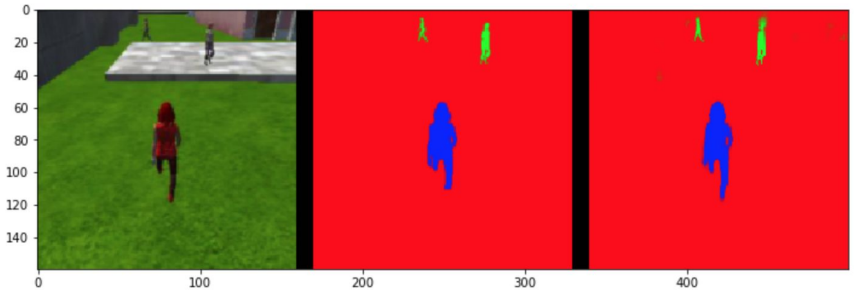
TABLE OF CONTENTS

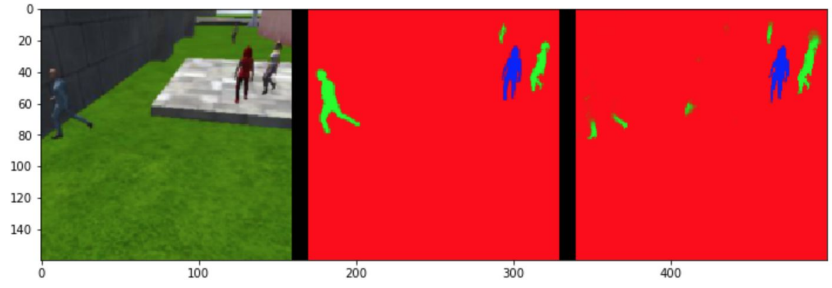
- Introduction
- Network Architecture
- Hyperparameters and Other Design Choices
- Experiments
- Discussion
- Future Enhancements

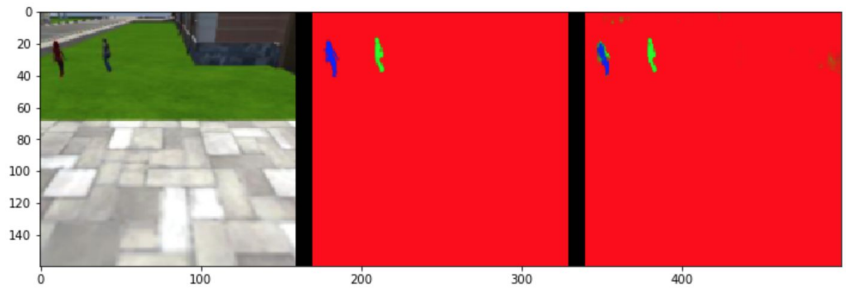
INTRODUCTION

In this project I used a fully-convolutional neural network to paint all pixels in an image which is part of a person. Two types of persons are identified, the "hero" target person, and everyone else. Identifying a specific person from everyone else are useful for "follow-me" operations of a drone. The training images are from a default data set (old and new) given by Udacity which are taken from images of cameras captured by a drone in a computer simulator. To measure how well the model is performing, the IOU (intersection over union) metric is used which takes the intersection of the prediction pixels and ground truth pixels and divides it by the union of them. Only the images with the people are evaluated with this metric. I used the AWS Udacity Robotics Laboratory Community AMI to train my model.

Note that this this model will not work well to follow another object (e.g dog, cat, car). This is because we only trained using classified images of people and in particular identifying our target person wearing a red outfit. To follow another object (such as a dog, cat and car) we should feed the architecture with classified images of that object (dog, cat, or car) which we didn't do. We have only fed the model with classified images of humans from the simulator. Because of this, it learned to distinguish between the target human with red clothes and the other people in the simulator.



