**Maman 12**

## **Q1**

### **Research/Training phase**

* I’ve split the training data into train/validation
* After extracting SIFT descriptors (128) I’ve tried reducing dimensionality for computing relief using PCA.
  + Total explained variance by 50 components: 0.86
  + Total explained variance by 80 components: 0.95
  + Total explained variance by 90 components: 0.97
* While this looks like a promising path to reduce computation, I’ve dropped it.
* Calculating SIFT features took some time. I’ve used the fact that cv2 doesn’t use pythons GIL and reading images is IO bound operation, to run in asyncio fashion to reduce run-time.
* I’ve trained kmeans over 20% of the training descriptors, with k=100 and n\_init = 4.
  + Increase # of descriptors to (maybe) get better means
  + Increase n\_init to (maybe) get better means
* After that, I’ve used my kmeans over the training data to produce a list of descriptor: cluster.
* Then I used histogram over all the descriptors clusters and normalized it.
* Why normalization is important here? Because some images might have 100 descriptors and other 1000. Normalization will get them to the same level. I’ve chosen l1 normalization here as I find it simply explaining the % a “feature” contributes. While there are many ways to normalize the data, I’ve decided not to dwell on it.
* Anyway, After this process, I’ve got / learnt procedures to transform a file\_path into a feature (1, 100) where sum(feature) = 1.
* I’ve used scikit learn encoder label to learn the labels and their transformations.
* Using all of the above, I now have train/validation features and labels.
* I’ve used xgboost classifier as its simply better than SVM both in results and computing speed.
* Xgboost hyper parameters were initially chosen by me from my own experience with the algorithm.
* The classifier (xgboost) outputs the roc auc (ovr) and mlogloss(multi) for both validation and training set during the training. This helps track for over/under fitting. Since xgboost builds many trees, I use an early stop parameter, to stop at the best iteration (to avoid overfitting on the training). The early stop uses the roc auc over the validation set to know when to stop.
* Eventually the training function yield clf, files\_to\_features, le
* Clf – the classifier (xgboost trained) – boosted trees (max depth = 6)
* Files\_to\_features (a delayed function, that uses the sift, semaphore and kmeans we’ve learnt/defined, and waiting for paths to execute)
* Le – label encoder, where we transform labels from strings to integers.

# **Testing phase**

At the beginning, we’ve split the data into Training and Testing sets. Later on we took this Training set and divided it into Training set and Validation set, and did all the work above.

Which means the testing set doesn’t know ANYTHING about what happened thus far.

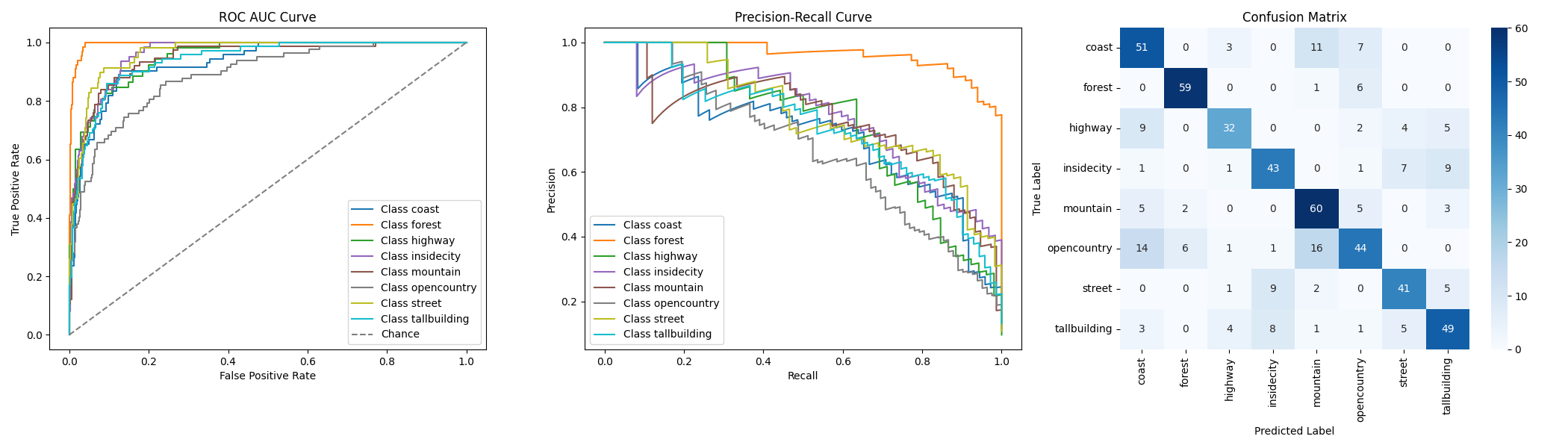
* All data was split using stratification over the label.
* We’ve used the files\_to\_features function to get features from the testing set
* We used labelencoder to encode testing labels
* we’ve then used the classifier over the testing data to get probabilities.
* We’ve ran ROC AUC One VS Rest and averaged it.
* Both macro and micro averages were roughly the same with 0.938~ ROC AUC which is f$cking awesome!
* **Those results surprised me as I was 100% sure the descriptors will not represent the images well enough. And additional features like colour histogram and histogram of gradients will be needed.**
* I even thought about increasing my feature space using the 2 ideas above. But once I saw the results are very high, I’ve let it go.
* I also didn’t manually check the ROC curve / AUC score for each label individually because I see no merit in it.
* While it is possible one (or more) class performance is weak, we’re not going to do anything about it, so I simply don’t care.

