Udacity Robotics Nanodegree
Project 4
Deep Learning Project: Follow Me

Frank Zerofsky

February 4, 2018

Objective:

The objective of this project is to apply a method called semantic segmentation to a set of images taken from an aerial drone. The aerial drone will use the segmented images to follow a specific target within a simulation environment. Semantic segmentation is used to both identify objects and their location within the scene. This specific project identifies a person wearing a red shirt referred to as the target or hero. The aerial drone must search for the target, and once found, follow the target.

FCN Architecture:

In order to implement a solution to this problem, a fully convolutional network (FCN) must be built and trained with a training data set. The training data set is a series of images taken from the perspective of a camera mounted to the aerial drone. A diverse set of images are needed to properly train the network. These include images with the target at short and long distances along with alternate angles. The images also include non-targets and the environmental surroundings.

The fully convolutional network is made up of three main parts the encoder, 1x1 convolution, and the decoder. The encoder is a set of fully connected convolutions which down sample the input image while increasing the number of filters with each convolution.

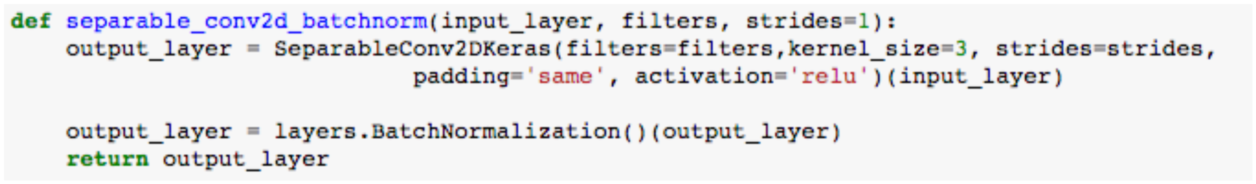
The input image starts as a 160 x 160 x 3 (height x width x depth) tensor. The first encoder is set with 2^5=32 filters and a filter size of (3 x 3). Encoder 2 has 2^6=64 filters and a filter size of (3 x 3). A stride of 2 is used for each convolution with same padding. This results in lost detail as the output layer is 80 x 80 x 32 from encoder 1. Encoder 2 further decreases the size of the layer while increasing filter depth to 40 x 40 x 64. The formula and evaluation for each encoder output layer size is shown below [1].

$$output_{height} = \frac{input_{height} - filter_{height} + 2 * P}{S} + 1$$
$$output_{width} = \frac{input_{width} - filter_{width} + 2 * P}{S} + 1$$

Encoder 1	
Filter size (height x width)	3 x 3
Stride (S)	2
Padding (P)	1 (Same)
Output Shape	80 x 80 x 32

Encoder 2	
Filter size (height x width)	3 x 3
Stride (S)	2
Padding (P)	1 (Same)
Output Shape	40 x 40 x 64

Batch normalization is used between each convolution in the network in order to allow for higher learning rates [2]. The basic idea is to treat each convolution in the network as its own neural network. By normalizing the input to each layer, the network is able to train faster.

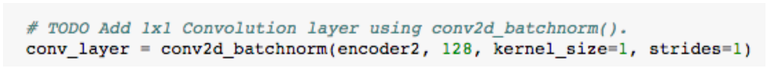


```
def separable_conv2d_batchnorm(input_layer, filters, strides=1):
    output_layer = SeparableConv2DKeras(filters=filters,kernel_size=3, strides=strides,
                             padding='same', activation='relu')(input_layer)

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

Figure 1: separable convolution function with batch normalization

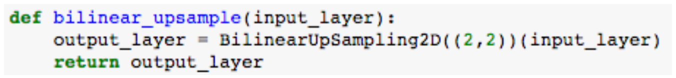
Encoder 2 is fed into a 1x1 convolutional layer. The 1x1 convolution is used to preserve spatial data about the image classifications. The number of filters in this layer is 2^7=128. The 1x1 convolution does not flatten the tensor to two dimensions as happens in a fully connected output layer. By using the 1x1 convolution, the spatial data of the tensor is preserved. This is important for this project application, in order to not only identify the target within the image, but where the target is to make control decisions for the aerial drone system. The below figure is the code for the 1x1 convolution where the kernel size is set to 1 and the stride is set to 1.



