**Satellite Road Extraction**

*Installation:*

* python 3.6
* pip install -r requirements\_cpu.txt.

*Preprocessing and data cleaning:*

* The original training data had a few input images for which the corresponding labels were missing and vice-versa, on removing such cases we were left with 805 (1500\*1500) images.
* Each of these images were broken into 4 and a final data set of 3216 images (750\*750) were obtained.
* Advantages of Splitting images into smaller images:

1. Faster Training and Inference.
2. Since the augmentations over a batch are done during run time, this enables us to have greater number of batches and hence our model would see a larger variety of data. (Original image split into four would have different random augmentation applied to each part.)

* Out of the 3216 training images around 195 images had the input satellite data missing (white patches) but the labels were available. Although an entirely white/black patch is harmless for the model if there is no label corresponding to it but since we had labels corresponding to missing data, such images were removed and two such examples are shared below.

A close up of a logo

Description automatically generated

A close up of a logo

Description automatically generated

* After removing such cases 3201 images were left for training.
* All the original 13 test image and label pairs were good. So, splitting each into 4 the performance of model would be evaluated on 52 image and label pairs.

*Training:*

* Out of the 3021 images 303 images (10%) were used for validation and finally the model was trained on 2718 images.
* The training script uses tensorflow 2.0 and tf-keras.
* The input pipeline is setup using tensorflow Dataset API which compatible with tensorflow core and is much faster than commonly used keras ImageDataGenerator.
* Augmentations:

1. Horizontal and vertical flips.
2. Height and width shift, which improves detections at image boundaries.
3. Random hue, delta and saturation.
4. Brightness adjustments, which is very effective in case of overhead imagery due to variations in time when an image is captured and shadows.

* Implemented stochastic weight averaging of multiple weight snapshot saved using cyclic learning rate(ref. [**https://arxiv.org/pdf/1803.05407.pdf**](https://arxiv.org/pdf/1803.05407.pdf)). Details are discussed in callbacks section that follows.
* **Model Architecture:** UNet with efficientnetb4 as backbone is used. Efficient nets are state of the art performing models currently.
* Another Model UNetPlusPlus has also been trained and comparison of results is below.
* **Loss function:** Dice loss (combination of binary cross entropy and dice coefficient) is used.
* The model has been trained for 50 epochs and at the end of 5th, 13th, 25th and 42nd epochs a weight snapshot is saved. This would be clearly explained in call backs section that follows.

*Callbacks:*

* **ModelCheckpoint**: Saves model best weights based on validation dice loss but we will be using snapshots saved using cyclic learning rate since there is no early stopping used, weights saved by model check point might overfit.
* **Custom callback to save weights for** **SWA**: This is added a call back and the code is available in cb package of the project. The cyclic learning rate decaying along a cosine helps in faster convergence and at the end of each cycle model weights corresponding to a separate optimum is saved and based on the performance of these snapshots on validation set weight averaging of different snap shots is done which helps in finding a broader optimum. The figure below shows how dice loss and IOU score vary in sync with cyclic learning rate and every time the cycle ends weights are saved. These results are on validation set and confirm the effectiveness of saving weights in this manner. All these are available in logs folder and can be viewed by running Tensorboard.

*A picture containing text, map, person, lot

Description automatically generated*

* **TensorboardImage**: Images can be logged at end of each epoch to see how exactly the model is performing while training. And If you carefully see the prediction on right in the below image is actually better than ground truth in some areas.

*A picture containing game, basketball

Description automatically generated*

*Results and Evaluation:*

* Below results are on test set of 52 images (originally 13) which the model has never seen nor validated using these images.
* Evaluation is done on all weights and their combinations and the best model can be selected.
* SWA\_a\_b means weight averaging of models saved at epochs a and b respectively.
* The ones highlighted in red performed the best on test set.
* The one highlighted in green had highest precision of 88%.
* Weight averaging appears to increase precision and in general it generalizes well over test data even if it has a different distribution.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| UEfficientNet | mean\_iou | mean\_precision | mean\_recall | mean\_accuracy |
| Weights at epoch 5 | 0.599 | 0.8527 | 0.669 | 0.964 |
| Weights at epoch 13 | 0.602 | 0.8570 | 0.669 | 0.9649 |
| Weights at epoch 25 | 0.605 | 0.8606 | 0.672 | 0.9653 |
| Weights at epoch 43 | 0.608 | 0.8644 | 0.672 | 0.9658 |
| SWA\_5\_13\_25 | 0.56 | 0.88 | 0.61 | 0.9633 |
| SWA\_25\_43 | 0.6107 | 0.858 | 0.688 | 0.9656 |

* Comparison of results UEfficientNet and UnetPlusPlus:
* The comparison is done at same epoch, due to time constraint I was able to train UnetPlusPlus till % epochs only.
* The results show that UEfficientNet is better of the two Architectures in this scenario and certainly converges faster.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | mean\_iou | mean\_precision | mean\_recall | mean\_accuracy |
| UEfficientNet | 0.599 | 0.8527 | 0.669 | 0.964 |
| UNetPlusPlus | 0.562 | 0.8390 | 0.6324 | 0.9604 |

*Geometry Object Extraction from Masks:*

* I have worked on geometry extraction previously. If given the geographical bounds of these images we can extract the geojson Feature collection from the masks and integrate it in the map data.
* Following are the sample results on some masks and output is the Shapely Multiline string object.
* It shapely object (Multiline String) output is a reflection of original in the x axis and is rendered in jupyter notebook.

A close up of a map

Description automatically generatedA close up of text on a white background

Description automatically generated

*Further Improvements:*

* More experiments cab be done using other backbones that Efficient net offers.
* Network Architectures such as D-link net Unet with Resnet backbones are also worth trying.
* The original image can be broken into even smaller parts (9 images of 500\*500 each) which would add more variety with augmentations but when in production we might need to do inference a lot more times and there can be service boundary problems, such as we try to extract the geometry from these we will need to make a lot more associations among adjacent tiles after extracting geometry from each of them.
* Also, if we break into very small tiles then we might often start missing context around the roads which fall at the edges of roads, which might lead to bad results.
* Instead of directly ignoring 195 images, we could have seen if there is a label that actually correspond to the missing data part. And if there is no label corresponding to that missing data part, such images could have been retained and we could have got 20 to 30 more images for training. For example, the image below can be retained but I realized this when I had already started training.

**

* Augmentations like grid distortion and shear transforms are also worth trying and could have been used.
* Binary focal loss can be used in combination with dice loss. Binary focal loss focuses mainly on features that are hard to detect and are overwhelmed by the domination of loss function value by easily classified examples.(ref. <https://arxiv.org/pdf/1708.02002.pdf>)