Deep Learning Project: Follow Me!
Robotics Nanodegree, Udacity

Stephan Hohne

November 5, 2017

Abstract

This report describes my solution of the deep learning project. First I describe the convolutional network as applied to the given semantic segmentation task. Then I describe the data collection process, the network architecture and training process. In the last section I present the results and discuss the performance of the tested models. The code for this project can be found in this repository.

Contents

List of Tables

List of Figures

1 Fully Convolutional Network for Semantic Segmentation

The goal of this project is to build and train a convolutional neural network that enables a drone to follow a person in a simulated environment.

The network is supposed to do semantic segmentation of the given scene and detect the location of the hero person. This task can be accomplished with a fully convolutional network (FCN) which consists of convolution layers only. Such architectures are comprised of an encoding and a decoding stage.

The encoding stage consists of two separable convolutions followed by a regular 1×1 convolution. The purpose of the encoder is to extract spatial features from the image. The complexity of the encoded features increases with the number of layers.

The first encoder layers are implemented as depthwise separable convolutions (see this paper) which perform a spatial convolution while keeping the depth channels separate. The underlying assumption is that the information encoded in the spatial dimensions is independent of the depth channel information, and can be handled separately. This technique reduces the number of parameters, thus increasing efficiency for the encoder network.

The final layer in the encoder block is a 1×1 convolution layer with a stride of one and same padding. The output tensor is of rank four with size $B \times H \times W \times D$ where B is the batch size and D is the filter size. In comparison, a fully connected layer would flatten the output tensor to rank two with size $B \times D$ and thereby loose the spatial information.

The purpose of the decoder block is to scale the encoder output up to the size of the input image and decode the data stored in the previous layers. It is comprised of three parts.

- A non-trainable bilinear upsampling layer. The value of an output pixel is determined by interpolation of its nearest neighbors in the input.
- A non-trainable concatenation layer. It implements skip connections and thereby retains some of the information from the previous higher resolution layers.
- One or more separable convolutions which learn from the data stored in the concatenated layer.

The final layer of the model is a regular convolution with softmax activation. It assigns class probabilities to each pixel in the image.

Figure 1: Drone and hero paths for data collection run 1.

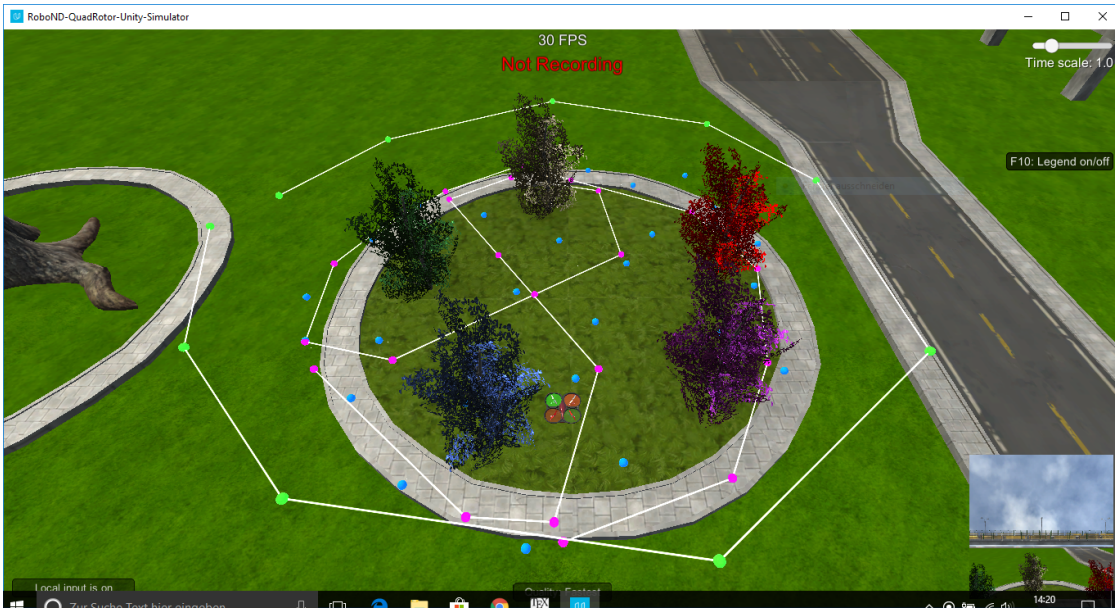

Figure 2: Drone and hero paths for data collection run 2.