```
# TODO Add 1x1 Convolution layer using conv2d_batchnorm().
conv_layer = conv2d_batchnorm(encoder2, 128, kernel_size=1, strides=1)
```

Figure 2: 1x1 convolution layer

The output of the 1x1 convolutional layer is then up-sampled in the decoder layers back to the original size. This is done using a process called bilinear up-sampling. Bilinear up-sampling works by linearly interpolating the pixel intensity value from the smaller image pixels to the larger layer. This process is done twice in the FCN, one time per encoder layer.



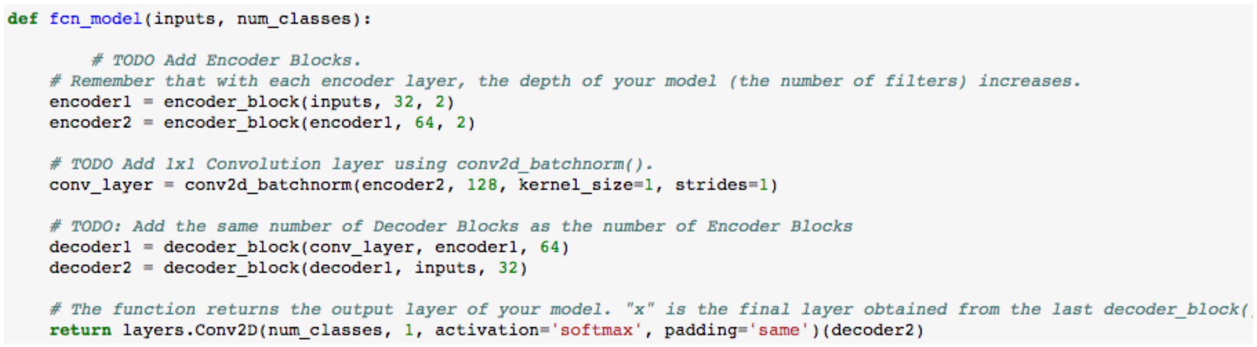
```
def bilinear_upsample(input_layer):
    output_layer = BilinearUpSampling2D((2,2))(input_layer)
    return output_layer
```

Figure 3: bilinear up sampling function

The final attribute of this fully convolutional network are the skip connections. The skip connections are used to add information back to the image after the bilinear up-sampling process. During the down-sampling process, the details of the image are lost during classification in order to create a more computationally efficient network. The skip connections are implemented using a layer concatenation technique. This technique allows different depth input layers to be combined. Therefore, the layer with more spatial information can provide information to the classified layer.

The final output layer of the network is a softmax activation function which outputs the probability value for each pixel classification. This is used for determining which pixels belong to the target.

The final fully convolutional network architecture is shown below in both code and illustration form. Note in the illustration, the boxes for each network layer are not to scale, but the filter depths are labeled.



```
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.
    encoder1 = encoder_block(inputs, 32, 2)
    encoder2 = encoder_block(encoder1, 64, 2)

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    conv_layer = conv2d_batchnorm(encoder2, 128, kernel_size=1, strides=1)

    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    decoder1 = decoder_block(conv_layer, encoder1, 64)
    decoder2 = decoder_block(decoder1, inputs, 32)

