JANUARY 27, 2019

RoboND-DeepLearning-Project

GOALS OF THE PROJECT:

The goal of this project is to build a segmentation network, and train it to identify a target person from images produced by a quadcopter simulator. Once you have a trained network, you can use it to find the target in the environment and follow its path.

PROJECT STEPS:

1. Setting up your local environment: This is how you will test your initial network design to make sure there are no errors. Otherwise, you can use up your Workspace GPU hours or rack up charges on Amazon Web Services (AWS) with simple debugging instead of training. You will also use your local environment to evaluate your Workspace/AWS trained model with the Follow Me simulator.

2. Brief overview on how the simulator works and its basic controls.

3. Collecting data from the simulator to train your network.

4. Building your neural network.

5. Setting up your Classroom Workspace or, if you prefer, setting up your AWS Amazon Machine Images (AMI).

6. Training your network and extracting your final model and weights from Udacity GPU Workspace or AWS Instance.

7. Downloading your model from cloud and testing your model with the Follow Me simulator.

8. Getting your project ready for the final submission!

RUBRIC POINTS:

BUILDING ENCODER FUNCTION

1ST Separable Convolutions function

1) **Description**

   Separable convolutions, also known as depthwise separable convolutions, comprise of a convolution performed over each channel of an input layer and followed by a 1x1 convolution that takes the output channels from the previous step and then combines them into an output layer.

2) **Advantage over regular convolution**

   Gives us decent decrement in number of parameter that will reduces process resources needed and decreases training time

3) **Code**

```
output = SeparableConv2DKeras(filters, kernel_size, strides,padding, activation)(input)
```

2rd Batch normalization

1) **Description**

   Batch normalization is based on the idea that, instead of just normalizing the inputs to the network, we normalize the inputs to layers within the network. It's called "batch" normalization because during training, we normalize each layer's inputs by using the mean and variance of the values in the current mini-batch.

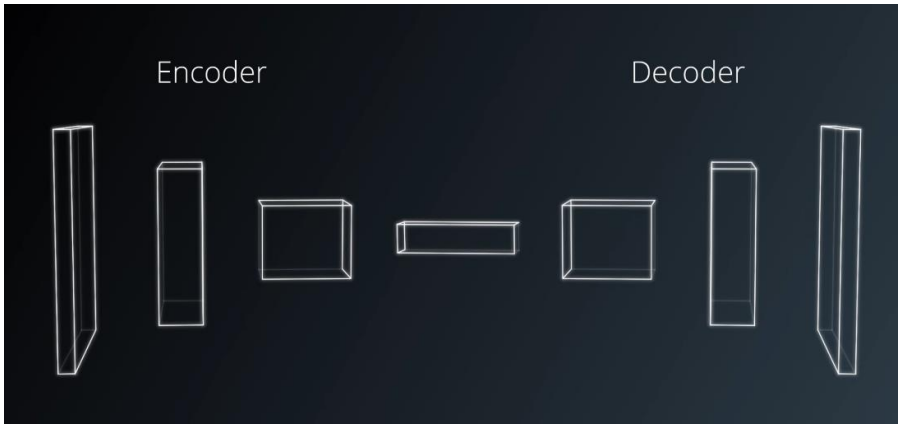
2) **Code**

```
output = layers.BatchNormalization()(input)
```

3rd Full encoder block

3) Parts

- separable convolution
- normalization to keep convolution data around zero mean
- size decrease with each encoder layer but depth increases

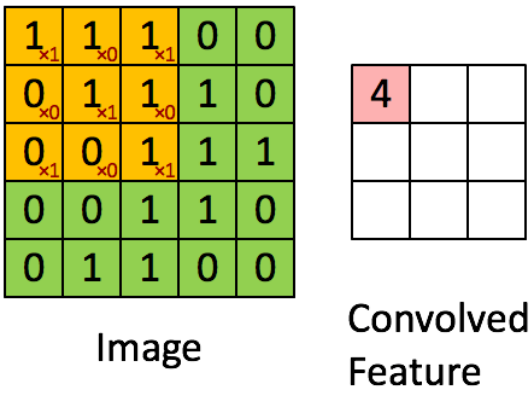


BUILDING DECODER FUNCTION

1nd Normal convolution function

1) FUNCTION

- Important when dealing with images because images has a lot of data in one sample.
- Basic idea of normal convolution is squeezing images to smaller size without losing important details.
- This is made by summing all data in one patch and saving it to the new layer pixels.
- This operation gives us the ability to recognize shapes and objects in any position of picture.
- Also, this decreases needed weights tremendously.