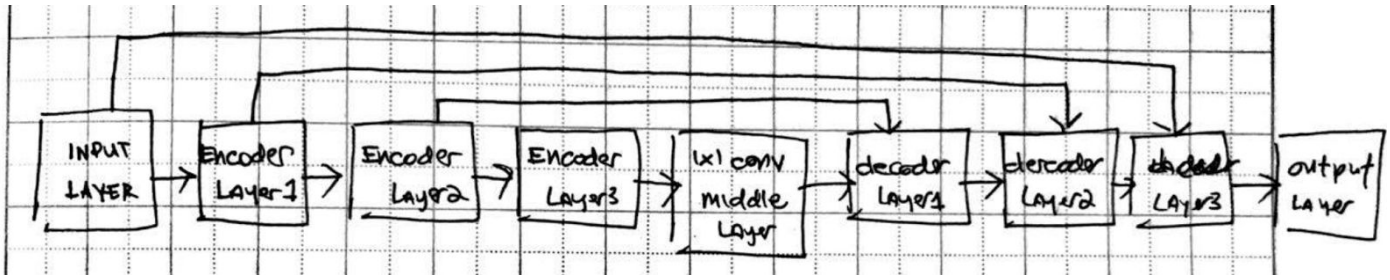




NETWORK ARCHITECTURE

I used a encoder-decoder architecture of a Fully-Convolutional Neural Network. This is popular because it is said to be faster than previous approaches. The encoder part reduces the spatial dimension and then the decoder part gradually recovers the spatial dimension. We encode images for the network to essentially learn details about the image for classification. This encoder parts have pooling which down-samples the image. This is used so that the model will generalize to images it hasn't seen before and reduce the risk of overfitting. The downside of this is that it loses information and reduces the spatial dimension. The role of the decoder is to recover this spatial dimension. The decoder part maps the low resolution encoder feature maps to full input resolution feature maps for pixel-wise classification.

I used three layers for each encoder and decoder part of the architecture, with the intent of the adding more layers in the case the target score is not obtained. It seems that three layers are sufficient to receive the target score but for a better score, we can consider adding more layers.




```
def fcn_model(inputs, num_classes):

    f = 32
    encoder_layer1 = encoder_block(inputs, f, 2)
    encoder_layer2 = encoder_block(encoder_layer1, 2*f, 2)
    encoder_layer3 = encoder_block(encoder_layer2, 4*f, 2)
    mid_layer = conv2d_batchnorm(encoder_layer3, 4*f ,kernel_size=1,strides=1)
    decoder_layer1 = decoder_block(mid_layer, encoder_layer2, 4*f)
    decoder_layer2 = decoder_block(decoder_layer1, encoder_layer1, 2*f)
    decoder_layer3 = decoder_block(decoder_layer2, inputs, f)
    outputs = layers.Conv2D(num_classes, 1, activation='softmax', padding='same')(decoder_layer3)

    return outputs
```

```
Tensor("input_1:0", shape=(?, 160, 160, 3), dtype=float32)
Tensor("batch_normalization/batchnorm/add_1:0", shape=(?, 80, 80, 32), dtype=float32)
Tensor("batch_normalization_2/batchnorm/add_1:0", shape=(?, 40, 40, 64), dtype=float32)
Tensor("batch_normalization_3/batchnorm/add_1:0", shape=(?, 20, 20, 128), dtype=float32)
Tensor("batch_normalization_4/batchnorm/add_1:0", shape=(?, 20, 20, 128), dtype=float32)
Tensor("batch_normalization_6/batchnorm/add_1:0", shape=(?, 40, 40, 128), dtype=float32)
Tensor("batch_normalization_8/batchnorm/add_1:0", shape=(?, 80, 80, 64), dtype=float32)
Tensor("batch_normalization_10/batchnorm/add_1:0", shape=(?, 160, 160, 32), dtype=float32)
Tensor("conv2d_2/truediv:0", shape=(?, 160, 160, 3), dtype=float32)
```

It has the following features:

- **Encoder part with pooling**
- **1x1 Convolutional**
- **Skip connections**
- **Bilinear Interpolation Upsampling with Transposed convolutional layers in the decoder part**

The encoder layers look like layers from a usual classification neural networks with a final fully-connected layer with pooling techniques to reduce the risk of overfitting. Pooling makes each layer lose information. It is found that adding connections from encoder layer to the decoder layer called skip connections to add more information makes the results less course. To connect the the encoder layers to the decoder, instead of a fully connected layer, we use a 1x1 convolutional layer to retain the spatial information. Upsampling in the decoder part used bilinear interpolation for better results as discussed in technical articles I've read.

