# Computer Vision Project - Fully Convolutional Networks for Semantic Segmentation

Oren Nuriel, Aviad Weinstein

The Hebrew University of Jerusalem

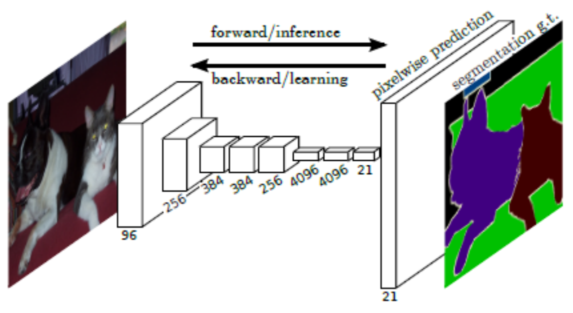
***Abstract*— This paper was published in CVPR2015. The purpose of the paper was to use a fully convolutional network, in order to predict a pixelwise semantic segmentation of an image. Adapting classification networks (VGG net) into a fully convolutional network and fine-tuning it, made this paper feasible.**

***Keywords*— Fully Convolutional Neural Networks, Semantic Segmentation, Computer Vision.***

## II. Introduction

### A. *Background*

Semantic segmentation is "breaking down" the image into semantic contexts, such as: people, cars, streets, etc.



The advantage of using a fully convolutional network is the ability to classify any size image, and produce an output of corresponding spatial dimensions.

### B. *Shift-and-stitch - The naïve approach*

Semantic segmentation predictions can be obtained from coarse outputs by stitching them together. This approach was successful in related papers, although it might be unfeasible given certain circumstances, due to the increasing amount of times required to run the network (or algorithm).

## III. Architecture

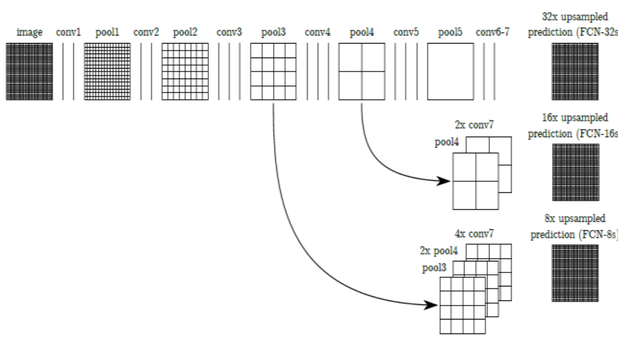


image
conv1
pool1
conv2
pool2
conv3
pool3
conv4
pool4
conv5
pool5
conv6-7
32x upsampled prediction (FCN-32s)
2x conv7
pool4
16x upsampled prediction (FCN-16s)
4x conv7
2x pool4
pool3
8x upsampled prediction (FCN-8s)

### A. *Network Architecture*

The researchers used the VGG network as a base point to start their network. As seen in the image above, some changes were made to the basic architecture. First, the researchers changed all fully connected layers to convolution layers using a 7x7 filter to reduce the original VGG  input layer of to a 1x1 using the same weights, and then using a 1x1 filter to replicate the other fully connected. Second, they replaced the last fully connected layer, used for prediction, with a 1x1-convolution layer with 21 outputs, used for our network's forecast. Third, adding up-sampling layers turning the image to its original spatial dimension, and outputting a pixelwise prediction (in our case, there are 21 classes in the PASCAL VOC dataset). Last, the researchers added skip connections showing empirically that the more skip connections the better the results.

### B. *Skip connections*

The researchers tested the efficiency of the skip connections. They tested three combinations:

1) *FCN32*

This architecture contains no skip connections. The results achieved using this network were the least efficient of the three.

2) *FCN16*

This architecture contains one skip connection. After down sampling to an image of shape (h/32, w/32, 21 channels), in order to fuse the skip layer it is required to do two things: First, up sample back one-step, so these layers are of the same dimension. Second, to convolve the output of the pooling layer from 512 to 21 channels so it is possible to fuse them channel-wise. As expected, this architecture showed average results and reached second place between these combinations.

