Semantic Segmentation with Keras

initial experiments and implementation details to help you get started

Rafael Espericueta

Inst. of Computer Graphics and Vision
Graz University of Technology, Austria

Technical Report
ICG–TR–001
Graz, August 14, 2017

contact: Rafael Espericueta rafaelespericueta@gmail.com

Abstract

Getting started with a new platform is often the most difficult barrier to using it effectively. This document is primarily meant to help researchers new to Keras get started using it for semantic segmentation, with minimal frustration, and to provide some useful starter code.

Keywords: *Technical report, Keras, Semantic segmentation*

1 Overall problem description and goal

Keras is proving to be an increasingly popular platform for deep learning due to its relative ease of use, its versatility, and its ability to serve as a front-end for TensorFlow, Theano, and more recently MicroSoft's framework, CNTK. Keras has also been incorporated into TensorFlow itself (and optimized for TensorFlow). The goal of this work is to implement Martin Hirzer's semantic segmentation architecture [1], but using the Keras platform with a TensorFlow back-end, and to apply it to Martin's Graz city segmentation dataset. Our hope is that this groundwork will ease other researchers' transition, should they wish to begin using Keras in their own work. A secondary and more longterm goal is to explore various other architectures that researchers have used with great success for semantic segmentation, such as ResNet and DenseNet, to compare the results. Future iterations of this report will continue to pursue these goals. Stay tuned!

2 Deep learning for semantic segmentation

The initial architecture implemented was based on that described in [1,2]. This uses an encoder/decoder configuration with an overall hourglass shape, with a bottleneck in the middle. VGG-16 was used for the encoder, with its final classification layer replaced by a decoder that uses deconvolution layers to ultimately recapture the original pixel structure of the input image, but now segmented into four classes (background, facade, vertical facade edges, and horizontal facade edges). To help recapture pixel-level information, a skip-connection connedted the convolved output of pool4 in the encoder to the output of a cropped deconvolution layer (layer 24) of the decoder. A listing of all the model layers (output from `model.summary()`), is shown below.

```
Layer (type)                     Output Shape          Param #     Connected to
==================================================================================================
input_1 (InputLayer)             (None, 560, 840, 3)   0
__________________________________________________________________________________________________
block1_conv1 (Conv2D)            (None, 560, 840, 64)  1792        input_1[0][0]
__________________________________________________________________________________________________
block1_conv2 (Conv2D)            (None, 560, 840, 64)  36928       block1_conv1[0][0]
__________________________________________________________________________________________________
block1_pool (MaxPooling2D)       (None, 280, 420, 64)  0           block1_conv2[0][0]
__________________________________________________________________________________________________
block2_conv1 (Conv2D)            (None, 280, 420, 128) 73856       block1_pool[0][0]
__________________________________________________________________________________________________
block2_conv2 (Conv2D)            (None, 280, 420, 128) 147584      block2_conv1[0][0]
__________________________________________________________________________________________________
block2_pool (MaxPooling2D)       (None, 140, 210, 128) 0           block2_conv2[0][0]
```

```
____________________________________________________________________________________________________
block3_conv1 (Conv2D)            (None, 140, 210, 256) 295168      block2_pool[0][0]
____________________________________________________________________________________________________
block3_conv2 (Conv2D)            (None, 140, 210, 256) 590080      block3_conv1[0][0]
____________________________________________________________________________________________________
block3_conv3 (Conv2D)            (None, 140, 210, 256) 590080      block3_conv2[0][0]
____________________________________________________________________________________________________
block3_pool (MaxPooling2D)       (None, 70, 105, 256)  0           block3_conv3[0][0]
____________________________________________________________________________________________________
block4_conv1 (Conv2D)            (None, 70, 105, 512)  1180160     block3_pool[0][0]
____________________________________________________________________________________________________
block4_conv2 (Conv2D)            (None, 70, 105, 512)  2359808     block4_conv1[0][0]
____________________________________________________________________________________________________
block4_conv3 (Conv2D)            (None, 70, 105, 512)  2359808     block4_conv2[0][0]
____________________________________________________________________________________________________
block4_pool (MaxPooling2D)       (None, 35, 52, 512)   0           block4_conv3[0][0]
____________________________________________________________________________________________________
block5_conv1 (Conv2D)            (None, 35, 52, 512)   2359808     block4_pool[0][0]
____________________________________________________________________________________________________
block5_conv2 (Conv2D)            (None, 35, 52, 512)   2359808     block5_conv1[0][0]
____________________________________________________________________________________________________
block5_conv3 (Conv2D)            (None, 35, 52, 512)   2359808     block5_conv2[0][0]
____________________________________________________________________________________________________
block5_pool (MaxPooling2D)       (None, 17, 26, 512)   0           block5_conv3[0][0]
____________________________________________________________________________________________________
conv2d_1 (Conv2D)                (None, 11, 20, 4096)  102764544   block5_pool[0][0]
____________________________________________________________________________________________________
dropout_1 (Dropout)              (None, 11, 20, 4096)  0           conv2d_1[0][0]
____________________________________________________________________________________________________
conv2d_2 (Conv2D)                (None, 11, 20, 4096)  16781312    dropout_1[0][0]
____________________________________________________________________________________________________
dropout_2 (Dropout)              (None, 11, 20, 4096)  0           conv2d_2[0][0]
____________________________________________________________________________________________________
conv2d_3 (Conv2D)                (None, 11, 20, 4)    16388       dropout_2[0][0]
____________________________________________________________________________________________________
conv2d_4 (Conv2D)                (None, 35, 52, 4)     2052        block4_pool[0][0]
____________________________________________________________________________________________________
conv2d_transpose_1 (Conv2DTransp (None, 24, 42, 4)     256         conv2d_3[0][0]
_____________________________________________________________________________________________________
cropping2d_1 (Cropping2D)        (None, 24, 42, 4)     0           conv2d_4[0][0]
______________________________________________________________________________________________________
add_1 (Add)                      (None, 24, 42, 4)     0           conv2d_4
                                                                   cropping2d_1[0][0]
_______________________________________________________________________________________________________
conv2d_transpose_2 (Conv2DTransp (None, 400, 688, 4)   16384       add_1[0][0]
_______________________________________________________________________________________________________
cropping2d_2 (Cropping2D)        (None, 360, 640, 4)   0           conv2d_transpose_2[0][0]
_________________________________________________________________________________________________________
activation_1 (Activation)        (None, 360, 640, 4)   0           cropping2d_2[0][0]
=========================================================================================================
Total params: 134,295,624
Trainable params: 119,580,936
Non-trainable params: 14,714,688
```

