Semantic Segmentation

Artificial Intelligence for Embedded Systems

Uzair Akbar*, Hugo Schrallhammer†, Saqib Javed‡, Saad Bin Shams§
* † ‡ § Department of Electrical and Information Engineering
Technical University of Munich (TUM)
Germany
Email: *uzair.akbar@tum.de, †hugo.schrallhammer@tum.de, ‡saqib7.javed@tum.de, §saad.shams@tum.de
Matric. No.: *03697290, †03660308, ‡03697880, §03693218

***Abstract*—This project aims to explore the use of fully convolutional networks to tackle the problem of semantic segmentation. Such networks are able to take inputs of arbitrary size and produce corresponding sized outputs, with efficient inference and learning. Casting of already trained contemporary networks into a fully convolutional architecture and fine-tuning the weights for our own task were explored. For our project we focused on *VGG*, and GoogleNet classification networks, and a comparison was done on their performance for our application. Lastly, hardware-accelerated inference of the final network was done on Raspberry Pi 3 after model reduction by filter pruning.**

***Index Terms*—Semantic Segmentation; Transfer Learning; Fully Convolutional Networks; Model Reduction.**

I. Objectives and Expected Significance

We initially started off with an already trained contemporary classification network and only trained the deeper, decision making layers. The classification networks which we aimed to use was *VGG*. We then planned on casting such classification networks into a fully convolutional architecture and then add deconvolution layers to re-construct the original image dimensions.

Semantic segmentation has a lot of use-cases these days. Most of them, such as autonomous driving, image-based search and medical-imaging have real-time or near real-time requirements. Therefore, it is also of interest to make our segmentation network in such a way that it is able to fit into the memory of an embedded system (here, a Raspberry Pi) and has a reasonable time-delay. Using smaller models and model reduction techniques have been explored for this very reason.

II. Rationale and Related Background

Image recognition or classification is one of the fundamental problems in the field of computer vision. Having now become rather trivial ever since the advent of AlexNet in 2012 [1], the interest has shifted more towards local tasks like bounding box object detection, part and key point prediction and local correspondence.

Semantic segmentation falls naturally along this progression towards more fine-grained image classification. Put simply, semantic segmentation refers to pixel-wise classification; classifying every pixel within an image into semantically interpret-able classes. By "semantically interpret-able," we mean that the classes have some real-world meaning. For instance, we might want to color all the pixels of an image that belong to cars as blue. However, in unsupervised segmentation methods, the result is not necessarily semantic and instead it finds region boundaries more generally [3]. Hence, semantic segmentation lets us achieve a much more detailed understanding of images than image classification or object detection. This is critical in a broad range of fields, including autonomous driving, medical imaging, and image search engines.

Convolutional networks have been driving advances in a wide range of areas in computer vision, including semantic segmentation [4]–[6]. However, most such approaches involve complicated pre- and post-processing, such as superpixels and edge-detection. In contrast, because a fully convolutional neural network extracts such low-level features automatically in shallower layers, it precludes the need for such complications in addition to being more efficient for training and providing flexibility in terms of datasets (a fully convolutional model can accept images of any size as input).

To avoid training a network from scratch and for quick prototyping, we shall initially explore adapting contemporary networks (VGG net, LeNet, etc.) to our problem and fine-tuning the weights.

III. Approach

Our main approach for the stated problem is derived from [2], where the authors have used famous classification networks, and modified them for semantic segmentation. The authors in [2] show that VGG performs best for such a task as shown in Table I. Also of interest is GoogLeNet because of it's low number of model parameters, which we shall discuss later.

TABLE I
Comparison of modified AlexNet, VGG16 and GoogLeNet for segmentation as shown in [2].

Particular	FCN-AlexNet	FCN-VGG	FCN-GoogLeNet
Accuracy (IoU)	39.8	**56.0**	42.5
Parameters	57M	134M	**6M**

Due to better performance for the task at hand (Table I), we chose *VGG16* for our architecture. We casted VGG16

into a fully convolutional architecture by removing the last fully connected layers and appending equivalent convolutional layers and copying over the weights from the fully connected layers. We then added a deconvolution layer at the end so as to reconstruct the original dimensions of the input image. With this new architecture (now referred to as FCN32s) we were able to reach a pixel-wise cross-validation accuracy of *60.027%*. We then took further steps to overcome the challenges that prevented our model from reaching a higher, more reasonable performance.