2 Data Collection

I collected image data in the `QuadRotor` simulator. The goal was to collect sequences that the drone will encounter in application, with the target person in it. I ran two different path layouts.

1. Drone patrols directly over the hero. The paths are laid out such that the drone follows a straight line and the hero zigzags around that line, as shown in figure 1.
2. The drone circles around the hero and many distracting people. See figure 2.

I collected 270 data sets in the first run and 186 data sets in the second run. One such data set consists of four camera images. I ran the image processing script `preprocess_ims.py` which returned 456 image/mask combinations. I used these data points as additional training data, so I added them to the training data set and left the data sets for validation and evaluation as they were.

The resulting new training data set `train_combined.zip` (file size 56.7 MB) consists of 4587 image/mask combinations and can be downloaded from the repository. I used this data set to train the network as described in section 3.3. The results of this experiment can be found in table 4.

3 Network Architecture and Training

I trained the network on a AWS `p2.xlarge` instance. The relevant code can be found in the notebook `model_training.ipynb`.

The evaluation is done using the intersection over union metric as the final score.

I conducted three different experiments in order to find the combination of network architecture and hyperparameter values that achieves the highest possible final score. I used the data sets `train.zip`, `validation.zip` and `sample_evaluation_data.zip` as a reference, so the results of the training runs are comparable.

For all training runs described here, the number of validation steps is 50 and the number of workers is 4.

3.1 Architecture Parameters

I tuned and tested the following parameters of the FCN architecture described in section 1.

- The filter sizes of the encoder blocks. I started with a depth of 32 for the first layer, 64 for the second layer and 128 for the 1×1 convolution.

layer	purpose	volume	implementation
`input`	input	$160 \times 160 \times 3$	`InputLayer`
`layer_1`	encoder	$80 \times 80 \times 32$	`SeparableConv2D`
`layer_2`	encoder	$40 \times 40 \times 64$	`SeparableConv2D`
`layer_3`	1×1 conv	$40 \times 40 \times 128$	`Conv2D`
`layer_4`	decoder	several sizes	`decoder_block`
`layer_5`	decoder	several sizes	`decoder_block`
`output`	output	$160 \times 160 \times 3$	`Conv2D`

Table 1: Overview of FCN model used for semantic segmenation of drone images, as shown in figure 3. For implementation and detailed model summary see the notebook `model_training.ipynb`.

- The number of separable convolution layers in the decoder block. I started with a two layer configuration.

For setting the initial values of these parameters I used knowledge from the classroom material, the segmentation lab and communication with other students on Slack. One such advice is to keep the network simple at the outset and add complexity later.

To test these assumptions I ran the architecture experiment as shown in table 2. I compared the filter size sequence $32 \rightarrow 64 \rightarrow 128$ with the sequence $32 \rightarrow 64 \rightarrow 128$. I tested the usage of one and two separable convolution layers in the decoder block.

The network with two separable convolutions in the decoder and a filter size sequence of $32 \rightarrow 64 \rightarrow 128$ achieved the highest final scores. Therefore I used this configuration in the subsequent experiments. This network is described in table 1 and depicted in figure 3.

3.2 Hyperparameter Tuning

I tested the reference architecture described in section 3.1 with a variety of values for the learning rate, the batch size and number of epochs. I used the reference data sets for training, validation and evaluation. The results of this experiment are given in table 3. The corresponding model files can be found in the `weights` directory named as `model_weights_tune_n` where `n` is the number of the run.

