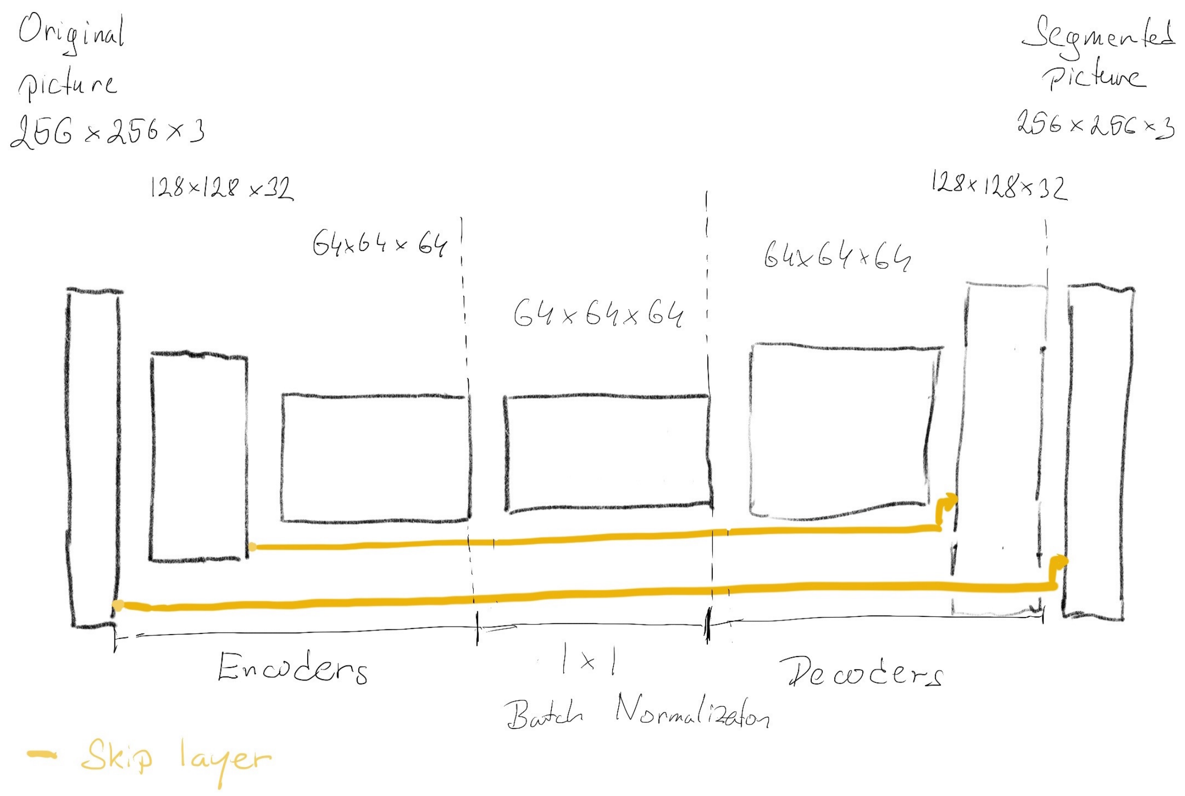
Project: Deep learning

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

The write-up conveys the an understanding of the network architecture.

The student clearly explains each layer of the network architecture and the role that it plays in the overall network. The student can demonstrate the benefits and/or drawbacks of different network architectures pertaining to this project and can justify the current network with factual data. Any choice of configurable parameters should also be explained in the network architecture.



After some experimentation I ended up with "5" layer network. The network has the following layers.

The implementation is as floowing. The individual layers are explained below

def fcn_model(inputs, num_classes):

```
# Add Encoder Blocks.
# Remember that with each encoder layer, the depth of your model (the number of
filters) increases.
# 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 too much of a cost
l1 = encoder_block(inputs, 32, 2)
l2 = encoder_block(l1, 64, 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
l3 = conv2d_batchnorm(l2, 64, 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.
l4 = decoder_block(l3, l1, 64)
l5 = decoder_block(l4, inputs, 32)

x = l5
# 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')(x)
```

2 layers of encoders

We have 2 layers of encoders. Each layer of encoders is designed to "compress" the previous layer to produce some features of the picture. The compression is achieved by convolutional layer with stride > 1.