IV. Challenges and project Outcomes

Despite decent preliminary results, we continued to identify and solve the problems which prevented our model from performing even better. As for implementation of our model on an embedded system, we wanted to have a good balance between accuracy and the timing profile. Each challenge and how we catered to it is explained the subsections that follow.

A. Intersection Over Union (IoU) and Dice Coefficient

Up-to this point we were using cross-entropy as a loss function for training our model and pixel-wise accuracy as a cross-validation metric. Having explored better loss functions and evaluation metrics in the context of semantic segmentation, we came across Intersection over Union (IoU) and Dice coefficient. Intersection over union is basically area of overlap over area of the union of an image label with the ground truth, which is a far better evaluation metric for semantic segmentation. Furthermore, we used the Dice Coefficient instead of cross-entropy in the loss function. The dice coefficient can be summed up as follows:

$$Dice\ Coefficient\ =\ \frac{2|X \cap Y|}{|X| + |Y|}$$

$$Dice\ Coefficient\ =\ \frac{2TP}{2TP + FP + FN}$$

We then compared the results of the cross-entropy loss function and the dice loss function and then also both of them combined. The results have been combined together inside the figure 1. As it is apparent we get the best of the both worlds when we use both the loss functions together.

B. Unfreezing shallow layers and data augmentation

Up-till now we had frozen the shallow feature extraction layers during training. This was done for faster prototyping and also to prevent our very large network to over-fit on our very small dataset. To further increase the accuracy of our model we now unfreeze the shallow layers so that they are also trained, this however poses a problem of over-fitting because of the small data set that we are using. Therefore in order to avoid over-fitting we have to introduce data augmentation (random rotation, flipping and random crops of the input image). After this the model was retrained and we were able to reach an accuracy of more than 70%. This can be seen in the figure 2.

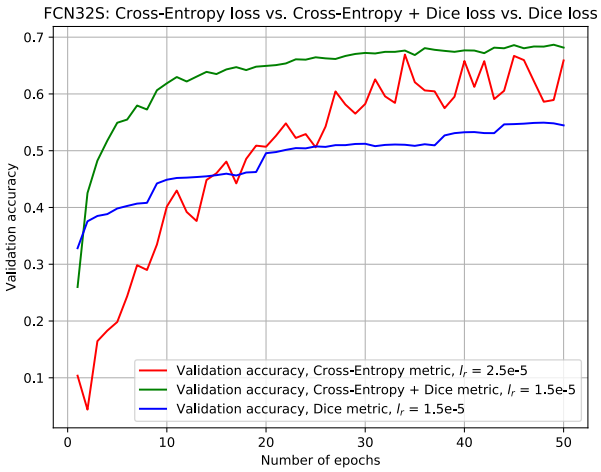


Fig. 1. Dice loss, entropy loss and combined loss comparison

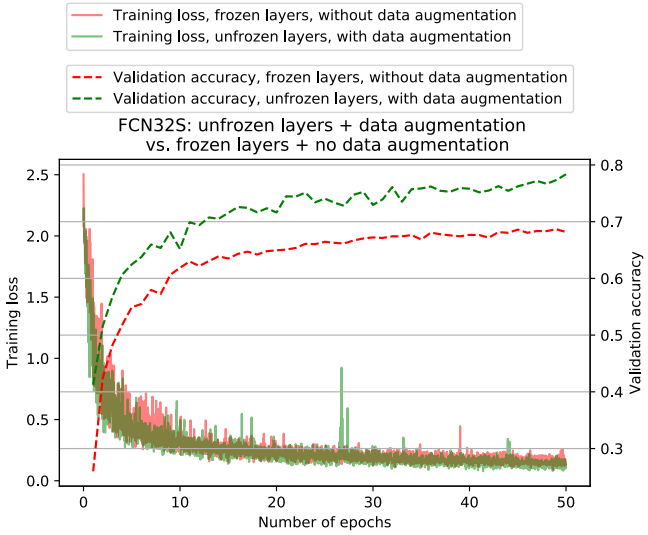


Fig. 2. Unfreezing and data augmentation increases accuracy

C. Finer Segmentation

According to [2], if we want to have even better accuracy and better segmentation we need to include skip connections from the shallow layers to the deeper layers. What this does is that it adds low level features directly into the decision layers. This effectively increased our model's pixel wise accuracy to 88.01% and the Intersection over union of 0.81. This can be seen in the figure 3.