Martin Hirzer actually (following [2]) used three different models, of increasing power and complexity, where the trained weights of a simpler model were used as initial weights of the next, more complex model. Due to time limitations, so far only the second of these models (in terms of complexity) was implemented. Unlike in Martin's work, [1], the VGG-16 encoder was initialized with ImageNet weights (thus the large number of non-trainable parameters described in the above table), to see if transfer learning would be beneficial in training the model.

3 Implementation details

Keras is currently in rapid evolution, and occasionally one needs to work around occasional bugs or places where Keras needs to be extended. Whenever this was needed, it was carefully noted in the comments within the code, which can be found in the following github repository:

https://github.com/shimmeringvoid/FCN_VGG16

Once the model was compiled, it took over 1 GB of storage, and proved too large to run (as is) using the available GeForce 780 GPU due to its lack of memory. To test the code, it had to be run using only the CPU. This made a full run on the entire dataset impossible due to time constraints.

4 Results

As a very preliminary test, the code was run on the first 10 images of the dataset only, using 11 epochs of 100 images each (and their associated label masks), randomly selected with replacement from the 10 images. The encoder's weights were frozen, for this initial experiment. These first 10 images were successive frames from a video, so were almost identical. The model was able to memorize these images with an accuracy of over 0.98.

Figure 1 shows the semantic segmentation predictions that resulted from one of the training images being input to the (over)trained model, along with three predictions for later images that weren't part of the training set. As expected, the further the images were from the ones used for training, the poorer the segmentation results. Generalization is of course severly limited with only 10 extremely similar training examples.

As a second experiment, the model was trained on 100 images randomly selected from the entire training set of 29,146 images. This time, the training consisted of 100 epochs of 100 image-label pairs. A typical training image prediction is depicted in Figure 2. It appears that the network is mainly learning that some classes are more probable than others at various heights within an image. A much larger training set is needed to attain meaningful segmentation results. We hope to soon remedy this situation.

In cases with imbalanced class representation, it's often useful to use class weights to help relatively improbably classes fare better against more

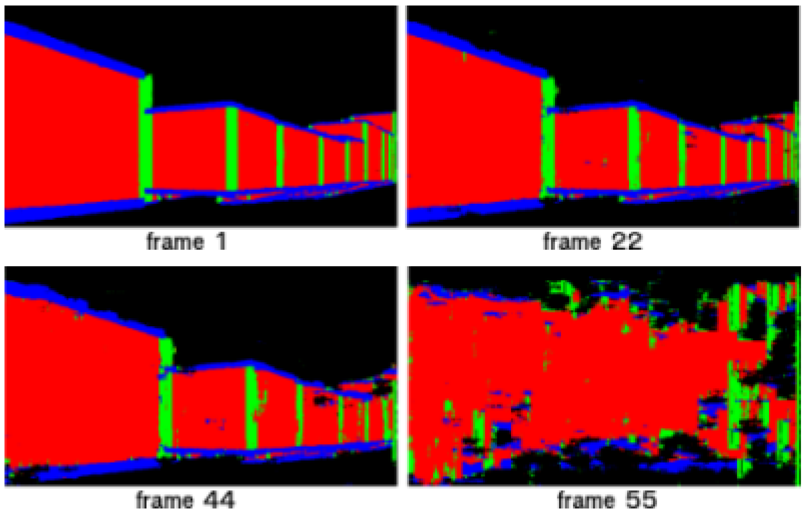

Figure 1: Segmentation predictions, from training on frames 1 through 10

frequently occuring classes. An attempt was made to implement median frequency balancing as described in 3, using the convenient `class_weight` parameter passed to the `fit_generator` Keras function. Alas, it isn't yet implemented for dense prediction as needed for semantic segmentation. A web search provided a possible work-around, though it seems a bit of a kludge, and this hasn't yet been implemented. The kludge involves flattening out the 2D array (of pixels), and telling Keras that it's a time series, and then making use of a `sample_weight` parameter, with all the weights equal. This will be implemented in the near future, when the model is trained on the entire dataset.

