Transfer learning on the Planet dataset

Lidia Luque, username lidilu

March 9, 2022

1 Introduction

The aim of this project was to use an off-the-shelf trained model (ResNet-18 trained on ImageNet), modify it to accommodate different input data, and train the model using the trained weights as starting weights. This is called transfer-learning and is commonly used to help the training to converge faster and/or to get more robust models when only limited training data is available.

2 Dataset

The dataset used in this project is the dataset curated by Planet which consists of 40479 .tif images of spatial dimensions 256x256 pixels. Each image consists of four channels, where the first three are the RGB channels and the last one contains data from the near infrared spectrum. The images are crops of satellite images of the Amazon rainforest. There are 17 classes, and the label for each image includes at least one of the classes, but often several. The dataset was split into a validation dataset consisting of 33% of the images and a train dataset containing the rest. A seed of 0 was used to split the dataset. The same seed was used throughout the code.

3 Models

Three models were made, all with a ResNet-18 trained on ImageNet as a starting point (downloaded with `torchvision.models.resnet(pretrained=True)`).

- **single3**: The input to this model are the three RGB channels of the images in out dataset. For this, only the linear layer was modified so that the network has 17 outputs instead of 1000 in the output. This was done by overwriting the linear layer, named 'fc' in this ResNet-18 implementation, to `nn.Linear(512, 17)`. All other layers remained the same. The weights for this last layer were initialized automatically by torch.

- **single4**: The input to this model are all four image channels (RGB + near infrared). For this, appart from chaning the linear layer to have 17 outputs as explained above, the first convolutional layer in the network also had to be changed. This layer is called 'conv1' in this ResNet-18 implementation, as was changed to

  ```
  nn.Conv2d(4, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,3), bias=False)
  ```

  Note that this is simply the original layer with 4 instead of 3 input channels to the convolution. The weights for this last channel were initialized to values from the Kaiming normal distribution with `nn.init.kaiming_normal_(w)`

- **double4**: This model requires two inputs: a 3-channel RGB image and a 3-channel near infrared image. The later is simply the same infrared channel copied three times so as to get 3 channels. This model consists of two parallel models consisting of all layers of the ResNet-18 model except for the linear layer. One model processes the RGB image, while the other takes the 3-channel infrared image. The output feature maps are flattened and concatenated, increasing the length of the output to 1024. Thus, the last linear layer is replaced by `nn.Linear(1024, 17)`

4 Training and validation

All three models were trained for 35 epochs with the hyperparamenters provided, no further hyperparameter tuning was done. The hyperparameters are as follows:

- Learning rate set to 0.005
- Learning rate scheduler StepLR with step_size=10 and gamma=0.3
- Adam optimizer
- The same transforms for training and validation, consisting of random crops of size 224, random horizontal flips and normalization using $mean = [0.7476, 0.6534, 0.4757, 0.0960]$ and $std = [0.1677, 0.1828, 0.2137, 0.0284]$. After these transforms, the relevant channels according to the model were selected.
- A mini-batch size of 32 for training and 64 for validation

Binary cross entropy, defined as follows, was chosen as the loss:

$$BCE_c = \frac{1}{N}\sum_{n=0}^{N} y_{n,c} \log \sigma(\hat{y}_{n,c}) + (1 - y_{n,c}) \log(1 - \sigma(\hat{y}_{n,c})) \tag{1}$$

where c denotes a class, N is the number of samples over which we calculate the loss, $\hat{y}$ is the prediction output, y is the label and σ denotes the Sigmoid function. Since this is implemented in torch, in the code this loss was simply defined with `nn.BCEWithLogitsLoss(reduction = 'mean')`

The models were evaluated on the validation dataset at each epoch. The average precision score (AP) for each class over all validation samples was calculated using

```
sklearn.metrics.average_precision_score(labels, pred, average=None, pos_label=1)
```

where labels and predictions are numpy arrays of shape (N_{val}, 17) where N_{val} is the number of samples in the validation dataset.

The trained model at the best AP score averaged over all classes (mean AP, evaluated on the validation dataset) was saved.

5 Results

The *double4* model preformed best with a max mean AP score of 0.580 at epoch 21. The *single3* and *single4* model preformed slightly worse with a max mean AP score of respectively 0.568 and 0.569 at epochs 20 and 23. All results that follow are taken from the best preforming model, that is, the *double4* model saved at epoch 21. The train and test (validation) losses are shown in figure 1 together with the mean AP for each epoch. While the train loss steadily decreases, the test loss first decreases before starting to increase slightly after epoch 10. This is due to the bias-variance tradeoff and points to a degree of overfitting in our model (since we save it at epoch 21).