The advantage of convolutional layers as opposed to dense layers is that the processing happens in predefined surrounding of neuron in question. This setup preserves spatial information so later we are able to tell where we detected the desired feature.

As a specialty this is using separable convolutions. That is we apply convolution only per layer and on the output apply 1x1 convolution to achieve desired depth. This results in a tensor of equal dimensions we would have gained otherwise but with much less parameters and computational resources required.

We used same padding in all cases simplifying the thinking about dimensions of the layers.

The encoder is implemented like this

```
def encoder_block(input_layer, filters, strides):
    output_layer = separable_conv2d_batchnorm(input_layer, filters, strides)
    return output_layer
```

1x1

The middle layer contains 1x1 and batch normalization. The batch normalization facilitates better learning. Also introduces some randomnes serving as regularization.

Implementation

```
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
```

2 layers of decoders

These serve as a counterpart to the convolution layer and performs deconvolution. They contain 3 individual steps.

- upsampling. This is to revert the stride of convolution which caused the picture to shrink now we are inflating it.
- we are adding the information from the skip connections. These serve as giving the network more context. It is achieved by piecwise concatenation so we need to make sure we are adding a layer of appropriate size.
- We are adding a layer of separable convolutions to counteract the convolution in encoders and batch normalization. Here if I understand it correctly the 1x1 convolution is used mainly so we remove some dimensions in the filter space and match the layer to for the next step.