We now refer to this model as FCN8s as shown in Figure 8. This was the best architecture we could train that came very close to the accuracy of the one the authors achieved in this

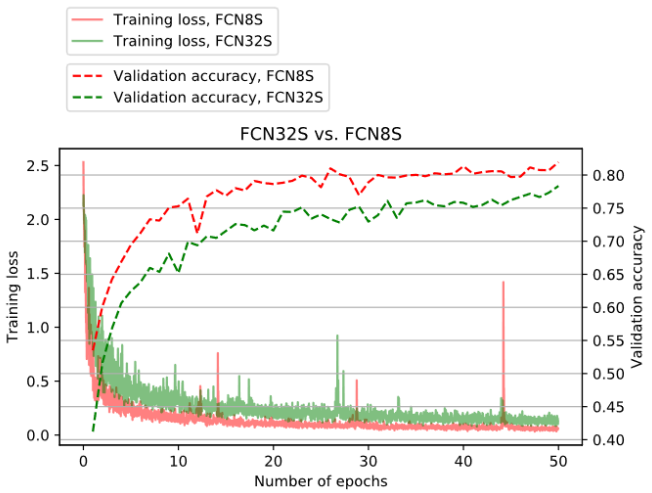


Fig. 3. Adding skip connections increased accuracy

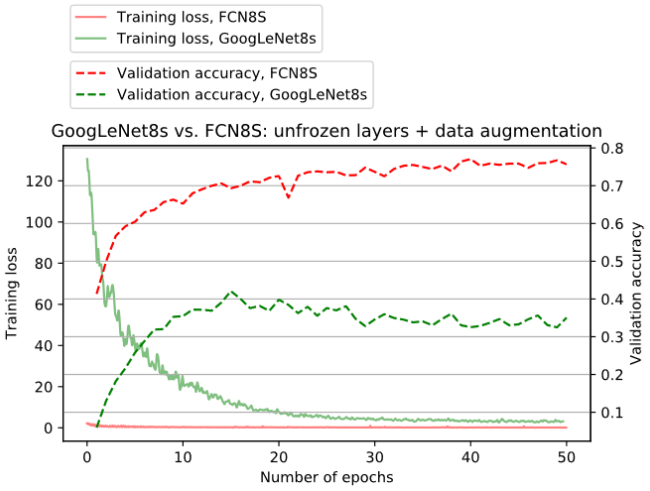


Fig. 4. Performance of our GoogLeNet

paper [2]. The architecture of this picture can be seen in the figure 8 and the comparison samples can be seen in the figure **??**.

D. Using a smaller model

Although we were able to get a very good accuracy in segmentation however, the model size was way too big for the Raspberry Pi which we were using to run the models on (521 MB).

At first when we started to use the Raspberry Pi we tried to setup the environment for our model to run. Since our model was designed on *PyTorch*, we tried to setup the same environment on the Raspberry Pi. However, PyTorch currently did no have much support and we had to build the entire code for Raspberry Pi from source. Afterwards we tried to run our model but it was way too big for it and it took well over 2 minute to output the results.Therefore, we had no option other than to reduce our model size. There were two ways we could do that and in this section the first way will be explained.

As it was shown earlier in the Table 1, that the size of the VGG was rather large and it was expected that it might not fit on Raspberry Pi. Therefore in comparison the second best accuracy is achieved by GoogLeNet. Which has a rather small size. We then implemented our architecture by modifying the architecture of GoogLeNet. The performance of our GoogLeNet is shown in Figure 4. The architecture for GoogLeNet we used can be seen in the figure 9.

With this architecture we reached an accuracy of 34.06% and an IoU of 0.32 however, the size was rather small (86.4 MB) which we were able to fit on the Raspberry Pi with ease and this brought down the execution time tremendously.