3) *FCN8*

This architecture is the most complicated as it contains two skip connections. As explained in FCN16, using the same technique, there needs to be a correction to the network so the up sampling process will work. The change that the researches did was to up sample once and fuse the results with the first skip layer, and then up sample again and fuse with the second skip layer, and finally they up sample to the original image size.

### C. *Padding*

There are two approaches to dealing with pixels lost due to convolution filters:

*a)* Using a padding for each layer independently to keep the dimension of image the same.

*b)* Padding the input layer with 100 zeroes to each direction in both height and width. With this approach, the padding on the fully convolutional 7x7 layer is 0 or 'valid'. This is the researchers approach.

From our experience the results obtained from the first approach achieved higher performance.

### *D. Weight initialization*

As mentioned before we initialize the networks weights with VGG16 weights. We will elaborate about specific layers that may need a more detailed explanation:

*a) The fully connected layers*

As we explained the fully connected layers were transformed to convolution layers with 7x7 and 1x1 filters, the weight initialization here does not change since the amount of weights in both layers are the same, they are just flattened.

*b) The skip layers*

These layers are initialized by keras defualt, since there is no special referral in the paper about these weights.

*c) Up-sampling*

These layers are initialized using weights that imitate bilinear interpolation using transposed convolution as mentioned in the paper.

## IV. Learning Details

### *C. Batch size*

This hyper parameter was a bit tricky, since our program calculates during running time the image's actual size for up sampling purposes. We encountered a problem if the images were not of the same size. After searching the web, we found several solutions for this problem:

*1) Batch size of one image*

By using one image as a batch no unfitting shape exceptions arises during running time, but by doing so we create two additional problems:

*a) Instability*

Since the networks gradient step is based only on one image this makes the steps very noisy, to fix this we use optimizers that take into consideration momentum.

*b) Running time*

Since every step uses only one image, we need to do a large number of steps, this factor increases time greatly.

*2) Batches containing only the same size of images*

Another solution is to use batches of images of the same size. The fear in this method is not having a variety of different batches for the network to train on, such a thing that can hurt the networks preference.

*3) Augmentation*

By taking patches from the images and learning on them we can create batches of the same size images. The researchers wrote in the paper that they have tried this method in order to test if it increases the network preference, but their results did not change significantly in this approach.

### *D. Optimization*

We tried different optimization techniques:

*1) SGD with momentum*

In the paper, the researchers mention that during training they used this optimizer on batches of size one. The main idea using this optimizer with a high momentum is giving a significant weight to history examples rather than the example encountered in this batch. We found that the results using this optimizer were the "worst".

*2) AdaDelta*

The method dynamically adapts over time using only first order information and has minimal computational overhead beyond vanilla stochastic gradient descent. There is no need for a default learning rate using this method.

*3) Adam*

This optimizer adapts the moment and learning rate during running time. This optimizer is very popular and is very effective. We found its results the most promising.

### *E. Training data*

For training, we used PASCAL VOC2011-12 and SBD as the researchers did. This dataset contains 8,948 images and 21 feature classes. In addition, we've tried a couple of other data sets (that hold a bigger amount of feature classes), wanting to see the results on all kinds of scenery, the additional datasets were: ADE Challenge Data 2016 and SBD alone. Unfortunately, the results we obtained for these datasets were bad, so we do not present their results.

### *F. Learning rate*

There is not much to say about this hyper parameter since the researchers wrote exactly what number they used, and said these numbers are the best they found. Since Adam optimizer was used this hyper parameter was less significant.

### *G. Computational power*

Since we have no super computer, we found this problem to be the most challenging. We have tried a few solutions until we found one that works:

*1) Running on home computer*

This solution was problematic for two reasons:

*a) No GPU*

Since most of our computers had no GPU, the running time of a single training step was estimated to run for two minutes, and as we have 100,000 training steps this option would take approximately 135 days.