Implementation looks like this

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

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

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

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

    return output_layer
```

Skip connections

We are using skip connections between encoders and decoders. Since the skip layer is implemented as element wise addition we need to make sure these are applied on the layers of same size. These layers help the network to keep more context of the original image a nd provides some form of regularization and defense against overfitting.

The write-up conveys the student's understanding of the parameters chosen for the the neural network.

The student explains their neural network parameters including the values selected and how these values were obtained (i.e. how was hyper tuning performed? Brute force, etc.) Hyper parameters include, but are not limited to:

Batch size

How many pictures are sent into the network in one step. The batch size si set so we can limit the amount of memory consumed.

Steps per epoch/validation steps

Step is a full forwards and backward iteration on one batch

Epochs

The epochs is a full forward an backward passes on all imags (in our cases it is steps per epoch times. If we see all images depends on the batch size).

Learning rate

This parameter tunes the proportion by which is weighted the change of weights during the back propagaiion step (the learning phase).

Filter depth of the layer

Marks number of feature layers generated during encoding phase. This was chosen arbitrarily. I started with 32 -> 16 -> 16 -> 32 but then tried to double that after consultation with the Slack channel. Seems to have improved the performance without huge impact on performance.

how we picked the parameters

The parameters were tuned by hand. Not randomly or brute force but by combination of these approaches. I picked initial value, ran and then tried to guess what change might influence the result in positive manner. Rinse repeat.

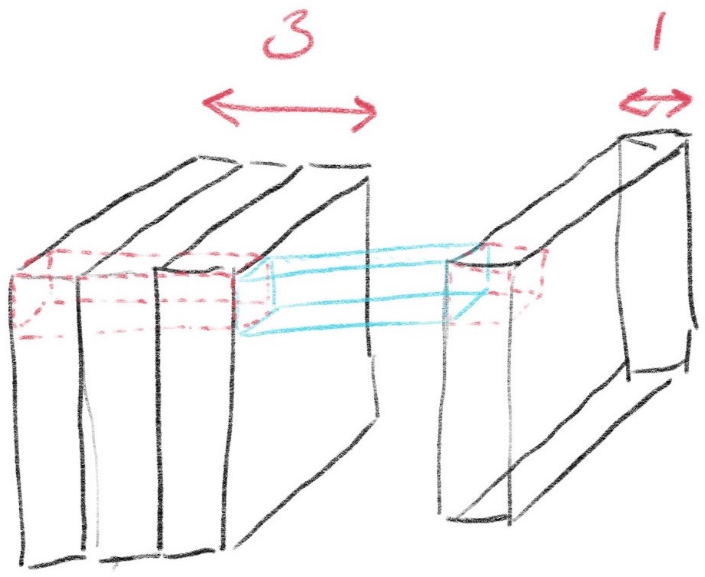
- Epoch is something I did increase once I felt the network is doing something since it linearly increses the training time. The value eventually landed at 80. Seems the network is still learning after that albeit slowly.
- The batch size is something I did not tune very much in my opinion I was not starved of - resources and I wanted make sure the network sees all the pictures.
- Learning rate was a bit difficult to tune. The main reason is that it is not completely clear how the number corresponds to anything tangible. As per recommendations I kept calm and decreased the value continuously. Ended up with 0.001. Also I took some guidance from the Slack.

If i had more time and was doing this more often I would probably try to automate this. I had the training going for a week on the background anyway. The biggest problem is it takes ages to complete.

I would try to do somethings like this. Pick some values in a grid. Perform the training evaluate the metrics and present them. This would serve as a starting point for some further tuning by hand.

The student has a clear understanding and is able to identify the use of various techniques and concepts in network layers indicated by the write-up.

The student demonstrates a clear understanding of 1 by 1 convolutions and where/when/how it should be used.

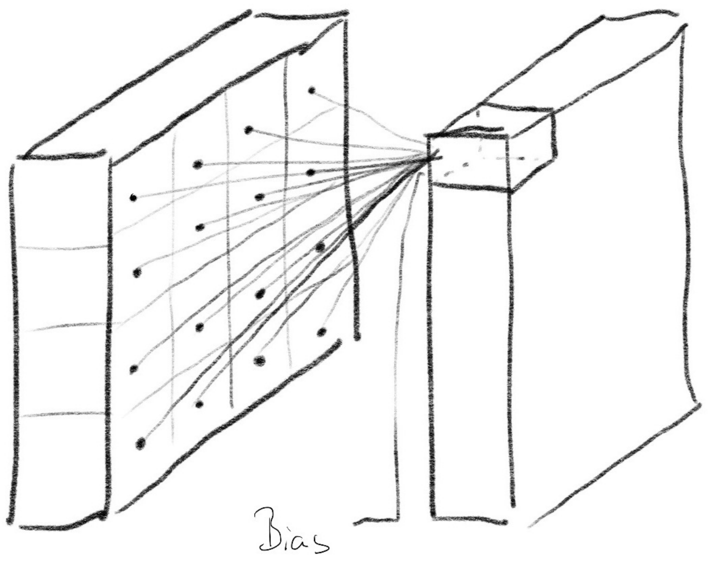


1x1 convolutions is convolutional layer as any other with couple of differences

- The kernel is 1x1 as the name suggests which initially is counterintuitive
- Stride is 1 so the output is size of the input
- The computation is potentially faster since it can leverage the fact it can be implemented as a matrix multiplication in this special case
- It leads to a reduction/expansion in the filter dimension. Since it is 1x1 kernel width and height is preserved but we can change the third dimension. This is done while we are losing no spatial information (w, h retained) and in case of reduction we are also not losing all information in the filter space since we are not just discarding the layers but convolving on them.

The usage seem to be three fold - Reducing dimensionality before expensive convolutions thus making them significantly cheaper to compute - Seems that the way it is used is often a nonlinear activation at the end so they can act as additional nonlinear piece in the network - cheap way to make the network deeper - enables implementation of separable convolutions

The student demonstrates a clear understanding of a fully connected layer and where/when/how it should be used.



Fully connected layer is the most general way how to connect layer in the neural network. It means that every node in one layer is connected to all nodes in the other layer.

For example in the network 4x4 connected to 2x2. it means there is 4x4x2x2=64 connections (not counting the bias). This grows very quickly. for network 32x32 connected to 32x32 this is 1048576 connections.

Let's assume that the input layer has dimensions `m x n`. and the output layer `o x p`. The number of connections is

```
number per one output node  = m x n + 1
```

The one os for bias. Multiplied by number of neurons in the output layer resulting in.