-

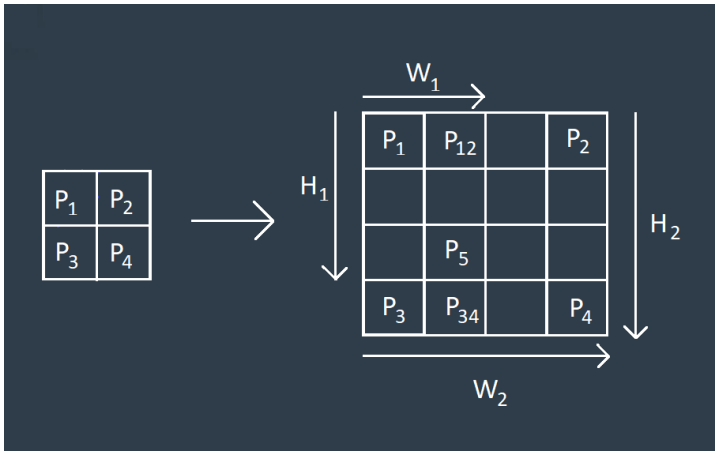
2) Code

```
output_layer = layers.Conv2D(filters=filters, kernel_size=kernel_size, strides=strides,
                             padding='same', activation='relu')(input_layer)
```

2th Bilinear Upsampling

1) Description

Bilinear upsampling is a resampling technique that utilizes the weighted average of four nearest known pixels, located diagonally to a given pixel, to estimate a new pixel intensity value. The weighted average is usually distance dependent.



$P_{12} = P_1 + W_1*(P_2 - P_1)/W_2$
$P_{34} = P_3 + W_1*(P_4 - P_3)/W_2$

2) Code

```
output = BilinearUpSampling2D((2,2))(input)
```

3rd Layer Concatenation

1) description

Concatenating two layers, the upsampled layer and a layer with more spatial information than the upsampled one, presents us with the same functionality. Implementing this functionality is quite straightforward, and gives us much spatial information in the next layer.

One added advantage of concatenating the layers is that it offers a bit of flexibility because the depth of the input layers need not match up unlike when you have to add them. Which helps simplify the implementation as well.

2) Code

```
output = layers.concatenate([input_layer_1, input_layer_2])
```

4th Full decoder function

1) Parts

I. A bilinear upsampling layer using the upsample_bilinear () function. The current recommended factor for upsampling is set to 2
II. A layer concatenation step. This step is similar to skip connections. You will concatenate the upsampled small_ip_layer and the large_ip_layer
III. separable convolution layers to extract some more spatial information from prior layers