    # 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')(decoder2)
```

Figure 4: code for FCN function

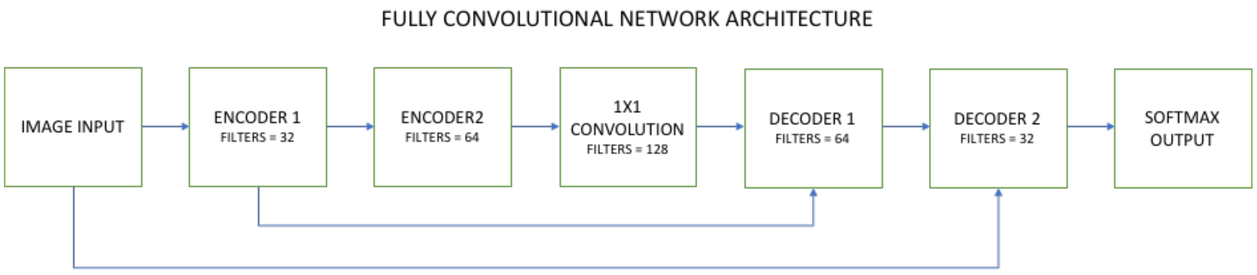


Figure 5: FCN architecture

Data Collection:

Additional data was collected within the simulator to augment the supplied data in the event that the supplied data was not sufficient. Data was collected for the training set and the validation set. Each of the data sets were taken at different places within the simulator environment. Images of the path for the hero, aerial drone, and bystanders are shown below for both data collection sets.

The green dots indicate the path of the aerial drone for data collection, the pink dots represent the path of the target hero, and the blue dots are spawn points for non-target pedestrians.

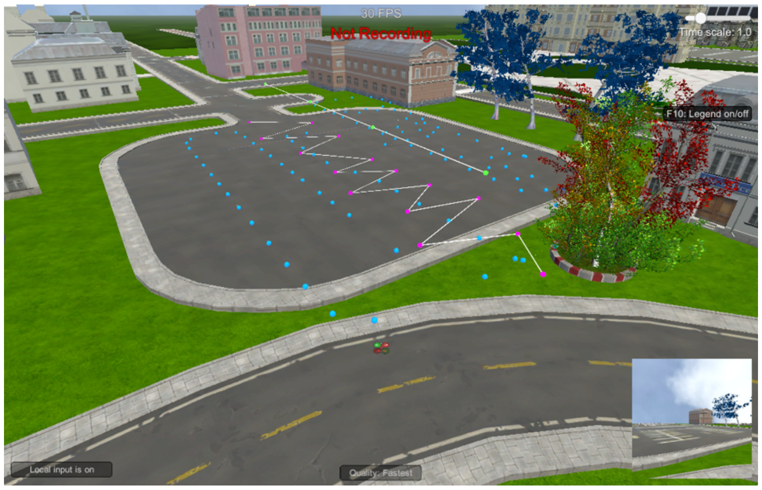


Figure 6: data acquisition paths for training data set

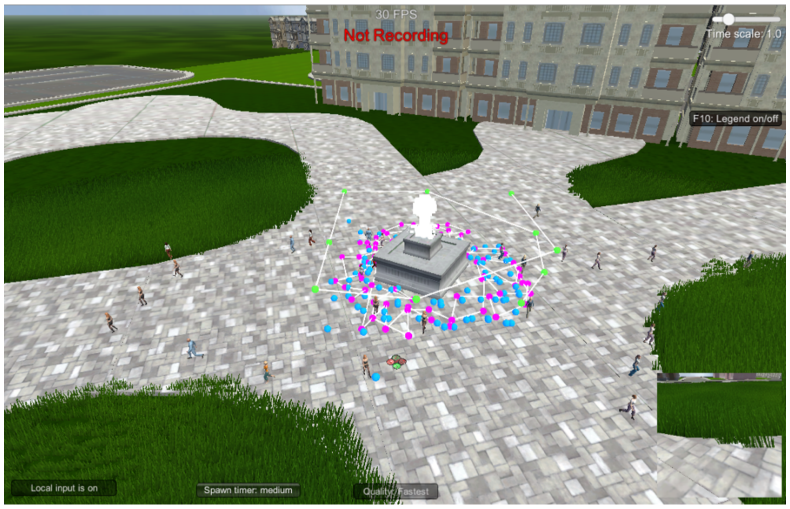


Figure 7: data acquisition paths for validation data set

Model Training and Hyper Parameters:

The model was developed using the Keras API for TensorFlow. Keras allows for network parameters to be easily and quickly adjusted between training runs. The following hyper parameters below were adjusted to optimize the training of the model.

<u>Epochs</u>:
An Epoch is one cycle forward and backward through the number of inputs set by the batch size. By increasing the number of epochs, the total amount of data seen by the network in increased.

Learning Rate:
When using stochastic gradient descent the learning rate is a constant value multiplied by the gradient of the loss value. This essentially effects the rate of change for the weights in the model. By having a smaller learning rate, the system will learn slower. The advantage to having a smaller learning rate allows the model to converge the loss function more efficiently.

Batch size and steps per epoch:
The batch size is the number of data inputs sent through the network in a single pass. The steps per epoch hyper parameter describes the number of batches of training data to go through the network in a single epoch. This value is recommended to be set to the number of training data points divided by the batch size. This will allow all the images to be seen by the network. The validation steps per epoch is the same as steps per epoch, but it applies to the validation data set.

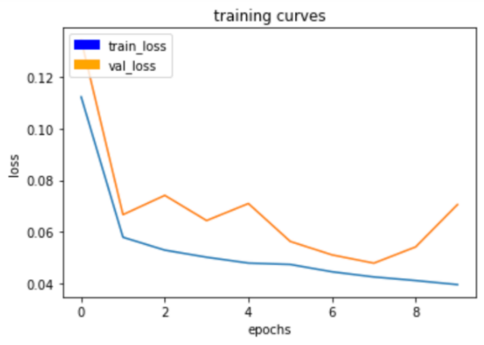
Workers:
The number of workers allows more processes to work simultaneously. The default setting for this project was 2. This value was kept constant because the network training was performed on a single AWS EC2.xlarge instance.

The network's hyper parameters were tuned manually between each training session. An initial starting point was picked in order to set a baseline. One to two parameters were then changed between runs depending on the results of the previous run. The training run hyper parameters and final scores are outlined below.

Training Run #1:

Learning Rate	Batch Size	Epochs	Steps	Val Steps	Workers	Final IoU Score
0.1	32	10	200	50	2	0.335

The first training run was a test to make sure the network was working properly, and the AWS instance was setup properly. A small batch size and number of epochs were chosen in order run the first training session quickly. This run allowed a baseline to be set for further training refinement.



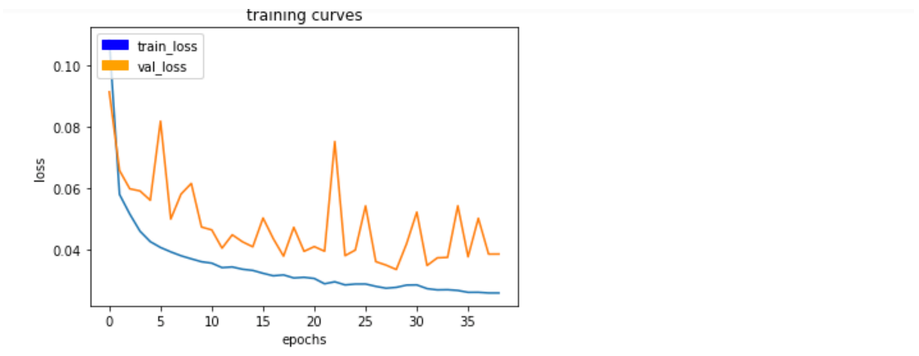
```
200/200 [==============================] - 72s - loss: 0.0398 - val_loss: 0.0706
```

Figure 8: training results run 1 loss vs. epochs

Training Run #2:

Learning Rate	Batch Size	Epochs	Steps	Val Steps	Workers	Final IoU Score
0.1	64	50	200	50	2	0.400

Based on the results of training run #1, the FCN did not have enough epochs to properly converge. Therefore, the epochs were increased from 10 to 50. By increasing the batch size, more images will be sent through the network in each epoch to allow a larger pool of data to be used by the training network.