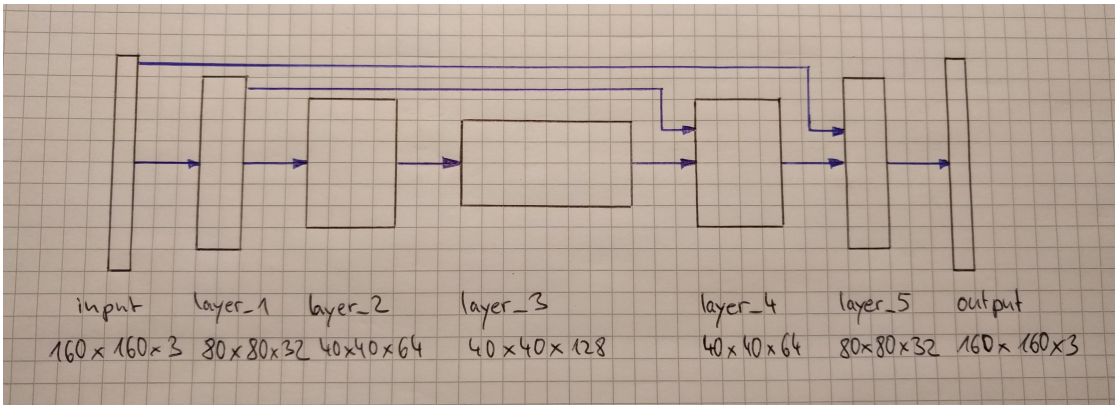

Figure 3: Diagram of FCN model. Rectangles represent input/output layers and encoder/decoder blocks. Blue arrows represent connections. Layer names follow the notebook `model_training.ipynb`. For further model properties see table 1. The internal structure of the blocks is not shown in this diagram.

Run	1	2	3	4
Architecture				
Decoder convolutions	2	1	2	1
Maximum depth	128	128	256	256
Evaluation: Final Score				
Run `a`	0.3908	0.3213	0.3834	0.3338
Run `b`	0.4277	0.3576	0.3955	0.4030
Average $\pm$ deviation	0.41 ± 0.02	0.34 ± 0.02	0.39 ± 0.01	0.37 ± 0.04

Table 2: Results of the architecture experiment. Maximum depth refers to the filter size of the 1×1 convolution layer. Learning rate is 0.005. Batch size is 32. Network configurations are trained over 10 epochs with 200 steps. The final scores are rounded to four significant digits for the individual runs and two significant digits for the average and deviation over the runs.

Run	1	2	3	4	5	6
Hyperparameter Values						
learning rate	0.008	0.008	0.008	0.008	0.005	0.005
batch size	20	32	32	32	32	32
number of epochs	4	4	10	16	16	50
steps per epoch	200	200	200	200	200	200
Evaluation						
final score	0.3198	0.4009	0.4260	0.4054	0.4387	0.4285

Table 3: Results of the hyperparameter tuning experiment on the reference architecture (see figure 3) using the reference data sets for training, validation and evaluation.

I started with default values from the lab and trained 4 epochs. In the subsequent runs I increased the number of epochs and observed an improving final score. At 16 epochs I noticed the final score dropping. So I decreased the learning rate in order to prevent overfitting. I ran experiment 5 with a learning rate of 0.005 and the final score improved.

In run 6 I increased the number of epochs further to 50. The training curve of this run is shown in figure 4. The final score did not improve compared to the previous run.

From the results of the tuning experiment I deduced that the current best learning rate to work with is 0.005.

I left the number of steps per epoch at 200, in future work this can also be tuned. The number of epochs can be increased and the number of steps decreased.

3.3 Training with additional data

When I had explored several configurations of architecture and hyperparameter values, I started to use the data collected in the simulator. I took the reference network as shown in figure 3 and the hyperparameter values of tuning run 6 (see table 3). I trained this configuration on the combined data (see section 2 for the data collection process). The results are given in table 4.