2) Code

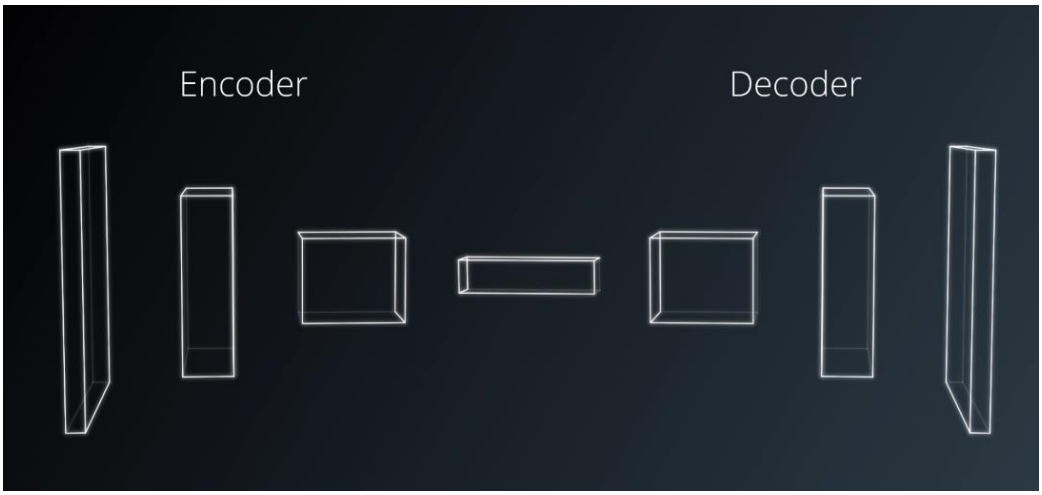
```
def decoder_block(small_ip_layer, large_ip_layer, filters):
    # TODO Upsample the small input layer using the bilinear_upsample() function.
    upsample=bilinear_upsample(small_ip_layer)
    # TODO Concatenate the upsampled and large input layers using layers.concatenate
    output = layers.concatenate([upsample, large_ip_layer])
```

```
# TODO Add some number of separable convolution layers
sep_conv1= separable_conv2d_batchnorm(output, filters)
sep_conv2 = separable_conv2d_batchnorm(sep_conv1, filters)
sep_conv3 = separable_conv2d_batchnorm(sep_conv2, filters)
return sep_conv3
```

BUILDNING TRAINING MODEL

1) Description

- Model consists of some encoder and decoder layers
- Encoder represents convolution layers
- Normally network consists of many convolution layers
- With each convolution layer width & height of layer decrease and depth increase
- After encoder layers we use 1x1 convolution that helps us to feed the model with images with different shapes because it only changes depth of the model
- Decoder layers is used to make fully convolutional network
- Specifically decoders is used to reverse encoder function to get images with spatial information



2) Code

```
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.


    layer1=encoder_block(inputs, 16, 2)
    layer2=encoder_block(layer1, 32, 2)
    layer3=encoder_block(layer2, 64, 2)
    layer4=conv2d_batchnorm(layer3, 128, kernel_size=1, strides=1)
    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    layer5=decoder_block(layer4, layer2, 64)
    layer6=decoder_block(layer5, layer1, 32)
    layer7=decoder_block(layer6, inputs, 16)
    # 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')(layer7)
```

TRAINING MODEL PARAMETERS

Before training model there we need to tweak some parameters to suit our project.

1) Batch size

Mini-batching is a technique for training on subsets of the dataset instead of all the data at one time. This provides the ability to train a model, even if a computer lacks the memory to store the entire dataset.

Mini-batching is computationally inefficient, since you can't calculate the loss simultaneously across all samples. However, this is a small price to pay in order to be able to run the model at all.

So, we need to choose suitable batch size for our model.

In this code chosen batch_size is 200.

2) Number of epochs

An epoch is a single forward and backward pass of the whole dataset. This is used to increase the accuracy of the model without requiring more data. This section will cover epochs in TensorFlow and how to choose the right number of epochs.

In this project **num_epochs = 10.**

3) Steps per epoch

Number of batches of training images that go through the network in 1 epoch. We have provided you with a default value. One recommended value to try would be based on the total number of images in training dataset divided by the batch_size.

In this project **steps_per_epoch = 400.**

4) Validation Steps

Number of batches of validation images that go through the network in 1 epoch. This is similar to steps_per_epoch, except validation_steps is for the validation dataset. We have provided you with a default value for this as well.

In this project **validation_steps = 50.**

5) Workers

Maximum number of processes to spin up. This can affect your training speed and is dependent on your hardware. We have provided a recommended value to work with.