I learned more about fully-convolutional networks and their features in the following sources:

- http://blog.qure.ai/notes/semantic-segmentation-deep-learning-review
- http://cv-tricks.com/image-segmentation/transpose-convolution-in-tensorflow/
- https://www.youtube.com/watch?v=ByjaPdWXKJ4&t=1027s
- https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf
- https://arxiv.org/pdf/1603.07285.pdf
- http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html#transposed-convolution-arithmetic
- https://github.com/vdumoulin/conv_arithmetic

HYPERPARAMETERS AND OTHER DESIGN CHOICES

Optimizer [Adam vs Nadam] and Learning Rates

At first, I used the Adam Optimizer for a few trials with a small learning rate as suggested in the lessons. A high learning rate might overshoot the desired output but a small learning rate might take too long to reach the destination. Also a small learning rate could get it stuck at a local minimum. I shifted to Nadam which is an Adam Optimizer with Nesterov Momentum which was suggested in a discussion on Slack. Momentum makes it converge more quickly and actually increased the score from about 31 to 41 with no other changes. I used the recommended learning rate of 0.002 for Nadam.

source:
https://www.quora.com/What-is-an-intuitive-explanation-of-momentum-in-training-neural-networks
https://keras.io/optimizers/#nadam

Batch Size

Computing the gradient over the entire dataset is expensive and slow. This is why we use batches. I chose a size of 40.

It is said that:

1. **This is usually 32- 512 data points.**
2. **In terms of computational power, while the single-sample stochastic GD process takes many many more iterations, you end up getting there for less cost than the full batch mode, "typically."**
3. **Optimizing the exact size of the mini-batch you should use is generally left to trial and error.**
4. **It has been observed in practice that when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize. The lack of generalization ability is due to the fact that large-batch methods tend to converge to sharp minimizers of the training function.**
5. **Batch size and learning rate are said to be linked. If the batch size is too small then the gradients will become more unstable and would need to reduce the learning rate.**

source: https://stats.stackexchange.com/questions/164876/tradeoff-batch-size-vs-number-of-iterations-to-train-a-neural-network/236393

Number of epochs, Steps per epoch, and Validation Steps per epoch

Number of Epoch

This is the total number of iterations on the same data. A large number of epoch might overfit your data while a small number of epoch might underfit your data.

Steps per epoch:

The number of steps (batches of samples) before declaring your epoch finished. This should be the number of training images over batch size because you should theoretically train your entire data on very epoch.

Validation Steps per epoch:

This should be number of validation images over batch size because you should test all your data on every epoch

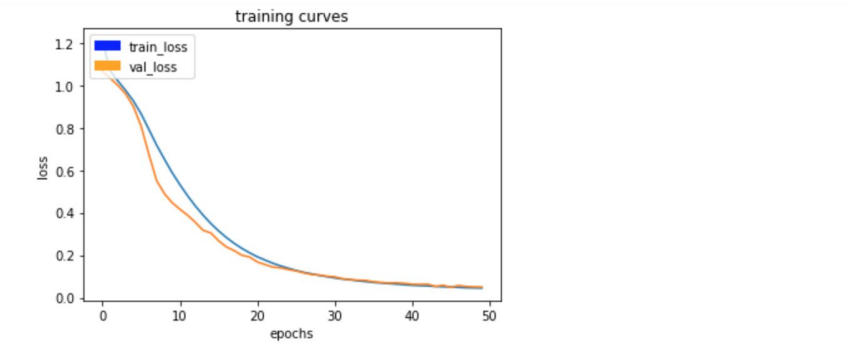
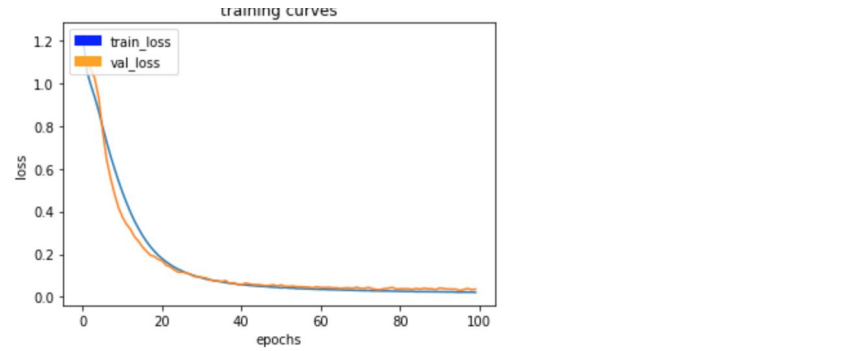
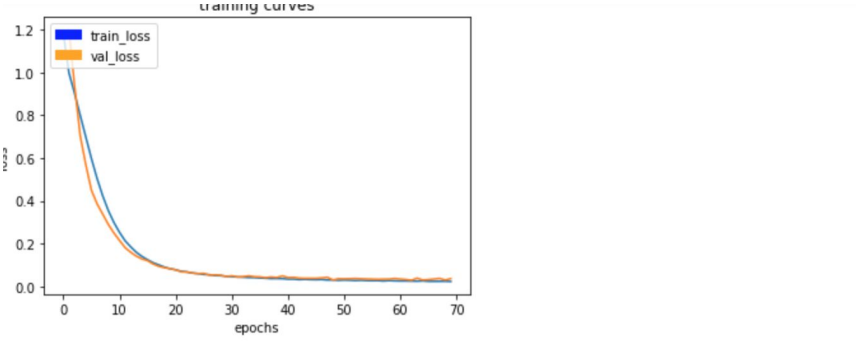
source:
https://keras.io/models/sequential/#fit_generator

