Foundation of Deep Learning

Maria Vakalopoulou
Stergios Christodoulidis

Segment features around residential buildings in UAV images of flooded area taken in Houston after Hurricane Harvey

Jean-Marc Dossou-Yovo Patrick Leyendecker Edouard Vilain

OVERVIEW OF THE REPORT

In the following report we are going to describe and analyse the solution we have designed and implemented to solve the provided problem in the Kaggle competition. We will assess the performance of our model and compare it to the benchmark solution of the U-Net. After, we will provide our insights as well as potential improvements that could lead to even more accurate results.

DESCRIPTION OF THE PROJECT

The fields in which we can use Deep Learning methods are vastly expanding due to technical advancements and a plethora of available data sources with large datasets, as we have seen during the Fundamentals of Deep Learning lectures. This has been proven especially powerful when working with image data and computer vision, coming to the point that Deep Learning methods have overtaken humans in terms of object detection, classification, localization and segmentation of single or multiple objects.

Especially the latter, image segmentation, has developed vastly over the last years, coming to use in Google portrait mode and Youtube stories as well as computer vision for self-driving cars. The method consists of classifying each pixel of an image as belonging to a certain class, thus making it a pixel-wise classification problem. In the scope of the Group Project we conducted a semantic segmentation on a dataset of images acquired by a small UAV in a rural area in Houston, Texas taken after Hurricane Harvey hit in order to assess the damages on public and private property. There are 25 categories of segments (see Table 1. in Appendix), that the pixels are to be classified into, making it a multivariate problem with each pixel belonging to a particular label. As it is a semantic segmentation problem, we do not differ between different instances of the same object.

Using a Convolutional Network (CNN) and supervised training using the training dataset containing images with pixel-wise labels we developed a model that can automatically segment pixels in the image in the testing dataset. The CNN consists of convolutional, pooling, up- and down-sampling (also known as encoder and decoder) and fully connected layers and uses shortcut connections (also known as gradient highways) of the U-Net architecture to address the loss of information when down-sampling and up-sampling an image. The goal was to achieve a higher score than the baseline model (see Figure 1. in Appendix) using the prescribed evaluation method of the Dice coefficient in the description of the Kaggle competition (see Figure 2. in Appendix).

In the following, we will describe the design of the solution as well as the process of implementation of the solution.

DESIGN OF THE SOLUTION

As part of our pre-processing, we decided to do data augmentation and create synthetic data by creating slightly modified copies of the existing training set. These methods tend to significantly improve segmentation models' performances and were perfectly suited to our case. Moreover, some classes (e.g. Trampolines) are sub-represented in our dataset due to their relative absence in the landscape. Naturally, our model will tend to omit these classes from prediction, unless they are enhanced using augmentation methods.

In terms of the architecture of the model, we chose to design a simplified version of the one found in the original U-Net paper (see Figure 1. in Appendix). We decided to make some alterations: By keeping an encoder of depth 5 and not reducing the amount of downsampling layers, but vastly reducing the amount of feature maps at each stage, we aimed to reduce computational time and capacity. Especially considering the computational limitations imposed by Google Colab. Instead of going from 64 to 128 to 256 to 512 to 1024 feature maps in the encoder stages, we used 8, 16, 32, 64 and 128 feature maps. Indeed, based on trial and error, we aimed to reach the best compromise between maximising the input image size and maximising the model's complexity while keeping within the bounds of Colab's computational resources. As a matter of fact, models were trained with more feature maps (multiplied by 2 at each depth) but performance was significantly worse as the model generalised poorly to validation data.

IMPLEMENTATION OF THE SOLUTION

In the first step of the implementation we install and import all of the necessary libraries. These include mostly popular Python as well as Pytorch Libraries. We also mount the Google Drive, where we uploaded the Training and Testing Set and planned to save our solutions.