```
(m x n + 1) x o x p
```

There are two big drawbacks to this type of layer.

- as suggested above there is a lot of weights to be stored and train so this is expensive both spatially and computationally
- since any output cell can be activated from any place in the network this might not be that useful in image detection or other aplication where there is some concept of space. I believe this is referred in the lectures as "preserves spatiality" in case of convolutional layers.

The usage according to the lectures is during classification and it is not. As said previously since fully connected layer is the most general one it should not be impossible to turn a fully connected layer into a convolutional layer but I think it is one of the ways how to improve the performance for certain tasks to constrain the network in some way (providing convolutional layers outright) that might lead the network cope with the task much quicker.

The student has a clear understanding of image manipulation in the context of the project indicated by the write-up.

The student is able to identify the use of various reasons for encoding / decoding images, when it should be used, why it is useful, and any problems that may arise.

Encoding is commonly done by convolutional layers. Convolution in images is basically a designed to apply a filter trying to find some features in an image. In classical image processing this is done by specific kernels like Canny edge detector or Sobel detector and many others.

Several layers of encoding is trying to emulate the natural process of finding some simple features in a picture lines, edges then using those in combination to learn about more complex features. As an example might be edges into corners into more complex shapes like faces etc.

On the other side is deconvolution. This is from these basic features somehow reconstruct the original image. This is to gain the same size as the original image.

The trick is that we are not telling the network what kernels it should use we are not burdened with feature engineering. It finds features itself during the backpropagation process thus we are not forced to do our feature generation. This is where the wide applicability and recent huge jumps in performance come from.

Typical problems that arise during designing a network is the fact that convolution is a reduction, smoothing in a sense. It always works on some area and the result is smaller. We have to deal with this somehow usually by some kind of padding or by acknowledging the result is smaller.

Second problem is computational intensity of convolution. since we have tend to generate a lot of feature layers the multidimensional convolutions can be quite intensive. One of the techniques used to mitigate this is depth separabel convolutions. These produce same shape tensor with less burden on resources. Nice example is worked out here