Figure 2 shows the accuracy of the predictions averaged over all classes at 20 thresholds, i.e. the accuracy when the decision threshold is set to t. The thresholds are linearly spaced from 0.5 to 1. When a class did not have predictions over the threshold, the tail accuracy for this class at that threshold was set to np.nan and not taken into account when calculating the mean. I am unsure as to why the tail accuracies are not monotonically increasing as it is my understanding they should be. It might have to do with how I chose the thresholds, which are simply linearly spaced from 0.5 to 1. While choosing the thresholds such that they separate a percentage of the validation data, as mentioned in the exercise text, would work for one class, in our case this would mean taking the mean of tail accuracies calculated with different thresholds, which seems strange.

Figure 3 shows the images with the highest predictions for the class 'primary' (predictions closest to 1). Figure 4 shows the images with the lowest predictions larger than 0.5 for the class 'primary' (predictions closest to 0.5). The number above each image indicates the image number. True positive

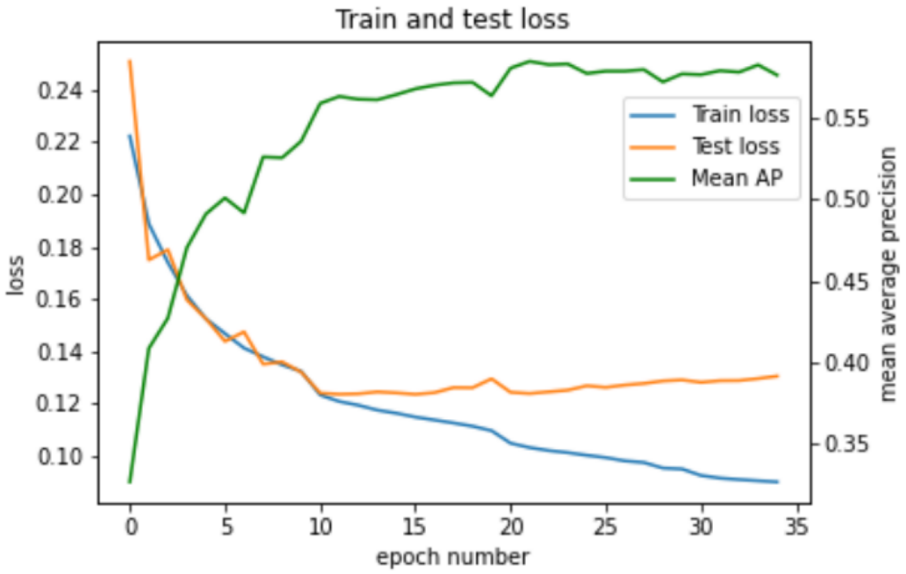

Figure 1: Train and test loss and test mean average precision score for the best preforming model, the model with two networks. The model was saved at epoch 21, which has the highest mean average precision score computed on the validation dataset.

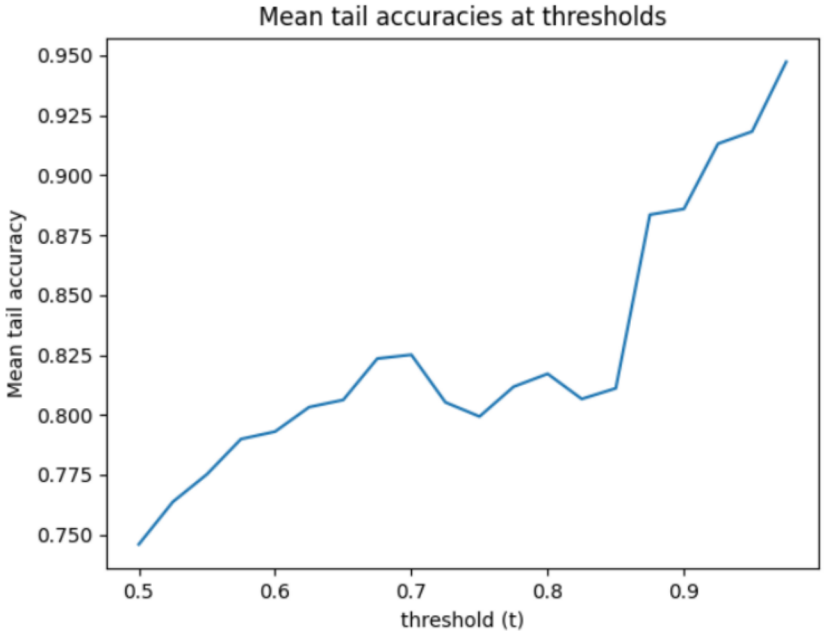

Figure 2: Mean tail accuracy computed at 20 values of t between 0.5 and 1.