https://stackoverflow.com/questions/45943675/meaning-of-validation-steps-in-keras-sequential-fit-generator-parameter-list

Workers

This is the number of parallel processes during training. This can affect your training speed and is dependent on your hardware. I tried a 2, 4, 10, but I didn't observe any change in training speed.

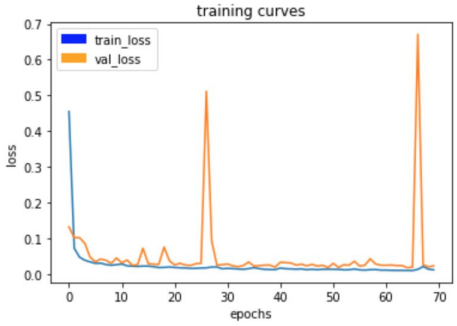
EXPERIMENTS

Trial 1 **Given all images in the given OLD default training data set (2701 images)** **Percent of training data containing a hero: 23.95%** **Optimizer: Adam Optimizer** **learning_rate = 0.0001** **num_epochs = 50** **batch_size = 40** **Final score: Forgot to save**	 68/68 [==============================] - 51s - loss: 0.0450 - val_loss: 0.0501
Trial 2 **Given all images in the given OLD default training data set (2701 images)** **Percent of training data containing a hero: 23.95%** **Optimizer: Adam Optimizer** **learning_rate = 0.0001** **num_epochs = 100** **batch_size = 40** **Final score: 0.328208590245**	 68/68 [==============================] - 51s - loss: 0.0215 - val_loss: 0.0362
Trial 3 **number_of_training_images = 4600** **(Removed 592 background images of the OLD dataset then flipped the rest)** **Optimizer: AdamOptimizer** **learning_rate = 0.0001** **num_epochs = 70** **batch_size = 40** **Percent of training data containing a hero: 27.60%** **Final score: 0.319695876581**	 16/116 [==============================] - 84s - loss: 0.0227 - val_loss: 0.0359

Trial 4

number_of_training_images = 4600
(Removed 592 background images then flipped the rest, old data set)

Optimizer: NadamOptimizer
learning_rate = 0.002
num_epochs = 70
batch_size = 40
Percent of training data containing a hero: 27.60%
Final score: 0.4249

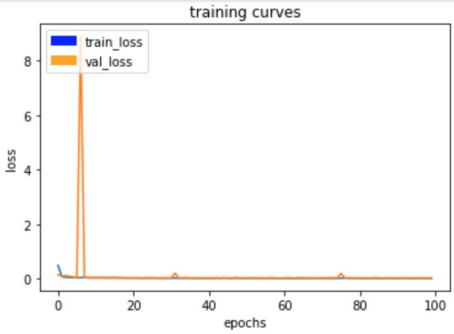



```
116/116 [==============================] - 84s - loss: 0.0116 - val_loss: 0.0228
```

Trial 5

number_of_training_images = 4600
(Removed 592 background images then flipped the rest, old data set)

Optimizer: NadamOptimizer
learning_rate = 0.002
num_epochs = 100
batch_size = 40
Percent of training data containing a hero: 27.60%
Final score: 0.41137

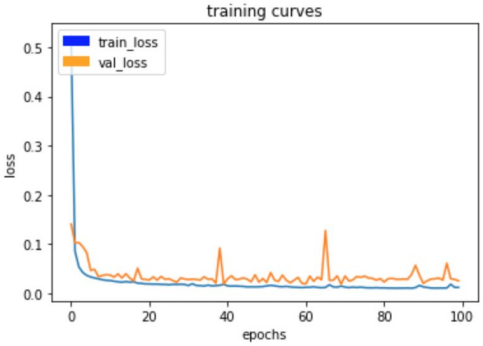



```
116/116 [==============================] - 84s - loss: 0.0090 - val_loss: 0.0257
```