In this project **workers = 2.**

TRAINING MODEL

1) GPU OPTIONS

To train model in this course there was 3 GPU options

1. Udacity GPU work space
2. Amazon AWS GPU instance
3. PC CPU training

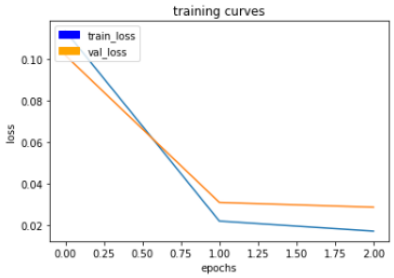
2) Data

Data is divided into:

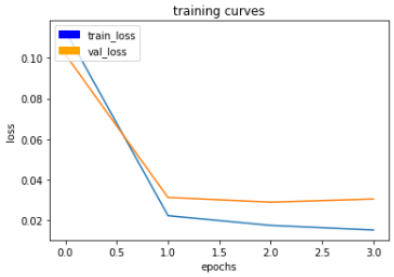
1. Training data
2. Validation data
3. Sample evaluation data

3) Results

```
400/400 [==============================] - 1183s - loss: 0.0222 - val_loss: 0.0311
Epoch 3/10
399/400 [==============================>.] - ETA: 2s - loss: 0.0173
```

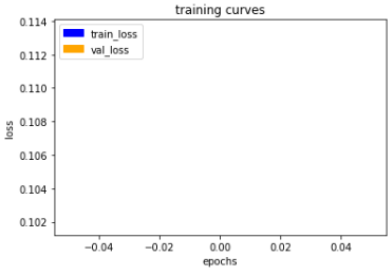



```
400/400 [==============================] - 1181s - loss: 0.0173 - val_loss: 0.0288
Epoch 4/10
399/400 [==============================>.] - ETA: 2s - loss: 0.0151
```

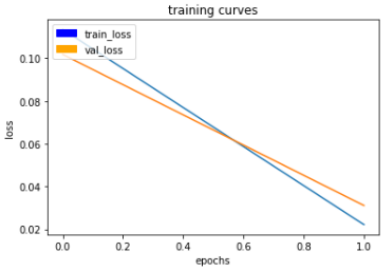



```
400/400 [==============================] - 1182s - loss: 0.0151 - val_loss: 0.0303
Epoch 5/10
399/400 [==============================>.] - ETA: 2s - loss: 0.0141
```

```
Epoch 1/10
399/400 [==============================>.] - ETA: 2s - loss: 0.1133
```

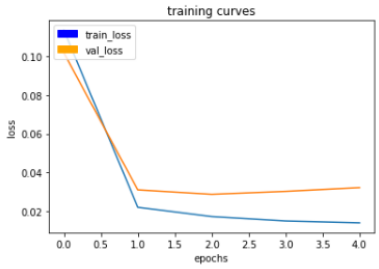



```
400/400 [==============================] - 1184s - loss: 0.1130 - val_loss: 0.1018
Epoch 2/10
399/400 [==============================>.] - ETA: 2s - loss: 0.0222
```

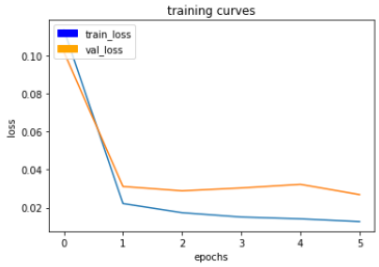



```
400/400 [==============================] - 1183s - loss: 0.0222 - val_loss: 0.0311
Epoch 3/10
399/400 [==============================>.] - ETA: 2s - loss: 0.0173
```

```
Epoch 5/10
399/400 [============================>.] - ETA: 2s - loss: 0.0141
```

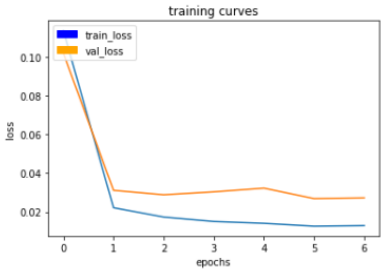



```
400/400 [==============================] - 1184s - loss: 0.0141 - val_loss: 0.0322
Epoch 6/10
399/400 [============================>.] - ETA: 2s - loss: 0.0127
```