1. Data Augmentation

After discovering the small size of the training set of solely 261 images and masks, we next start the process of Data Augmentation as we mentioned in the previous chapter. Here we create copies of the images in the training set and transform them. To do so, we first create the function `data_augmentation` that augments data for every label to reach a minimum number of 70000 pixels containing every label for a format of 1024 x 1024 for each image. Therefore we randomly select an image and apply randomly one of the transformations. These consist of random cropping, random rotating and centre cropping using the previously defined `Largest_rotated_rect` function. Finally we implement methods to preserve oversampling or undersampling for certain labels.

The paths for the final datasets are saved as train_images_aug and train_masks_aug and the `data_augmentation` function is applied, iterating through all images of the training set. After augmentation, our training set contains 696 images.

2. Data Preparation

In the next step we performed data pre-processing to prepare the data for the following training of the model.

First, we implemented the `SegmentationDataset` class, a subclass of PyTorch's `Dataset` structure. Not only does it load data from the path, but also applies the necessary transformations to fit our model's expected input format. Transformations applied are relatively simple. We first resize all images to squares of size (512,512). This was decided after several attempts at training models with images of size (1024,1024) which were less satisfying for two main reasons: it took more memory space thus gave less opportunities in model complexity; having to resize output images from (512,512) to (1024,1024) (because Kaggle evaluation are realised with the latter) offers more regular final masks. We then applied min-max normalisation across our dataset. Being a good practice in general for Deep Learning, we made sure it was

done in our case though we found out it produced no significant improvements in model training time and performance.

Second, we implemented the function `train_tune_dataset` to split the dataset into training and tuning sets using a 90% - 10 % split at first. This enabled us to evaluate our model on sets large enough to truly evaluate model performance. Once our model's parameters were set, we vastly reduced our tune set size to around 3% of the whole dataset (21 images). This enabled us to train our model on the largest part of our data while keeping an estimation of its performance on new images. We made sure that within this tune set, all classes were represented so that evaluation was the least possibly biased.

3. Building of the Model

After setting up the datasets, we started implementing the architecture of the model. As we mentioned in the previous chapter, we planned to build an U-Net model and began by determining in the definition of the class `UNet` (as a subclass of pytroch's nn.Module class). Each instance is defined by 3 parameters: the number of input channels (3 in our case), the feature maps of the first layer (8) and the output channels for the classes (25) (see Table 1. in Appendix).

As mentioned in the design of the solution, we used convolutions for downsampling rather than max-pooling and transpose convolutions for up-sampling instead of simple up-convolution. This is justified by the symmetry of the U-Net architecture which requires our down-sampling and up-sampling operations to be analogous. Papers have shown that performance is optimised when respecting the symmetry of up- and down-sampling. Thus, we implemented the 22 Layers for the Encoder, followed by the 22 Layers for the Decoder. These consist of 9 convolutional layers, 9 batchnorm layers and 4 down- and upsampling layers with a pattern of two convolutional and batchnorm layers alternating followed by a single down-/upsampling layer to build a module. Padding was used at each convolutional layer in order to preserve input shape for the output.

For the output we used one additional convolutional layer and implemented the Softmax function as our non-linear (softmax) activation function. The final output is a tensor of size (25,512,512) each slice corresponding to a probability vector indicating the belonging of each pixel to the corresponding class.

After having defined the `UNet` architecture, we also defined our forward pass function `forward`, applying ReLu activation after the convolutional modules, 10 times in the Encoding and 8 times in the Decoding layer.

Finally, we reach 46 Layers in total, with 112 Parameters in the beginning adding up 128.081 parameters in total (see Table 2. in Appendix).

4. Training of the Model

After having set up the model, we start the training process using the augmented dataset for training. We define our datasets using `SegmentationDataset`, apply the training and tuning dataset split and create the DataLoaders. Next, we instantiate the U-Net model and the training parameters. Note that all models were trained on Google Colab's GPUs which were necessary to train such large models. We used Adagrad Gradient Descent as our optimizer and Cross Entropy as our loss function. We improved the learning rate of the baseline model to 0,01 and trained during 150 epochs.