Trial 6

New data set
number_of_training_images = 4131

Optimizer: NadamOptimizer
learning_rate = 0.002
num_epochs = 100
batch_size = 40
Percent of training data containing a hero: 37.60%
Final score: 0.409

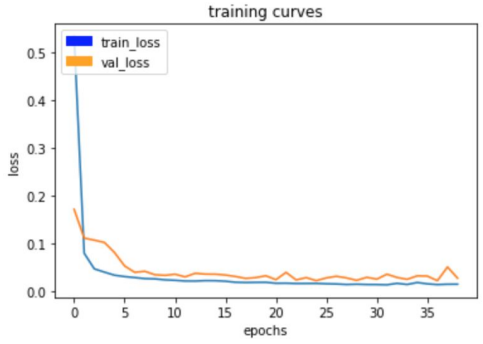



```
104/104 [==============================] - 76s - loss: 0.0115 - val_loss: 0.0249
```

Trial 7

New data set
number_of_training_images = 4131

Optimizer: NadamOptimizer
learning_rate = 0.002
num_epochs = 40
batch_size = 40
Percent of training data containing a hero: 37.60%
Final score: 0.4484




```
104/104 [==============================] - 77s - loss: 0.0157 - val_loss: 0.0282
```

DISCUSSION

I ran a number of trials and experiments with different values of hyperparameters, optimizers, and data as seen in the experiments section above. I used the old given data set provided by Udacity in other trials, and also removed some images (592 images total) of the given data that don't contain any people. This is because it is well known that when you train your model with a dataset is skewed or unbalanced, the classification would bias towards the class with more data. I also augmented the data by flipping all images which doubled the amount of data. This is a well known and simple image augmentation technique used to increase the amount of data as having more data makes the model learn better. These changes increased the percent of training data containing a hero from about 23% to 27%. Comparing trial 5 and trial 4, it seems like some sort of overfitting took place when I increased the number of epochs. Then I tried the new dataset provided which has 37% of images containing the target hero for trial 6. Here, the performance with a score of 0.409 is worse for 100 epochs which makes me feel that it is really overfitting. So I reduced the number of epochs from 100 to 40 and trained it again, this is the final run where the final score is 0.4484 which I am submitting.

Source:
https://medium.com/towards-data-science/image-augmentation-for-deep-learning-histogram-equalization-a71387f609b2
https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

Code used for flipping images

```
import os
import glob
from scipy import misc
import numpy as np

def flip_and_save_images(img_dir, extension):
  os.chdir(img_dir)
  files = glob.glob("*." + extension)
  for i, file in enumerate(files):
    print(i)
    img = misc.imread(file, flatten=False, mode='RGB')
    flipped_img = np.fliplr(img)
    misc.imsave("flipped" + file, flipped_img)

train_mask_dir = "/Users/mithisevilla/Desktop/self-driving/RND-term-1/follow-me/train/masks/"
train_images_dir = "/Users/mithisevilla/Desktop/self-driving/RND-term-1/follow-me/train/images/"

flip_and_save_images(train_mask_dir, "png")
flip_and_save_images(train_images_dir, "jpeg")
```

Code used for detecting the hero/target person

```
img_dir = "/home/ubuntu/follow-me/data/train/masks"
total_files, total_hero = 0, 0
os.chdir(img_dir)
for file in glob.glob("*.png"):
  total_files +=1
  img = misc.imread(file, flatten=False, mode='RGB')
  blue = img[:,:,2]

  if np.any(blue == 255):
    total_hero += 1

percent_hero = 100. * total_hero / total_files

print (percent_hero, "percent of files contain the hero")
```

FUTURE ENHANCEMENTS

One student said in Slack that having a deeper network with more layers would enable learning smaller details which might improve identifying the target from a far away (a far target would mean that it is just a few pixels in size that even a human person would have trouble identifying correctly). Another student suggested that having two sets of

encoder/decoder set would have a better chance of being correct as he read in some papers. It would be nice to check these theories. It is also said that the initial values of the weights play a significant role in the direction of learning. Perhaps we can also play with different initializer techniques.

More importantly I guess it that, it is also widely-known that the training data is almost always a larger factor that the architecture used (I learned this from Andrew Ng's Machine Learning Class). What I think would improve this is to get more data from the simulator with the target person in it, especially pictures where the target person is far away. Also, it is always better have a more balanced data set where the target person is there (maybe have at least 50% of the data). We can also do more image augmentation techniques to increase the data set as discussed in the links below.

Source:
https://medium.com/towards-data-science/image-augmentation-for-deep-learning-histogram-equalization-a71387f609b2
https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html