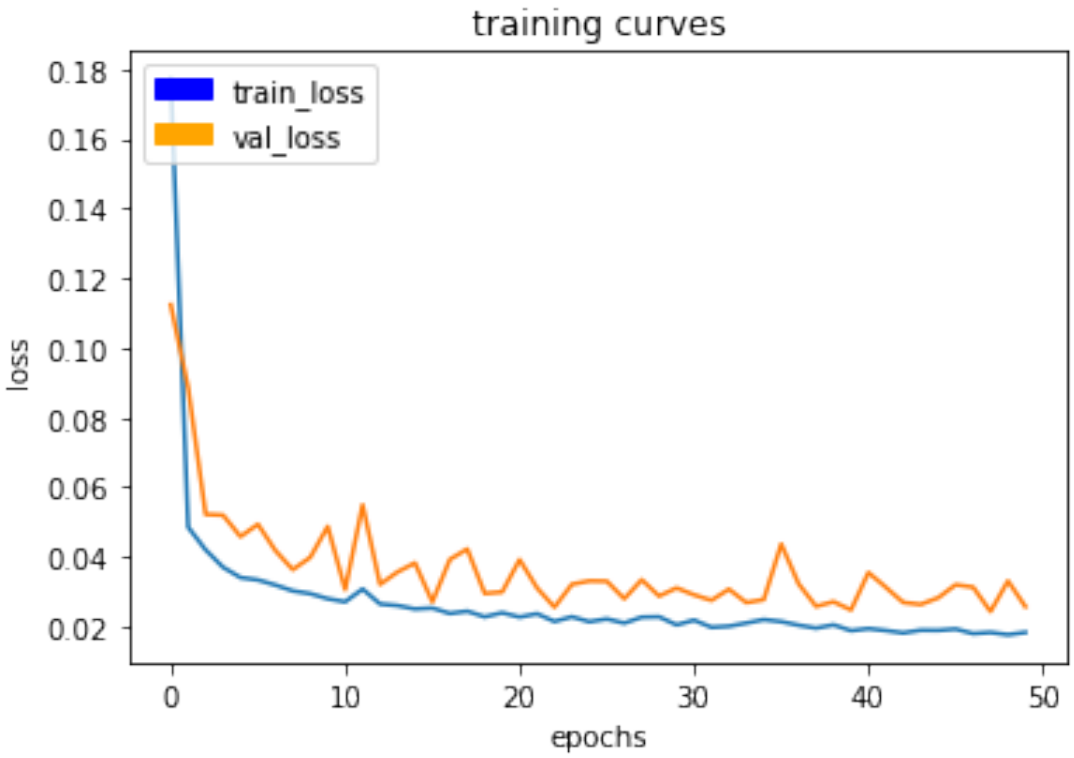

Figure 4: Training curves for tuning run 6 with 50 epochs at a learning rate of 0.005.

Run	1	2
Hyperparameter Values		
learning rate	0.005	0.005
batch size	32	32
number of epochs	16	50
steps per epoch	200	200
Evaluation		
final score	0.3734	0.4044

Table 4: Results of the collected data experiment using the collected data set for training and the reference data sets for validation and evaluation. The experiment is performed on the network configuration described in 1 and figure 3.

4 Results

The model trained in tuning run 6 (last column in table 3) shows the best performance. It reaches final scores of approximately 0.43. Therefore I submit the corresponding weight file `model_weights_tune_6`.

Against expectations, gathering additional data did not have a significant effect on the network performance. I assume that this can be improved by collecting much more image sequences, where the hero is present in the camera image, and with more variation in the background scenery.

4.1 Future Enhancements

The following improvements of the network architecture, the hyperparameter tuning and the data collection process seem to be promising next steps to increase the performance of the model.

- Tune the number of steps per epoch. Increasing the number of epochs and decreasing the number of steps per epoch might increase the model performance while taking the same training time.
- Experiment with learning rate decay and early stopping. Take full advantage of all the options provided by the Keras optimizers package.
- Use `sklearn.grid_search.GridSearchCV` to automate the hyperparameter tuning experiments.
- Collect more image data, possibly with the target being present in a higher fraction of the camera images.
- In the decoder block, use transpose convolutions (see Keras) instead of the less accurate bilinear upsampling technique.