Finally, we start the training loop to iterate through all batches of images in the training set and after the 150 epochs we save the best model to our Google Drive. Model selection was based on the evaluation of our model on the tune (or validation) set based on a Dice Score criteria. Indeed, as Dice Score is the evaluation method of the Kaggle competition, we implemented it as a subclass of pytorch's nn.Module class by defining the function `DiceScore` and used it as our model's selection criteria. Therefore, our final model is the model that best performed on our tune set in terms of Dice Score.

We also implemented early stop callbacks to our training loop. This consists of stopping training early if the model does not improve for a defined number of epochs (called patience). This felt useful as training was quite time consuming, each epoch takes around 2 and a half minutes to more than 4 minutes for our final model. Therefore, training would require anywhere from 2 to 9 hours and was optimised using our early stop callbacks. In reality, we abandoned its use figuring that in our specific case, training could reach plateaus before improving significantly again.

To increase the performance of our model, we tried to perform the task with a model architecture using the pre-trained Resnet50 from Pytorch as a backbone. We replaced the encoding part with this new architecture and trained the model again on the dataset. We had better results with a faster convergence (a Dice score of 68 with only

24 epochs) but the model is heavy to train and due to lack of computational performance we could not push this attempt further.

By using the Matplotlib Python package we plot the loss and accuracy curves, the train and validation loss and accuracy across epochs to get an overview of the model performance. When observing the accuracy curve for instance (see Figure 3. in Appendix), we find that training improves accuracy on the tune set during the first 20 epochs, we then reach a plateau of around 20 epochs during which accuracy is not increased. Using our early stop callback, this would have stopped training and returned the best model (found around epoch 15). This effect isn't expected as the model once again improves significantly after epoch 40.

5. *Predictions based on the Model*

In the final step we implemented the `TestDataset` function for the same purpose as the previous `SegmentationDataset` function to adjust the input data to be matching to our model. We selected the model that achieved the highest dice score and used it on the testing dataset.

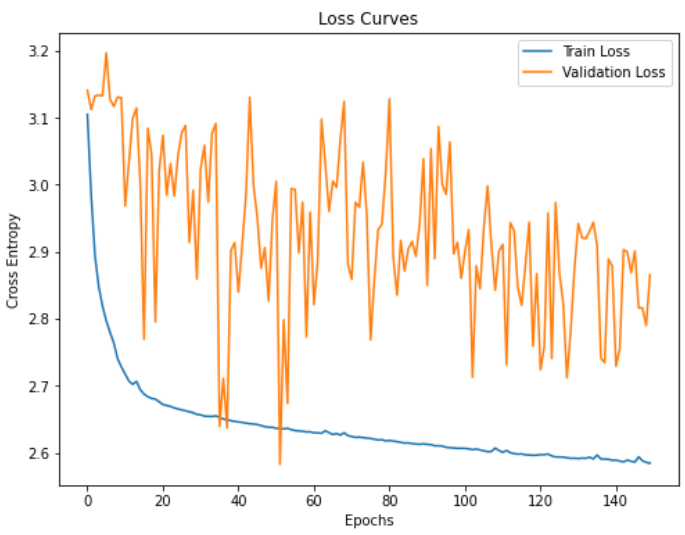
The output of our model gave a folder including the predicted images in PNG format. To submit on Kaggle we used the `prepare_submission.py` file that created a CSV as the final output consisting of the pixel of each image including the label of the classes the pixel belonged to.