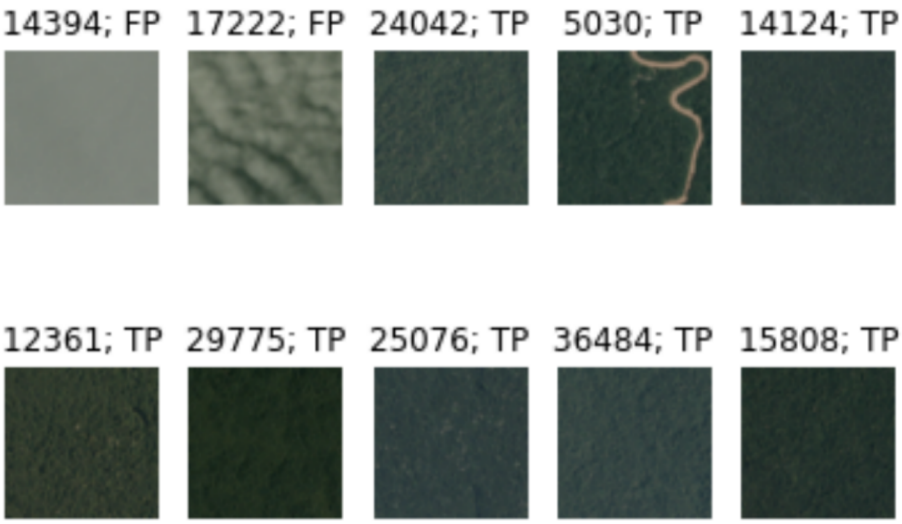

Figure 3: Images with the highest predictions for class 'primary'. This shows the images where the model's predictions were most certain for this class. The title for each image shows the image number and whether the prediction agrees with the label (TP) not (FP).

(TP) indicated that the image was also classifies as 'primary' in the label, while false positive (FP) indicated that the image was not classifies as 'primary' in the label. Figure 3 and 4 indicates that, at least to a certain extent, the output of the model can be interpreted as how certain the model is of the prediction, with outputs closer to 1 being more certain (Note: this is a very, very rough rule of the thumb and breaks down rather quickly...). Amongst the 10 images with outputs closer to 1 only 2 were misclassified, while amongst the 10 images with outputs closer to 0.5 8 were misclassified. We need to be aware of this when showing the outputs of any model: it's easy to cherry pick the best outputs, but this will give a very skewed impression on the performance of the model.

6 Using the code

train.py trains the chosen model. It has two user defined parameters, DATA_DIR which takes the path to the data (if the code is run in the ml-node, this path does not need to be changed) and MODEL, which takes a string defining the model to be trained: double4, single3 or single4. These are defined at the start of the script. Running the script as it is without changing any of the parameters will train the double4 network (as long as it is called from the ml-node at uio)

results.py has the same two parameters defined at the start of the script. It consists of different functions that can be called at the end of the script. Running it as it is will run the reproduction rutine.

- `save_predictions()` runs inference on all data in the validation dataset, calculates the mean AP scores and saves all predictions and mean AP scores.
- `reproduction_rutine()` prints out and returns the difference between the saved predictions and the predictions calculated on the fly.
- `test_train_loss_plot()` plots train and validation loss and validation mean AP for all epochs.
- `plot_tail_accuracies()` plots tail accuracies
- `show_top_bottom_images()` plots images with highest and lowest prediction scores for chosen class (default class 12, 'forestry')

OBS: training the model will save the new weights. Running results.py after retraining the model by running train.py will thus calculate the results on the newly trained model, which if only run for an epoch or two will give markedly worse results.

Figure 4: Images just above the threshold (taken to be 0.5) to predict the image to include class 'primary'. This shows the images where the model's predictions were most unsure for this class. The title for each image shows the image number and whether the prediction agrees with the label (TP) not (FP).