```
200/200 [==============================] - 148s - loss: 0.0260 - val_loss: 0.0387
```

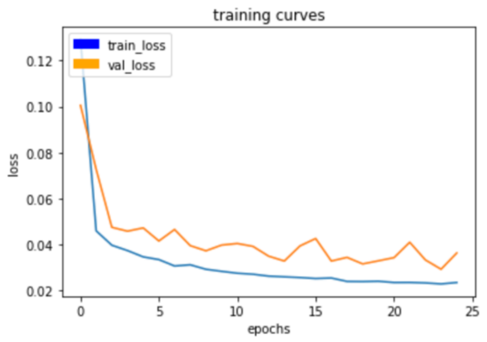
Figure 9: training results run 2 loss vs. epochs

Based on the runtime results for run 2, the training was stopped at 40 epochs. This was based on the validation loss not converging as expected. With only 10 epochs left in the training, the further computation was not deemed justified.

Training Run #3:

Learning Rate	Batch Size	Epochs	Steps	Val Steps	Workers	Final IoU Score
0.01	64	25	200	50	2	0.407

The learning rate for this training run was decreased by a factor of 10 to 0.01. This was done to decrease the validation loss value by creating the smaller updates to the weight after every epoch. The number of epochs was also decreased based on the results from training run 2. This was done to decrease the total training time.