One of the additional techniques is skip layer connections. These are designed to connect layers from encoder to teh decoder. The idea is that it gives the layer more context of the original picture along with the generated features. Probably serves as a way of regularization also.

The student displays a solid understanding of the limitations to the neural network with the given data chosen for various follow-me scenarios which are conveyed in the write-up.

The student is able to clearly articulate whether this model and data would work well for following another object (dog, cat, car, etc.) instead of a human and if not, what changes

would be required.

It is an interesting thought but my guess is it would not work well. The prime reason for this is that the network was not fed a single image of dog or cat during training phase so I would not expect it to generalize like that on new data. The features where most likely specifically tuned to recognize people.

What we would have to change is to simply feed it correctly labeled images of desired object to be detected in the picture so the network can tune itself properly and repeat the training phase. I do not think that the general shape of the network (FCN, depths etc) has to be changed since that seems to be working fine and type of task is still the same.

Discussion

To be honest I am very impressed with this exercise. I was studying some neural networks and some image manipulation during college 10 years ago. At that time neural networks were taught as a curiosity. This is first time I had a chance to actually train a deep neural netowrk. The performance on such a complex task that can be achieved by very simple tools on an average computer today is astounding.

The training itself took several days of computation power. Tweaking trying etc. Initially it took me a while to produce a network and pick the parameters so that the network starts doing something. Until that point (I think the main role played epochs=20 at which the training took several hours) the network did nothing and I was not sure if I am going in the right direction.

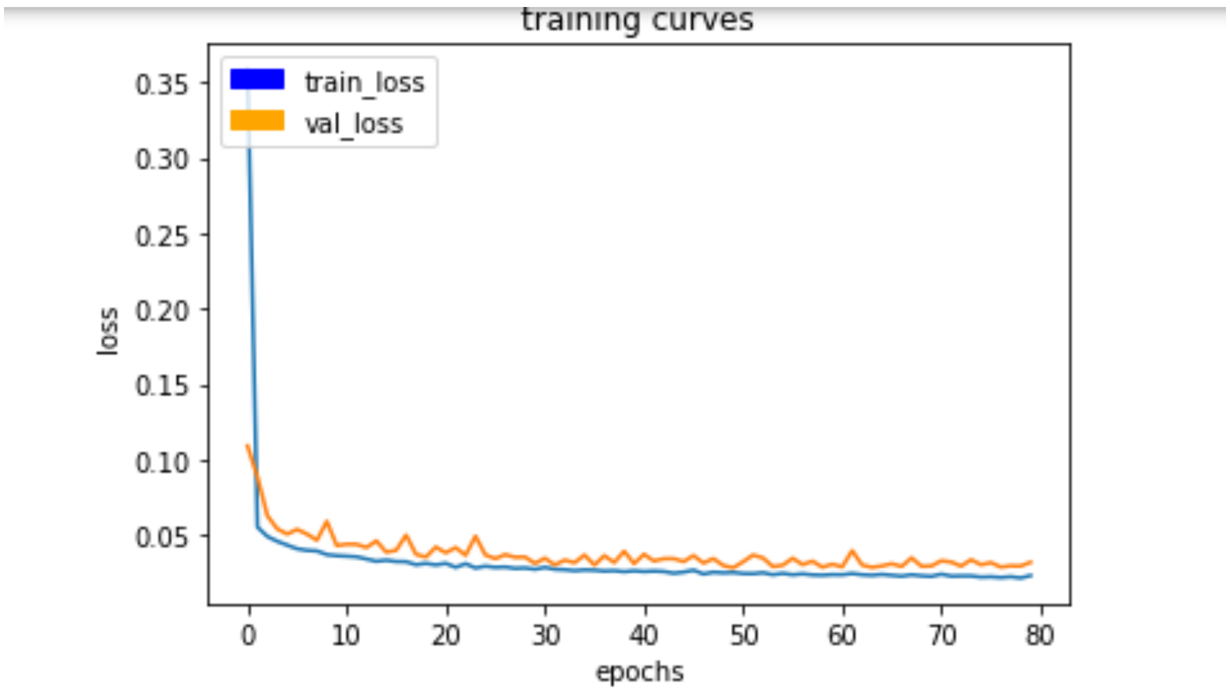
Unfortunately the lenght of the training (did not yet try the GPU powered training) does not lend to quick iteration exacerbated by the fact I have very little experience.

The training of the network was eventually performed with the following parameters. The meaning and how we achieved those parameters is described above.

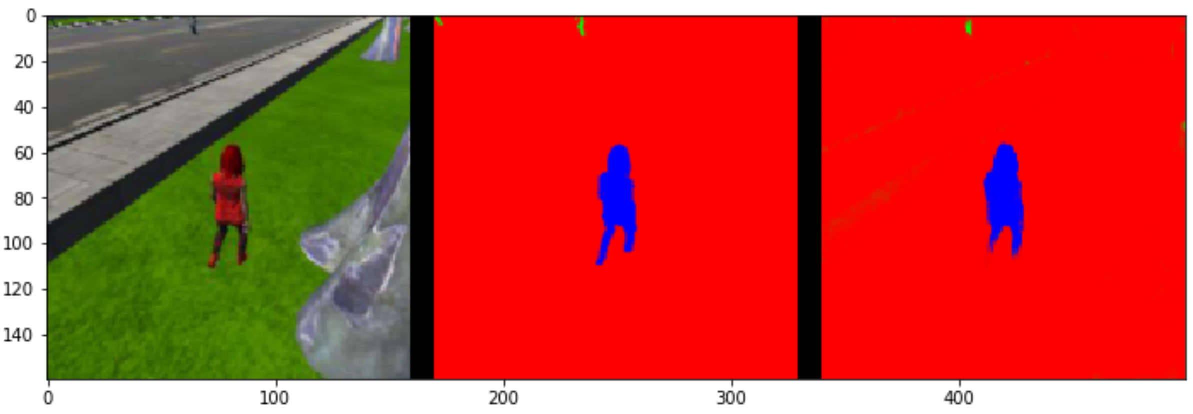
```
# Started with 0.1 and lowered gradually
learning_rate = 0.001
# Batch size is number of pictures per "iteration". Interplays with steps_per_epoch
# Ideally the network should see each picture during epoch
batch_size = 30