However, the code `median_freq_balancing.py` computes the necessary weights from the labels, for a future implementation using class weights. These weights are calculated as in [3] by the formula $w_c = \frac{\text{median frequency}}{\text{freq}(c)}$, for $c \in \{0, 1, 2, 3\}$, where $\text{freq}(c) =$ the probability that a randomly selected pixel is of class c, and the median frequency is simply the median of the four $\text{freq}(c)$ values. These weights are used to weight each class in the calculation of cross-entropy loss. The more frequently occuring classes end up with weights < 1, whereas the least frequently occuring classes will have weights > 1. The rounded weights for the entire 29,146 labels, as revealed by `median_freq_balancing.py`, were [0.80, 0.18, 1.33, 2.28], reflecting the fact that facade pixels were most likely (red), followed by background (black), while horizontal edges were least likely (blue), followed by the second least likely vertical edges (green).

For a third experiment, the second experiment was repeated, this time

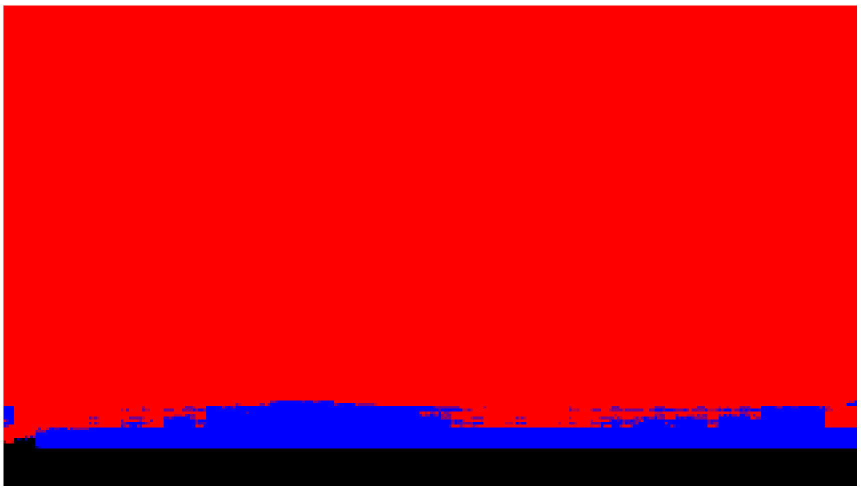

Figure 2: A typical segmentation prediction from experiment 2

using a more expressive model, by unfreezing the last two convolution blocks of the VGG-16 encoder. However, not surprisingly, this wasn't sufficient to noticibly improve the segmentation predictions of our model; indeed, it fared much worse. The increased number of weights to be trained results in a need for even more training data. There simply is no substitute for a sufficiently large amount of data, when it comes to deep learning.

These experiments are highly prelimary, but do at least provide evidence that the model may well work, given a sufficiently large training set. Training the net on the entire dataset using a GeForce Titan X GPU is planned for the near future, and the results will be included in a future iteration of this report.

5 Appendix: Setting up Keras, and using this framework

To set up Keras, one must fist set up TensorFlow. It's recommended to set up TensorFlow in two separate virtual environments, one that uses your GPUs and the other that only uses the CPU. This proved useful for this work; when the GPU was found to have insufficient memory, one could nonetheless debug the model using the CPU-only version. The most recent version of TensorFlow actually has Keras included with it. However, for now it's probably best to install Keras seperately and import what you need from it into your Python environment. In the future it will be best to use the TensorFlow internal version, since that is being optimized to run using a TensorFlow back-end. But for now, it should be considered a beta version. There are already detailed instructions for installing TensorFlow in its two flavors, so I won't reiterate that here. Keras installation into your TensorFlow virtual environemnts is trivial (please refer to its online installation instructions).

Further information about how to use this framework is included in the Github repository https://github.com/shimmeringvoid/FCN_VGG16 . In particular, the python code there is extensively commented, which should prove useful to people with no previous Keras experience.

6 References

[1] M. Hirzer, Efficient 3D Tracking in Urban Environments with Semantic Segmentation

[2] J. Long, E. Shelhamer, and T. Darrell. Fully Convolutional Networks for Semantic Segmentation.

[3] D. Eigen, R. Fergus. Predicting Depth, Surface Normals and Semantic Labels with a Common Multi-Scale Convolutional Architecture