nadam** optimizer which uses Nesterov Momentum and converges much faster.
Experimented with learning rates of 0.002, 0.005 and 0.001.

Batch Size

We run the model over the data in batches so that its feasible computationally and trains faster.

Large batch sizes tend to be bad at generalizing as they converge on sharp minimizers of the loss function.
Small batch sizes and learning rates are linked, in that if batch size is too small then gradients become more unstable and need a smaller learning rate.

Tensor flow optimizes resources for batch sizes if its to the power of 2.
So I chose 64.

Steps per Epoch

No of training images or validation images over batch size as we need to train and validate over the entire data on every epoch.

```
steps_per_epoch = no_of_training_pics // batch_size + 1
validation_steps = no_of_validation_pics // batch_size + 1
```

Workers

This determines the no of parallel processes during training. Experimented with 2, 5 and

10 but it didn't seemed to speed up or slow down training.

Experiments

Initial attempts of the model didn't include the extra separable conv layer in the decoder and the depth was also starting at 32 and so the depths of the encoder and decoder layers were 32, 64 and 128 with a 1x1 conv layer of depth 256 in between.

Because of this I was stuck in the range of 0.20 - 0.26.

With Adam optimizer I needed 60-100 epochs.
But thanks to Nadam optimizers use of momentum, the convergence was much faster and I needed only 30-40 epochs.

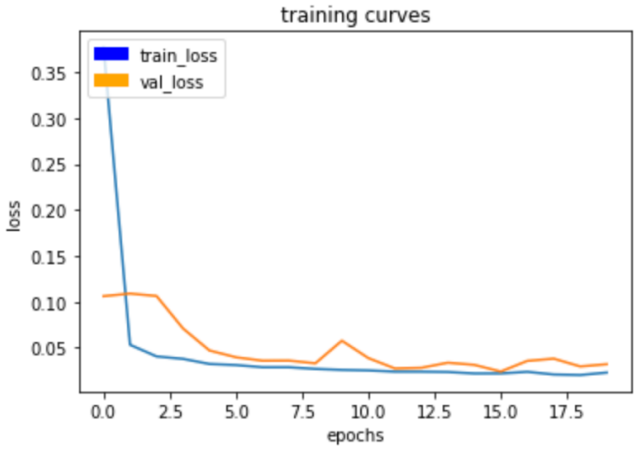
After discussing on slack I decided to up the depth to 64 as well as add an extra separable conv layer in the decoder.
This allowed me to reach around 0.37 to 0.38 IoU score with around 20-30 epochs.

I then decided to use another popular technique for training which was augmenting the data by flipping the images.
By having the extra data, the model learns better.

```
def flip_images(image_dir, extention):
    os.chdir(image_dir)

    images = glob.glob('*.' + extention)

    for index, image in enumerate(images):

        pic = misc.imread(image, flatten=False, mode = 'RGB')
        flipped_pic = np.fliplr(pic)

        misc.imsave('flipped' + image, flipped_pic)

    print('Saved')

#flip_images('/home/ubuntu/RoboND-DeepLearning-Project/data/train/masks/', 'png')
#flip_images('/home/ubuntu/RoboND-DeepLearning-Project/data/train/images/', 'jpeg')
#print(os.getcwd())
```

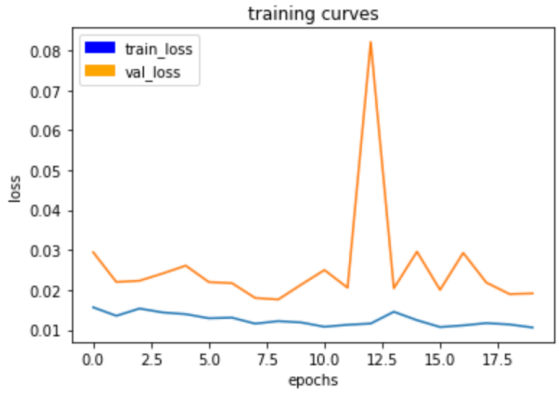
```
Saved
```

In the end I split the process into two 20 epoch ones in which the first 20 I used the formula shown in the previous section to calculate training and validation steps.
The second 20 epochs I used a 100 training steps with 25 validation steps.
My final score of IoU was 0.45583.




```
130/130 [==============================] - 100s - loss: 0.0229 - val_loss: 0.0320
```

The first 20 epochs




```
100/100 [==============================] - 221s - loss: 0.0106 - val_loss: 0.0191
```

The second 20 epochs

Conclusion

Even though this project seemed straightforward at the beginning, since we were mostly using code from the segmentation lab, it still turned out to be very challenging because of the threshold for IoU.

Learnt a lot about the importance of different types of optimizers, the need to have a lot of data(and its quality) as well as network architecture through the intricacies of convolutional layers and its dimensions.

This model can be improved by getting more images of the hero at a distance which was the defining metric keeping the scores low.

In general it would also be beneficial to have more images with the hero present. Currently its only 37% of the images.
I would like to build upon that premise by collecting custom data in both near and far scenarios.

Note: this particular model will not work for following other objects (e.g. dots, cats, cars) because we only trained using images of humans and specifically distinguish between a hero who is distinct(wears red top, black jeans) versus everyone else.

However this technique can be used for following cats, dogs, cars if we trained the model with those images.
In fact a major part of autonomous driving tech is about segmenting data from a scene and identifying common objects such as traffic signs, dividers, trees, bikes, people, signals etc.

Recently Apple released a paper which talks about segmenting object information from RGB-D camera data, this technique is called VoxelNet.
Below you can see the similarity in architecture of the Region Proposal Network(1 of the 3 networks in this technique) to our own project's network architecture.

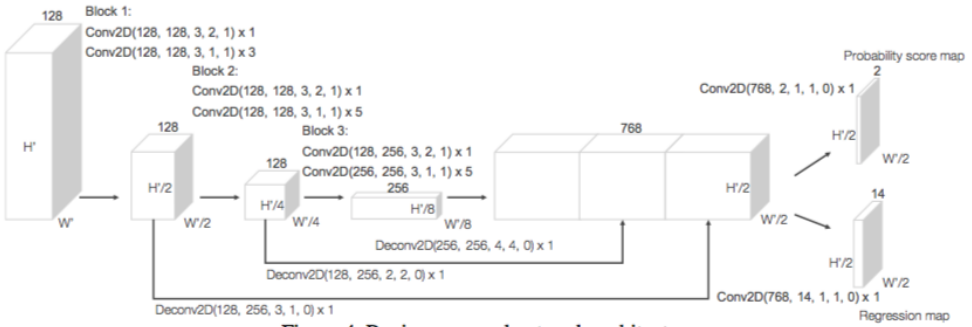


Figure 4. Region proposal network architecture.

https://arxiv.org/pdf/1711.06396.pdf