# picked 80 since it filled the night seems like the network is still learning
num_epochs = 80
steps_per_epoch = 200
validation_steps = 50
workers = 2
```

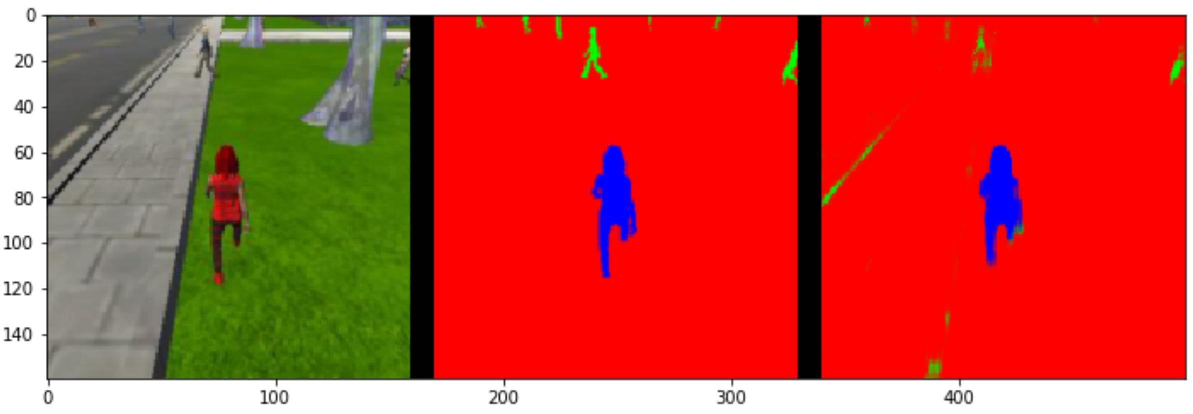
Major step in performance of network was the addition of feature layers in encoder layer. The training time increased almost unnoticably. Probably thanks to separable convolution implemenations of encoders.

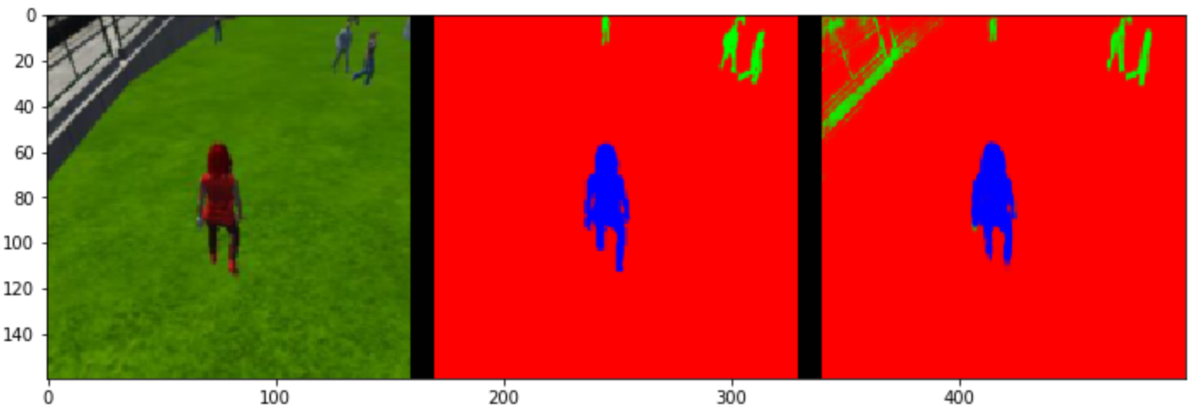
Here is a picture demonstrating the progress of learning. Seems that the learning of the process still continues after 80 iterations.

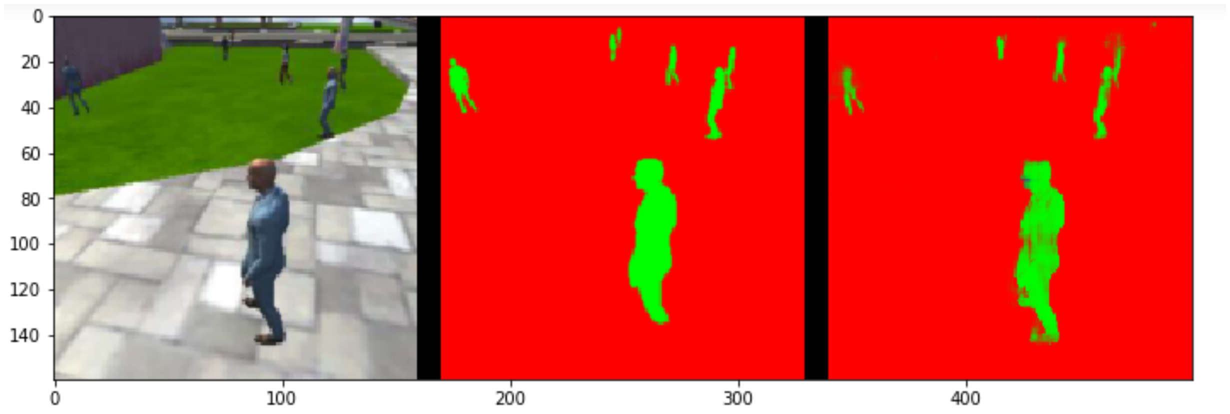


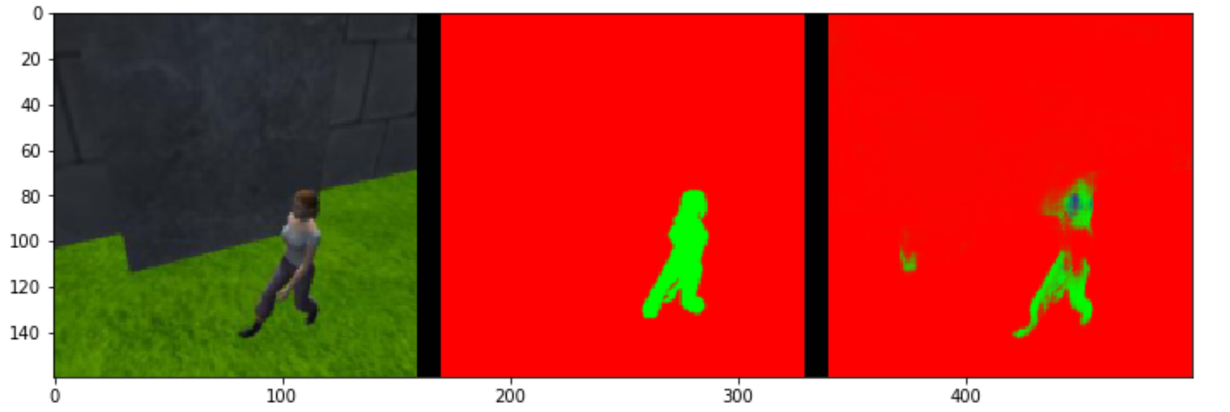
Here are couple of examples of the performance of the network. It is imprssive to see network correctly identify for example people in the distance etc.

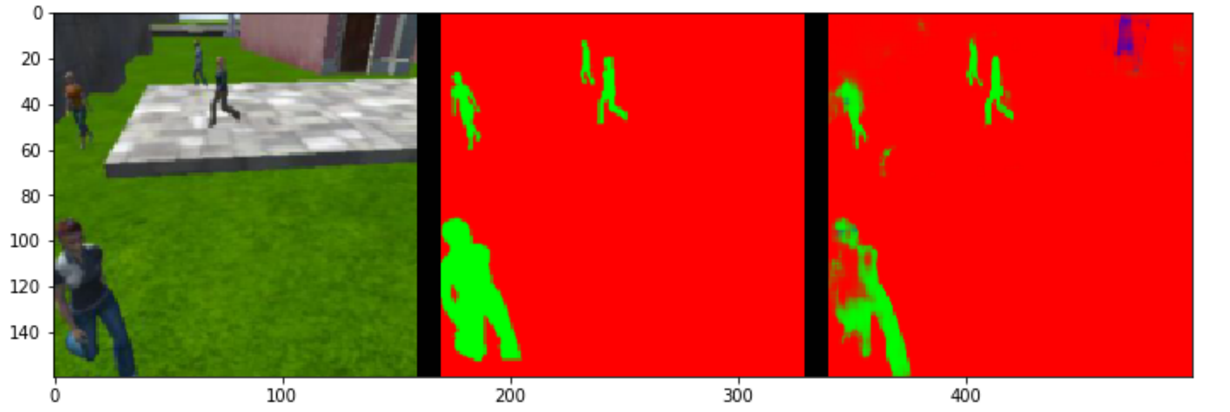

0
20
40
60
80
100
120
140
0
100
200
300
400

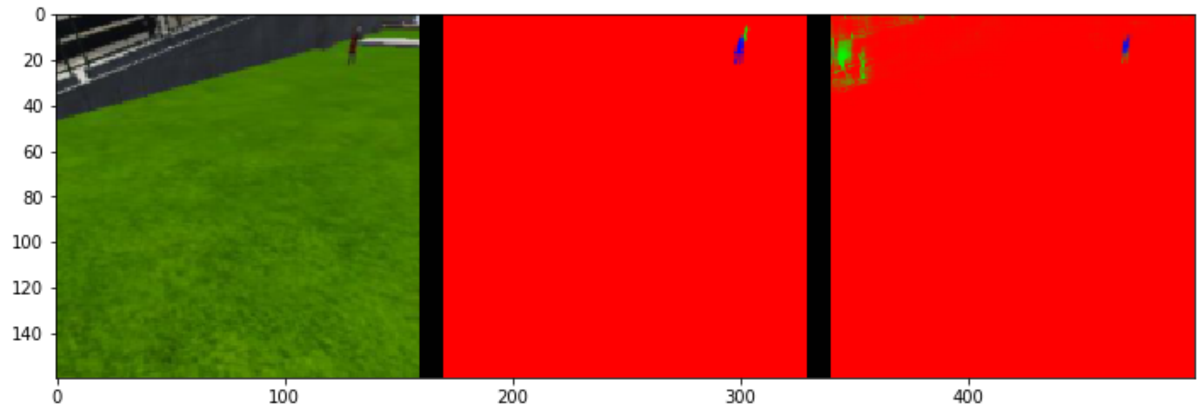

0
20
40
60
80
100
120
140
0
100
200
300
400

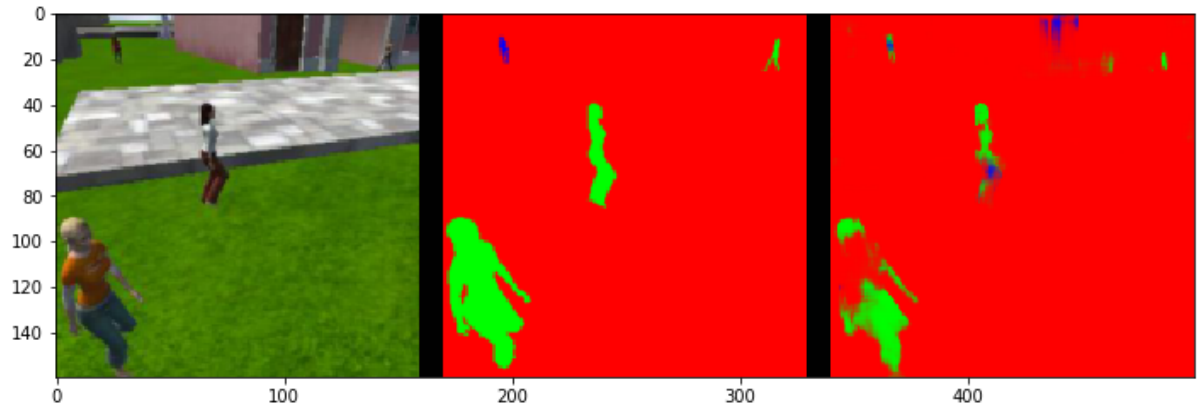

0
20
40
60
80
100
120
140
0
100
200
300
400

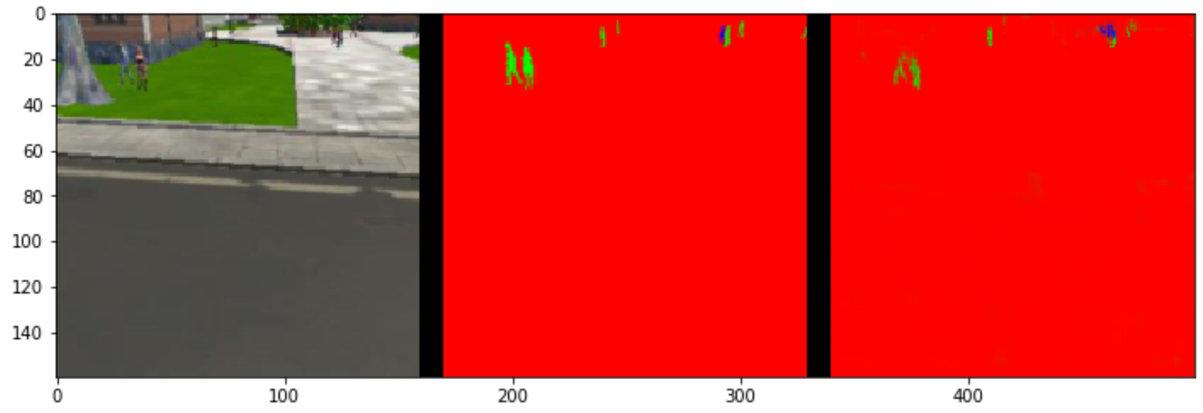

0
20
40
60
80
100
120
140
0
100
200
300
400


0
20
40
60
80
100
120
140
0
100
200
300
400


0
20
40
60
80
100
120
140
0
100
200
300
400


0
20
40
60
80
100
120
140
0
100
200
300
400


0
20
40
60
80
100
120
140
0
100
200
300
400


0
20
40
60
80
100
120
140
0
100
200
300
400