E. Model reduction of FCN8s VGG16

Although we were able to run this smaller model on the Raspberry Pi but the accuracy of the model was not high. There we explored further techniques to increase the

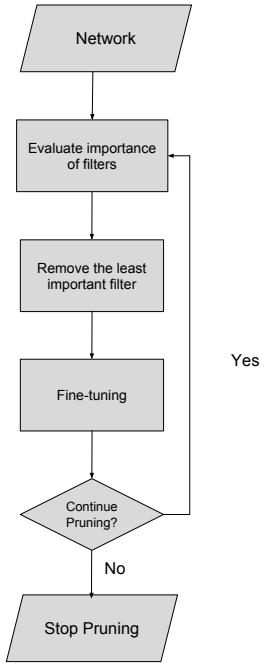


Fig. 5. Iterative filter pruning and re-training flow with 500 filters pruned per iteration.

accuracy of the model. Therefore, we ended taking the route of filter pruning our VGG FCN8s model (our best model yet). Although it had a higher accuracy compared to GoogLeNet but the size was bigger.

Inherently as of now, PyTorch does not support pruning out of the box. We had to resort to other techniques for model reduction. We ended up find a library which used this paper [7] as a technique of pruning the models presented to it.

The method that we use for filter pruning involves ranking the filters using the l_1 norm $\sum |F_{i,j}|$. We then remove 500 filters with the lowest score and retrain our model for 10

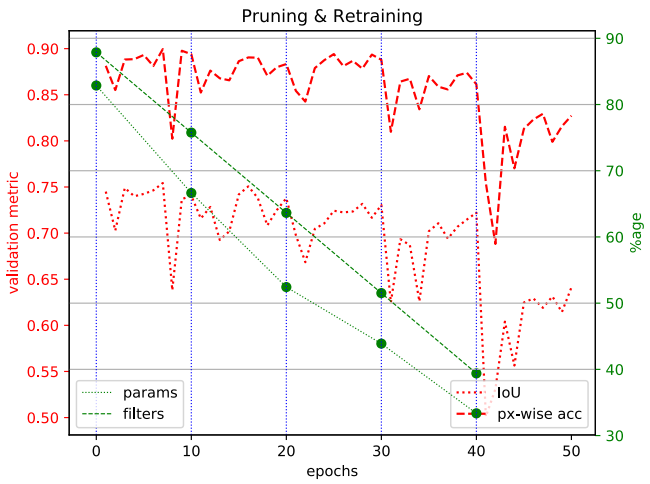


Fig. 6. FCN8s validation pixel-wise accuracy and IoU during 75% iterative filter pruning (500 filters pruned per iteration) and retraining for 10 epochs.

epochs for fine-tuning purposes. We repeat this process until we have pruned 75% of the filters.

This technique then gives a comparatively better model compared to GoogLeNet in terms of accuracy and a comparable size. This process of filter pruning is shown in the graph in figure 6. This gave us a model of a pixel-wise accuracy of *78.1%* and an IoU of *0.653*. With the pruned model we were able to reach a size of roughly 152 MB. Which although bigger than GoogLeNet but had a better accuracy and comparable execution time on the Raspberry Pi.

V. Unexpected Events

When we started the project we had a subscription of Microsoft Azure which we were using to develop and train our network. In the middle of the project the subscription to the cloud computer for training expired and we then had to resort to alternatives. We ended up getting a student account for Amazon Web Services and Google Cloud Platform, but we later found out that they did not allow us to use *graphics processing units* and rendered those accounts to be useless.

As an alternative we set up a remote desktop connection to a personal computer which had a *nVidia Titan* GPU,where we could possibly run our trainings. However we found a collaboration platform by Google which used GPUs for training but it had to be set up (*https://colab.research.google.com/*). The problem was that out of the box it did not support adding huge model and data files to online notebook directory. We therefore had to link the platform to our github repository and get the data from our Google Drive. This then worked in helping us train our network. There was one catch though, we had to stay connected to the network and if we were idle for too long once the training was done, our data would be lost. Constant availability comes at a cost therefore we decided to stick to this collaboration platform by Google. However, with the help of this platform we were able to divide workload a lot better.