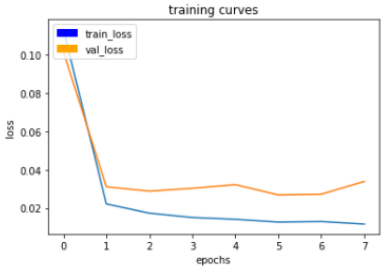



```
400/400 [==============================] - 1192s - loss: 0.0127 - val_loss: 0.0268
Epoch 7/10
399/400 [============================>.] - ETA: 2s - loss: 0.0130
```

```
Epoch 7/10
399/400 [============================>.] - ETA: 2s - loss: 0.0130
```

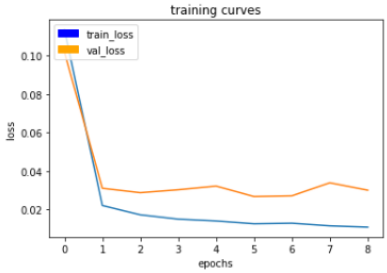



```
400/400 [==============================] - 1185s - loss: 0.0129 - val_loss: 0.0272
Epoch 8/10
399/400 [============================>.] - ETA: 2s - loss: 0.0116
```

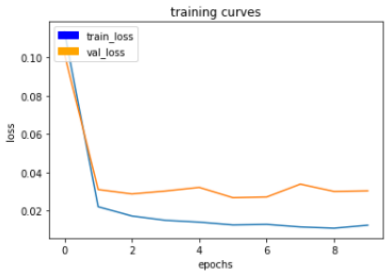



```
400/400 [==============================] - 1182s - loss: 0.0116 - val_loss: 0.0339
Epoch 9/10
399/400 [============================>.] - ETA: 2s - loss: 0.0109
```

```
400/400 [==============================] - 1182s - loss: 0.0116 - val_loss: 0.0339
Epoch 9/10
399/400 [==============================>.] - ETA: 2s - loss: 0.0109
```




```
400/400 [==============================] - 1184s - loss: 0.0109 - val_loss: 0.0301
Epoch 10/10
399/400 [==============================>.] - ETA: 2s - loss: 0.0125
```




```
400/400 [==============================] - 1183s - loss: 0.0125 - val_loss: 0.0304
```

PREDICTION

Now that we have your model trained and saved, you can make predictions on your validation dataset. These predictions can be compared to the mask images, which are the ground truth labels, to evaluate how well your model is doing under different conditions.

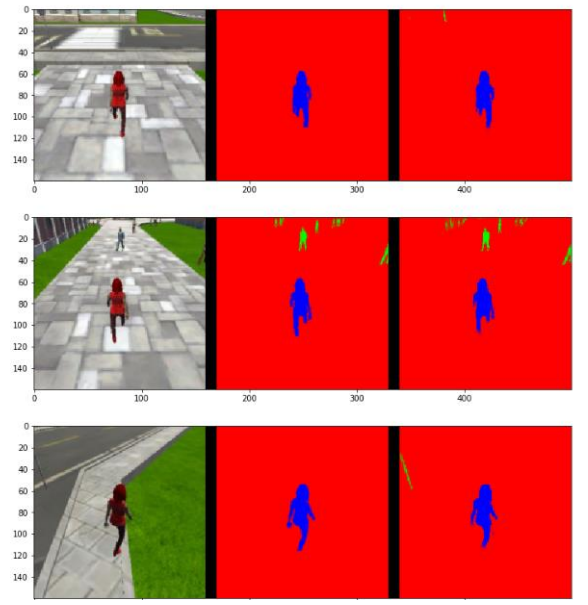
There are three different predictions available from the helper code provided:

a. **patrol_with_targ:** Test how well the network can detect the hero from a distance.
b. **patrol_non_targ:** Test how often the network makes a mistake and identifies the wrong person as the target.
c. **following_images:** Test how well the network can identify the target while following them.