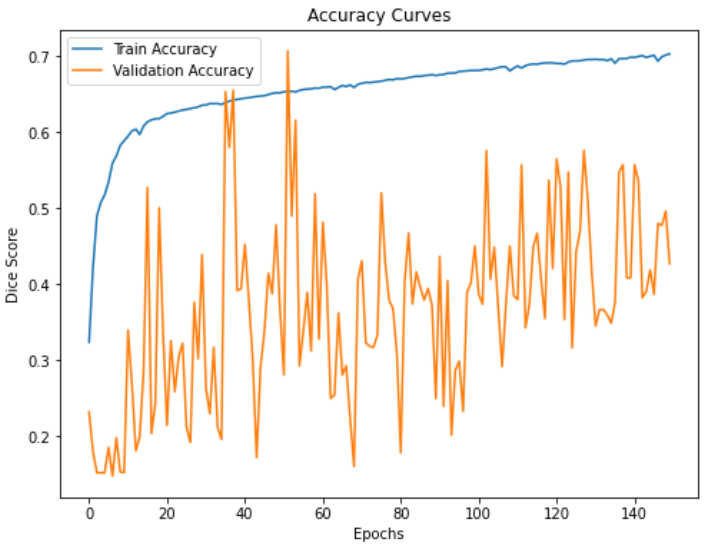
OUTLOOK

As a final result we managed to achieve an accuracy ranking of 0,69594, leading to a placement of 18 out of the 39 participating teams and the baseline model. Nevertheless, we managed to rank higher than the baseline model by adjusting some of the hyperparameters and conducting data augmentation.

We tried various further approaches to increase the accuracy of the result, such as the implementation of a Resnet Model with 24 Epochs, reviewing the saving function using Open Cross Validation, normalising the input images, increasing the number of feature maps and more. Nevertheless, plenty of our efforts on improving our performance were not reflected by our achieved dice score and we achieved the highest score with the described implemented solution.

The following figures correspond to classical training and accuracy curves obtained during training. Although train loss and accuracy improve over epochs, it is quite surprising to see the evolution of the model's performance on the validation set. Indeed, performance generalises poorly to unseen images and could influence a poor choice of final model. We are still to figure out a technique that would tackle this issue.





There are different additional steps we could have chosen in attempting to further increase our performance. These include building additional models that only focus on certain regions of the images through regional gridding and combine their performance in multiple pipelines in combination with deeper, wider, more computationally heavy models.