Using PyTorch has various advantages such as the fact that it helps you in a modular workflow and debugging of the code is a lot easier but its recent arrival in the market means that there are not a lot of in-built features which we could use. Such as running it on Rasberry Pi is not as straight forward as compared to TensorFlow or Keras. You cannot use one command as you use generally to install pytorch or any other package. Instead of that, you have to build it from the source repository and it takes around 3-4 hours to do that. We have to install it several times because our Pi was unable to run large model due to memory allocation error with the default configuration. For that purpose, we changed the build files and then installed it. Furthermore, there were not any out-of-the-box solutions available for model reduction which was necessary for us to run it properly on our Raspberry Pi and so we had to thoroughly understand and re-factor a lot of model-reduction code to get it to work with our very large segmentation networks which was by no means trivial.

VI. Timing Comparison

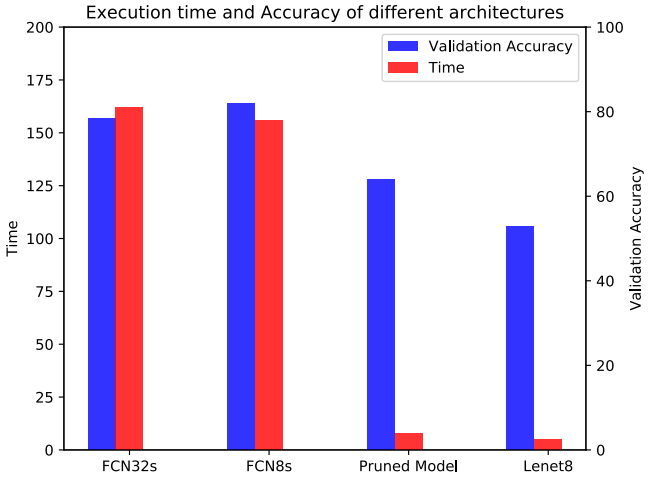


Fig. 7. Execution times of models on Raspberry Pi

An important part of the project is to highlight the execution times of the different models on the Raspberry Pi and how over the course of the semester we were able to decrease the execution times and increase the accuracy of the models. The timing comparisons are done and shown in figure 7. It is clear that the pruned VGG model had a small size and a therefore the execution time was comparable to GoogLeNet, but the accuracy of the model was higher that than the GoogLeNet model. Therefore, we were able to reach a good trade-off of execution time and accuracy of the model.

VII. Further Improvements

We were pressed for time since this was a semester project for a course. If had more time we would surely have been able to reduce the size further and increase the accuracy of the model via pruning. We could also have explored other

compression techniques such as quantization of the model. Although these were good to have but all of this requried more time and were add-ons. We believe we have sufficiently fulfilled the deliverables of the project course.

VIII. Project Performance

In comparison to what we had originally planned, we ended up going way beyond that. We thought that the model created using VGG16 would not be that slow on the Raspberry Pi. As it turned out that this turned out to be quite a challenge. And our route taken in minimizing the execution time on the embedded system and maximizing the accuracy has already been explained in the section IV. Everything that was delivered in the planned time frame. In terms of the original budget requirements we did not exceed them and were able to complete the entire project in initially planned budget.

References

[1] A. Krizhevsky et al., "ImageNet Classification with Deep Convolutional Networks," 2012.

[2] J. Long, E. Shelhamer, T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," *CVPR*, 2015.