Results

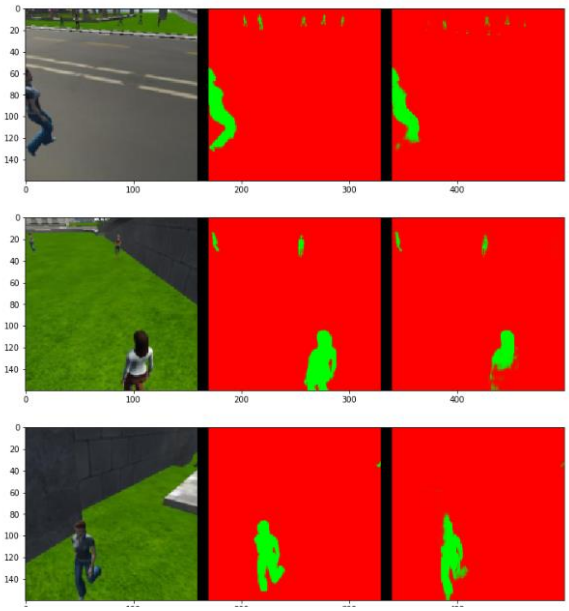
1) Image while following target

```
# images while following the target
im_files = plotting_tools.get_im_file_sample('sample_evaluation_data','following_images', run_
for i in range(3):
    im_tuple = plotting_tools.load_images(im_files[i])
    plotting_tools.show_images(im_tuple)
```



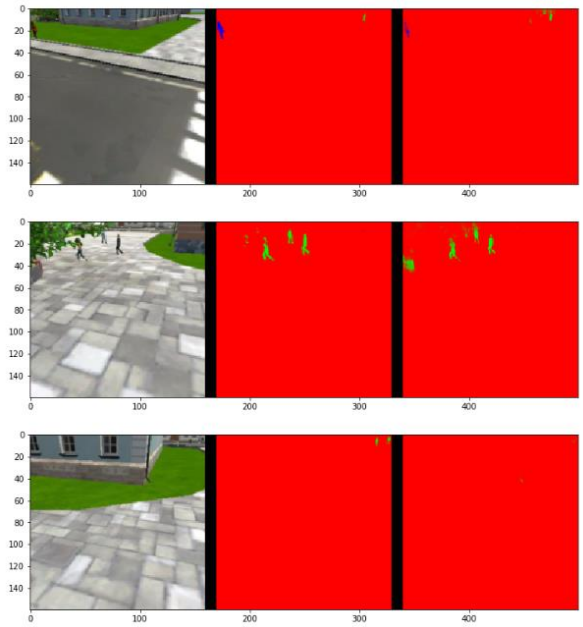
2) image while at patrol without target

```
# images while at patrol without target
im_files = plotting_tools.get_im_file_sample('sample_evaluation_data','patrol_non_targ', run_
for i in range(3):
    im_tuple = plotting_tools.load_images(im_files[i])
    plotting_tools.show_images(im_tuple)
```



3) Image while at patrol with target

```
# images while at patrol with target
im_files = plotting_tools.get_im_file_sample('sample_evaluation_data','patrol_with_targ', run_
for i in range(3):
    im_tuple = plotting_tools.load_images(im_files[i])
    plotting_tools.show_images(im_tuple)
```



EVALUATION

In [26]:
```
# Scores for while the quad is following behind the target.
true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_following, pred_following)
```

```
number of validation samples intersection over the union evaulated on 542
average intersection over union for background is 0.995957911228407
average intersection over union for other people is 0.3585011210517855
average intersection over union for the hero is 0.9057906380121693
number true positives: 539, number false positives: 0, number false negatives: 0
```

In [27]:
```
# Scores for images while the quad is on patrol and the target is not visable
true_pos2, false_pos2, false_neg2, iou2 = scoring_utils.score_run_iou(val_no_targ, pred_no_targ)
```

```
number of validation samples intersection over the union evaulated on 270
average intersection over union for background is 0.9884735577236363
average intersection over union for other people is 0.7609383519973582
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 36, number false negatives: 0
```

In [28]:
```
# This score measures how well the neural network can detect the target from far away
true_pos3, false_pos3, false_neg3, iou3 = scoring_utils.score_run_iou(val_with_targ, pred_with_targ)
```

```
number of validation samples intersection over the union evaulated on 322
average intersection over union for background is 0.9966641110343256
average intersection over union for other people is 0.4415596235627761
average intersection over union for the hero is 0.223550821731076
number true positives: 119, number false positives: 0, number false negatives: 182
```