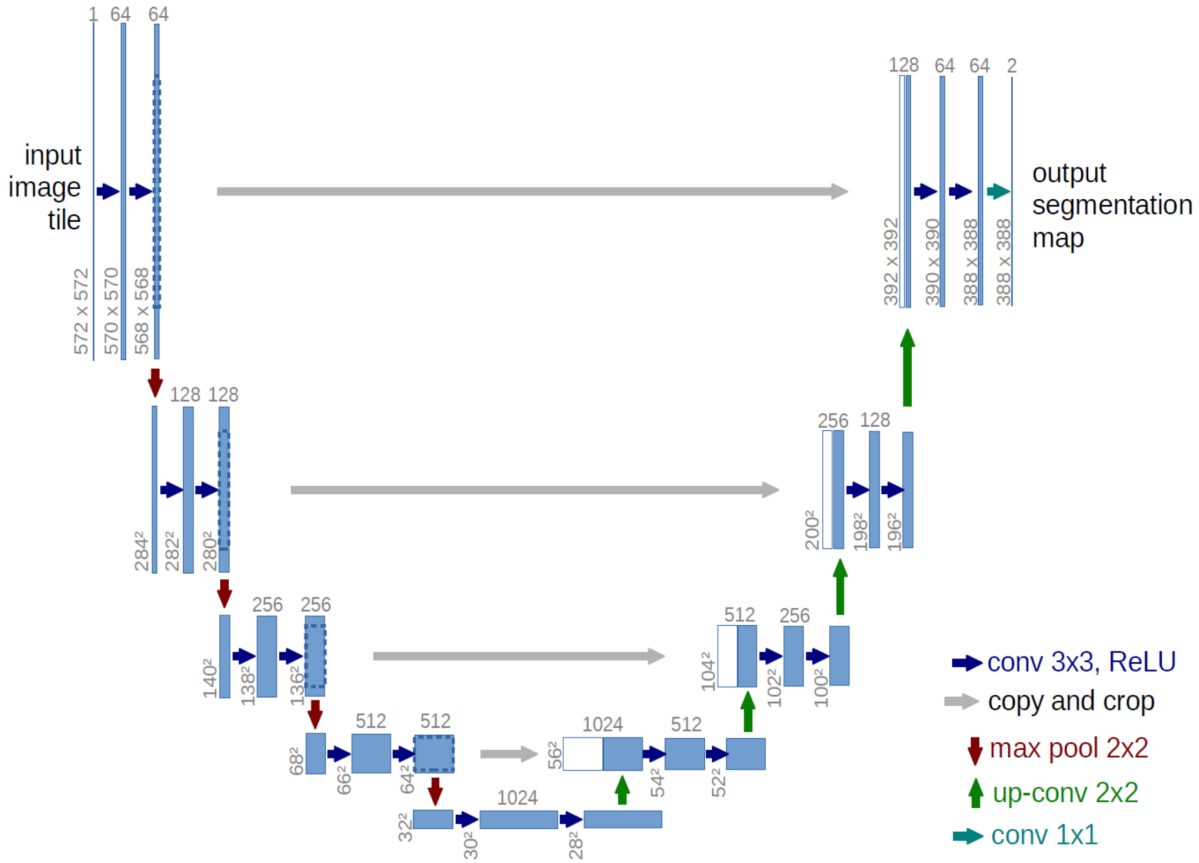
APPENDIX

Table 1.:

List of the present classes:

0:	Background
1:	Property Roof
2:	Secondary Structure
3:	Swimming Pool
4:	Vehicle
5:	Grass
6:	Trees / Shrubs
7:	Solar Panels
8:	Chimney
9:	Street Light
10:	Window
11:	Satellite Antenna
12:	Garbage Bins
13:	Trampoline
14:	Road/Highway
15:	Under Construction / In Progress Status
16:	Power Lines & Cables
17:	Water Tank / Oil Tank
18:	Parking Area - Commercial
19:	Sports Complex / Arena
20:	Industrial Site
21:	Dense Vegetation / Forest
22:	Water Body
23:	Flooded
24:	Boat

Figure 1. - https://arxiv.org/pdf/1505.04597.pdf



- Baseline U-Net
  - Resize to 1024x1024
  - Crop 512x512 patches
  - Batch size:8
  - Epochs: 100
  - cross entropy
  - SDG
  - lr=0.1
- Score 0.62310

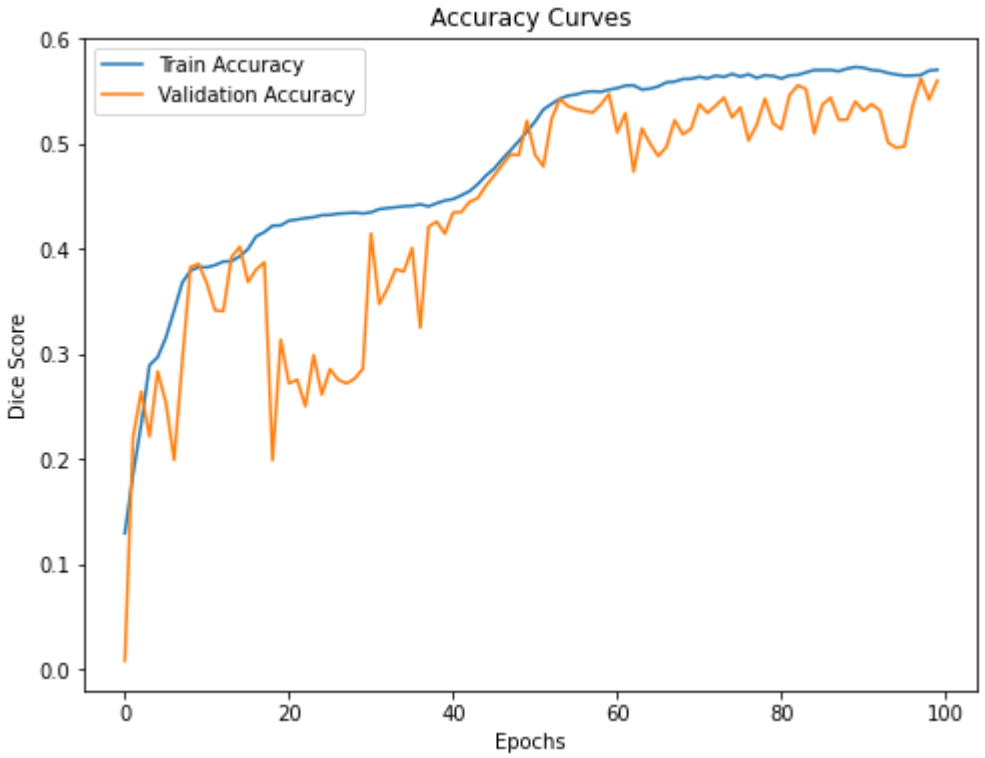
Figure 2.:

$$Dice = \frac{2 \times TP}{(TP + FP) + (TP + FN)}$$

Table 2.:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1        [-1, 4, 1024, 1024]             112
       BatchNorm2d-2        [-1, 4, 1024, 1024]               8
            Conv2d-3        [-1, 4, 1024, 1024]             148
       BatchNorm2d-4        [-1, 4, 1024, 1024]               8
            Conv2d-5          [-1, 4, 512, 512]              68
            Conv2d-6          [-1, 8, 512, 512]             296
       BatchNorm2d-7          [-1, 8, 512, 512]              16
            Conv2d-8          [-1, 8, 512, 512]             584
       BatchNorm2d-9          [-1, 8, 512, 512]              16
           Conv2d-10          [-1, 8, 256, 256]             264
           Conv2d-11         [-1, 16, 256, 256]           1,168
      BatchNorm2d-12         [-1, 16, 256, 256]              32
           Conv2d-13         [-1, 16, 256, 256]           2,320
      BatchNorm2d-14         [-1, 16, 256, 256]              32
           Conv2d-15         [-1, 16, 128, 128]           1,040
           Conv2d-16         [-1, 32, 128, 128]           4,640
      BatchNorm2d-17         [-1, 32, 128, 128]              64
           Conv2d-18         [-1, 32, 128, 128]           9,248
      BatchNorm2d-19         [-1, 32, 128, 128]              64
           Conv2d-20           [-1, 32, 64, 64]           4,128
           Conv2d-21           [-1, 64, 64, 64]          18,496
      BatchNorm2d-22           [-1, 64, 64, 64]             128
           Conv2d-23           [-1, 64, 64, 64]          36,928
      BatchNorm2d-24           [-1, 64, 64, 64]             128
  ConvTranspose2d-25         [-1, 32, 128, 128]           8,224
           Conv2d-26         [-1, 32, 128, 128]          18,464
      BatchNorm2d-27         [-1, 32, 128, 128]              64
           Conv2d-28         [-1, 32, 128, 128]           9,248
      BatchNorm2d-29         [-1, 32, 128, 128]              64
  ConvTranspose2d-30         [-1, 16, 256, 256]           2,064
           Conv2d-31         [-1, 16, 256, 256]           4,624
      BatchNorm2d-32         [-1, 16, 256, 256]              32
           Conv2d-33         [-1, 16, 256, 256]           2,320
      BatchNorm2d-34         [-1, 16, 256, 256]              32
  ConvTranspose2d-35          [-1, 8, 512, 512]             520
           Conv2d-36          [-1, 8, 512, 512]           1,160
      BatchNorm2d-37          [-1, 8, 512, 512]              16
           Conv2d-38          [-1, 8, 512, 512]             584
      BatchNorm2d-39          [-1, 8, 512, 512]              16
  ConvTranspose2d-40        [-1, 4, 1024, 1024]             132
           Conv2d-41        [-1, 4, 1024, 1024]             292
      BatchNorm2d-42        [-1, 4, 1024, 1024]               8
           Conv2d-43        [-1, 4, 1024, 1024]             148
      BatchNorm2d-44        [-1, 4, 1024, 1024]               8
           Conv2d-45       [-1, 25, 1024, 1024]             125
          Softmax-46       [-1, 25, 1024, 1024]               0
================================================================
Total params: 128,081
Trainable params: 128,081
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 12.00
Forward/backward pass size (MB): 963.00
Params size (MB): 0.49
Estimated Total Size (MB): 975.49
----------------------------------------------------------------
```

Figure 3.:


Accuracy Curves
Train Accuracy
Validation Accuracy
Dice Score
Epochs
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
20
40
60
80
100