*b) GPU too small*

Since the GPU, in an effort to minimize time, saves a large amount of the data on the card itself, we found that even a 4GB card is not big enough for holding the model including the back propagation steps for training without taking into consideration the images needed for training.

*2) Runnning on the aquarium computers*

This solution looked promising at start, but after running about one epoch the program would be killed by the system without even raising a exception so we could not even save our progress.

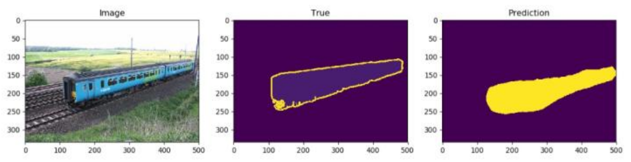
### 3) *Google platform*

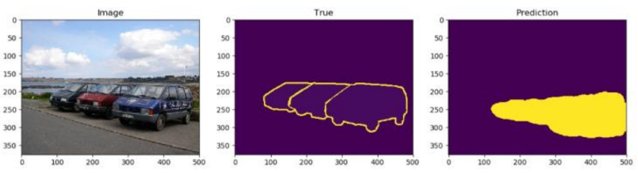
At last we found a working solution, Google's platform. By outsourcing our program to Google's servers we had managed to run our program for the whole training at once using Tesla V100 GPU.

## V. Results

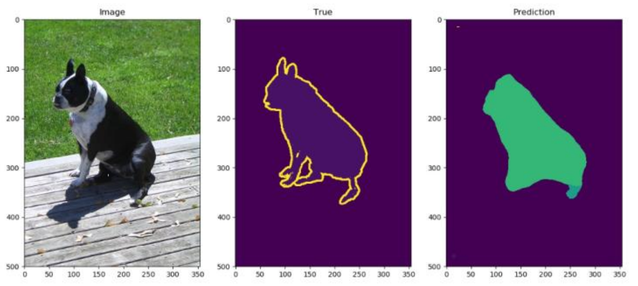
Unfortunately, all of our efforts did not recreate the paper's results. Our network managed to achieve 30% IoU on the seg11val test set, with 80% pixelwise accuracy and loss of only 0.8. We do not know why the results obtained were substantially different than the paper's. Finally, to check if our network architecture is correct and suitable for the problem, we downloaded pre-trained weights for our problem and visualized the results. In our opinion, these weights do a excellent job of semantic segmentation on the data and the results can be seen below:

Our training results:





The downloaded weights:



## VI. Appendix

### C. *Code Usage*

#### 1) *Installation*

To run our code on your computer you need to have the following standard libraries that are already available in the aquarium computers: numpy, tensorflow, keras, and matplotlib.

#### 2) *Downloads*

In order to run our code there is a need to download all dependences. Our code can be found on our github user:

```
https://github.com/onuriel/Semantic-
Segmentation
```

And due to file sizes the additional data can be found on our google drive:

```
https://drive.google.com/drive/folders/14N7w
iv-nt482pw9I806DqDgtPk5JdD7Y?usp=sharing
```

#### 3) *Prepare data*

After downloading everything You should run from the main directory the following code:

```
python3 utils/utils.py
```

After the program finished running you should place the `seg11valid.txt` file in the `benchmark_RELEASE\dataset` directory.

If you download the data on your own, before running the program you need to create a folder named "SegmentationData" in the "VOC2012" folder. Then run the utils.py from inside the "utils" folder using this command:

```
python3 utils.py
```

#### 4) *Running*

Here are instructions on running our code once you have all files needed:

a) *Run the training and ploting random result :*

```
python3 FCN.py -training True
```

b) *For adding a batch size to the training Run:*

```
python3 FCN.py -training True -batch_size
(batch size)
```

c) *Loding the pre trained weights and plot random results:*

```
python3 FCN.py
```

## References

[1] Jonathan Long, Evan Shelhamer, Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation"(2015).