In [29]:
```
# Sum all the true positives, etc from the three datasets to get a weight for the score
true_pos = true_pos1 + true_pos2 + true_pos3
false_pos = false_pos1 + false_pos2 + false_pos3
false_neg = false_neg1 + false_neg2 + false_neg3

weight = true_pos/(true_pos+false_neg+false_pos)
print(weight)
```

```
0.7511415525114156
```

In [30]:
```
# The IoU for the dataset that never includes the hero is excluded from grading
final_IoU = (iou1 + iou3)/2
print(final_IoU)
```

```
0.564670729872
```

In [31]:
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.424147648694
```

CODE OPTIMIZATION & FUTURE WORK

1) COLLETING MORE DATA

- To increase overall efficiency it is better to use much more data samples
- Additional Data samples can be collected from simulator

INCREASING SAMPLE QUALITY

- Increasing sample quality can help in increasing model performance
- The only disadvantage of this is the increase in processing power needed to train the model

INCREASING EPOCHS & DECREASING LEARNING RATE

- Increasing epochs can help increasing efficiency only if the model loss value is still decreasing
- To obtain the decrease in loss we need to decrease learning rate

INCREASING CONVOLUTIONAL LAYERS

- Additional way to increase model score is to increase convolutional layers which increases filters as well
- This will increase robot ability to recognize more complicated shapes thus increasing overall efficiency

FUTURE WORK

- I think it will be a good practice and will add me a lot of experience if I implemented this project with tensorFlow GPU on my notebook.

- I already tried to implement it and installed TensorFlow GPU and all necessary libraries for it and it worked, but I found a lot of error that I think it is related to jupyter and some differences bet CPU & GPU tensor libraries.

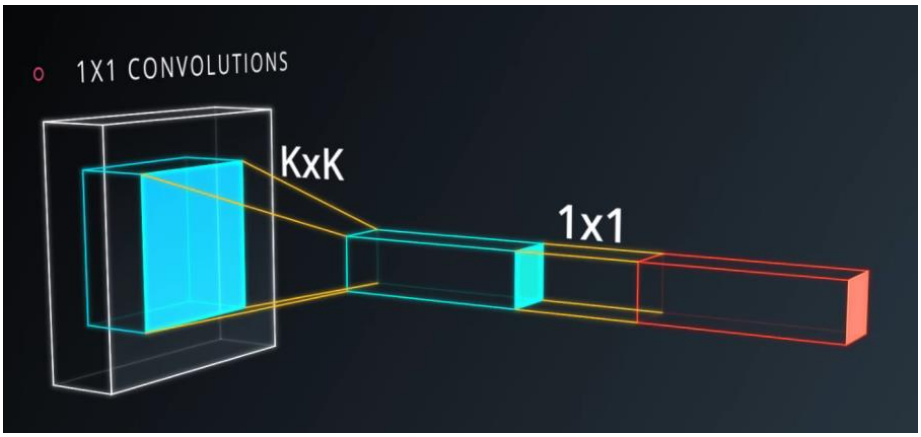
MODIFIED PARTS

1X1 CONVOLUTION DEMONESTRATION

In TensorFlow, the output shape of a convolutional layer is a 4D tensor. However, when we wish to feed the output of a convolutional layer into a fully connected layer, we flatten it into a 2D tensor. This results in the loss of spatial information, because no information about the location of the pixels is preserved.

But when we add 1x1 convolution we get layers with the 4D dimension, so, spatial information is preserved and we can feed this layer to output layers and get the spatial information.

It helps making model nonlinear, deeper, has more parameters.



FULLY CONNECTED LAYERS

- It is the layer that follows convolutional layers.
- After feature extraction we need to classify the data into various classes, this can be done using a fully connected (FC) neural network
- Doesn't has spatial information.
- Can be used to identify whether the object is in the whole image or not.

CAN WE USE THIS MODEL AND THIS DATA TO IDENTIFY ANOTHER OBJECT !?

Surely no.

The reason for that is:

- This data has labels only indicate weather hero is in the image or not.
- So if we want to train our model to detect another object we will need whole new data set for the new object