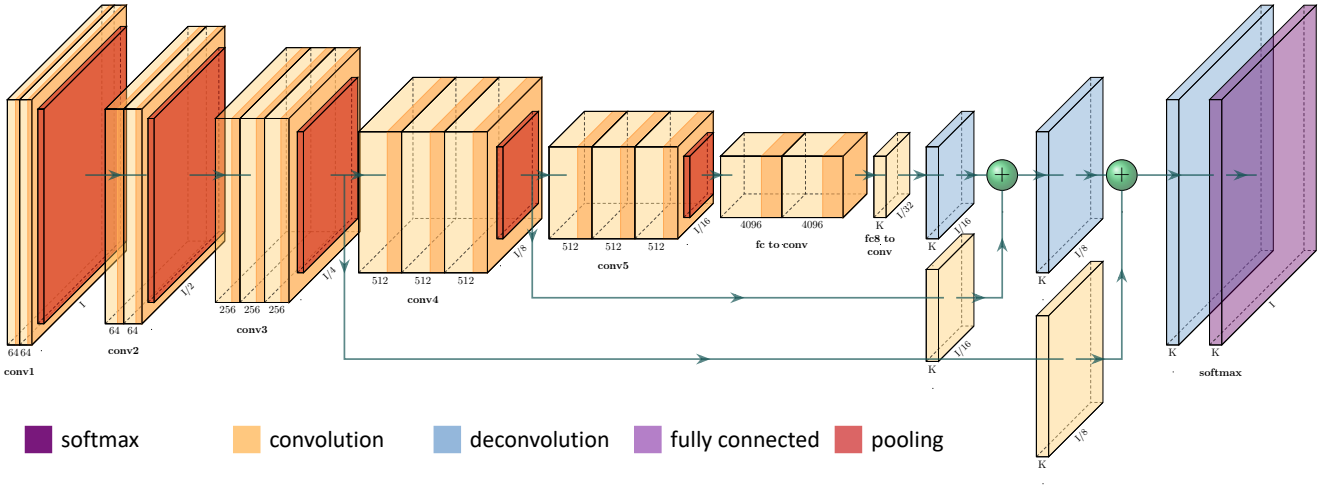
[3] M. Thomas, "A Survey of Semantic Segmentation," 2016 — arxiv.org/abs/1602.06541 .

[4] P. Pinheiro et al., "Recurrent convolutional networks for scene labeling" *ICML*, 2014.

[5] B. Hariharan et al., "Simultaneous detection and segmentation," *ECCV*, 2014.

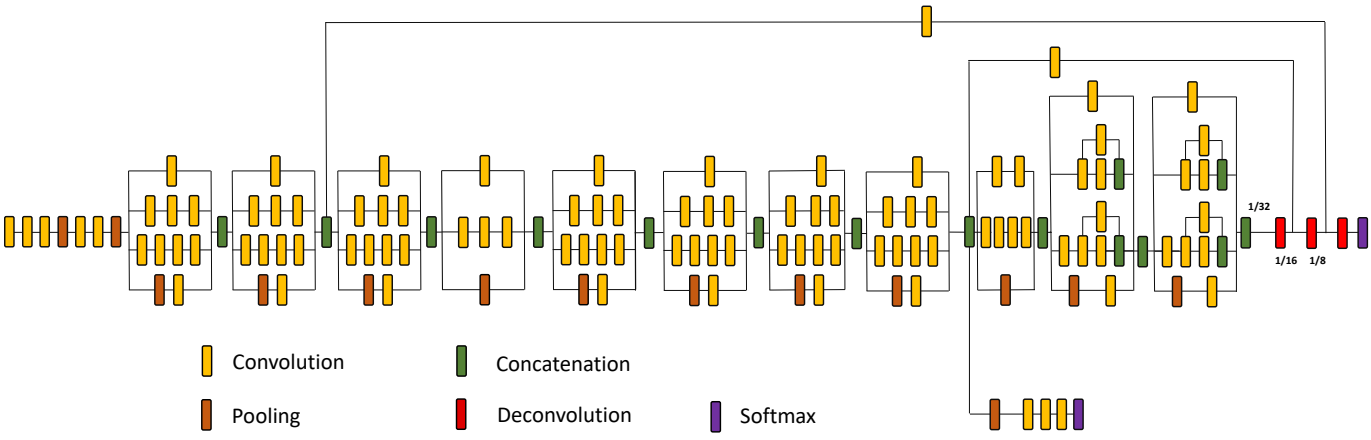
[6] C. Farabet et al., "Learning hierarchical features for scene labeling," *TPAMI*, 2013.

[7] H. Li et al., "Pruning filters for efficient convnets," 2016 — arXiv.org/abs/1608.08710 .



SOURCE: https://github.com/HarisIqbal88/PlotNeuralNet

Fig. 8. Modified VGG16 architecture (now referred to as FCN8s).



SOURCE: https://cloud.google.com/tpu/docs/inception-v3-advanced

Fig. 9. Modified GoogLeNet architecture (now referred to as LeNet8s or GoogLeNet8s).

Fig. 10. Selected samples. From left to right: image, ground-truth label, FCN8s VGG output (best model), 75% pruned FCN8s VGG output.