```
200/200 [==============================] - 149s - loss: 0.0234 - val_loss: 0.0362
```

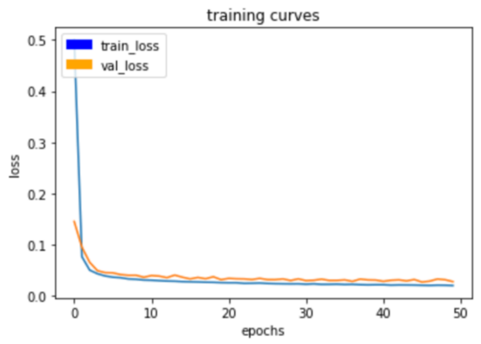
Figure 10: training results run 3 loss vs. epochs

The system started to converge better with the smaller learning rate. The validation loss still has not converged fully, therefore a smaller learning rate is to be investigated in the fourth training run along with a larger number of epochs.

<u>Training Run #4</u>:

Learning Rate	Batch Size	Epochs	Steps	Val Steps	Workers	Final IoU Score
0.001	64	50	200	50	2	0.412

The learning rate was further decreased due to the non-convergence of the validation loss value. Another factor of 10 was applied to the learning rate from the third training run. The number of epochs is increased as well as discussed above.



```
200/200 [==============================] - 148s - loss: 0.0205 - val_loss: 0.0281
```

Figure 11: training results run 4 loss vs. epochs

The training results of the fourth training run converged and settled at loss values of approximately 0.02 for both the validation and training sets. Both loss value curves are much smoother compared to the previous training runs where a coarser learning rate was used.

Final Results:

The final results of the network performance are determined using the intersection over union (IoU) method. The final training run accomplished a IoU score of 0.412 with the validation loss and the training loss values converged at approximately 0.02. The aerial drone was able to find and track the target in the simulator.

In order to further improve the training model, more data could be added to the training set. Another way could be to further tweak the hyper parameters. The hyper parameters for steps per epoch (training and validation) and workers were not adjusted for training this network. Optimization of the hyper parameters could yield better performance. The FCN model could also be changed to include more encoder and decoder layers with higher filter dimensions. This would increase training computation cost, but coupled with more training data could see appreciable gains in network performance.

The model for the fully convolutional network could be used to identify a different object such as a dog, cat, or car. In order to successfully identify a different target other than the "hero", the network would need to be retrained with a new dataset containing the desired target. Therefore, this specific project is only relatable to finding the hero in a red shirt, but with a new dataset could be used to find a new target.

Files:

GitHub repository: https://github.com/fzero6/deeplearning-follow-project.git

Files:

- model_training.ipynb
- follower.py
- preprocess_ims.py
- model_weights.hd5
- config_model_weights.h5

References:

[1] Udacity, "Introduction to Robotics," [Online]. Available: https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e.

[2] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," 2 March 2015. [Online]. Available: https://arxiv.org/pdf/1502.03167.pdf. [Accessed 2018].

[3] Udacity, "RoboND-DeepLearning-Project," [Online]. Available: https://github.com/udacity/RoboND-DeepLearning-Project.