## Optimal classifier hyperparameters:

1. While I choose very good hyper parameters, there are so many in xgboost algorithm. There is only one way to approximate the best hyper parameters and its through search.
2. This is np hard problem which called hyper parameter tuning.
3. To get such approximation one need to define a search space and use algorithms like grid search random grid search baysian hyper parameter optimization etc.
4. Just like any other machine learning algorithm, this is an approximation of the “real best values”
5. You can see in code the hyper parameters I’ve chosen, including the objective function, loss metrics I’ve chosen to look at, boosting algorithm (trees of course!!!), etc.

## Requested Plots:

* Code to generate plots were almost completely generated by gpt



We can see that the model learnt pretty well to classify “forests” and is least good at classifying “open country”. You can logically hypothesis that forests are very well defined and strong in certain colour while open country could be composed of few different sceneries.

### Q2

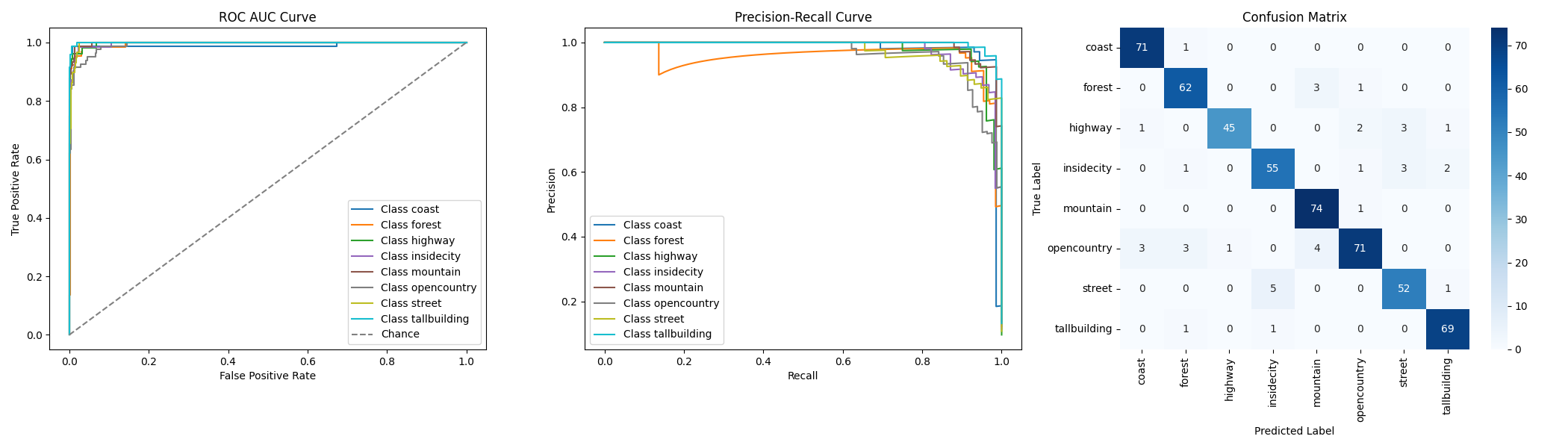
### Training Phase:

* Same as before, splitted training and testing data, so testing is excluded completely.
* When re-constructing vgg16, I’ve decided to use average pooling over the 512x8x8 into 512x1x1 into flatten to get 512 length vector (represent features) for each image.
  + Dodged the redundant kmeans here to save complexity and probably get better performance.
  + Why used average pooling and not flatten the 512x8x8 into 32768 feature vector?
    1. Curse of dimensionality. I’d rather work with 512 over 32K features.
    2. Average those features might (or not) condense the information to be more informative.
    3. Running with this yielded 0.99+ ROC AUC. So by evaluation didn’t even need to try the latter.
* Declared the SceneryImageDataset class which inherit from torch.utils.data.Dataset to declare my training data and add a transformation to the data (used the given normalize), then used a DataLoader object, which helps feed the data through the network in batches.
* Batch size = 32. Decent number to not blow memory up, nothing special.
* Same label extractor like before
* Same process for the validation
* Almost same training over xgboot classifier, only took the eta from 0.08 to 0.1 (this is the learning rate).
* Train roc auc = 1 while validation is about 0.995.
* returns the classifier

# **Testing phase**

* Very similar to previous question.
* Now we have the feature\_extractor and data loader we use to create the features
* We construct xgboost.Dmatrix using the testing\_features).
* Predicting using xgboost classifier and do the same roc\_auc\_score. Which is 0.9957.
* Major success.

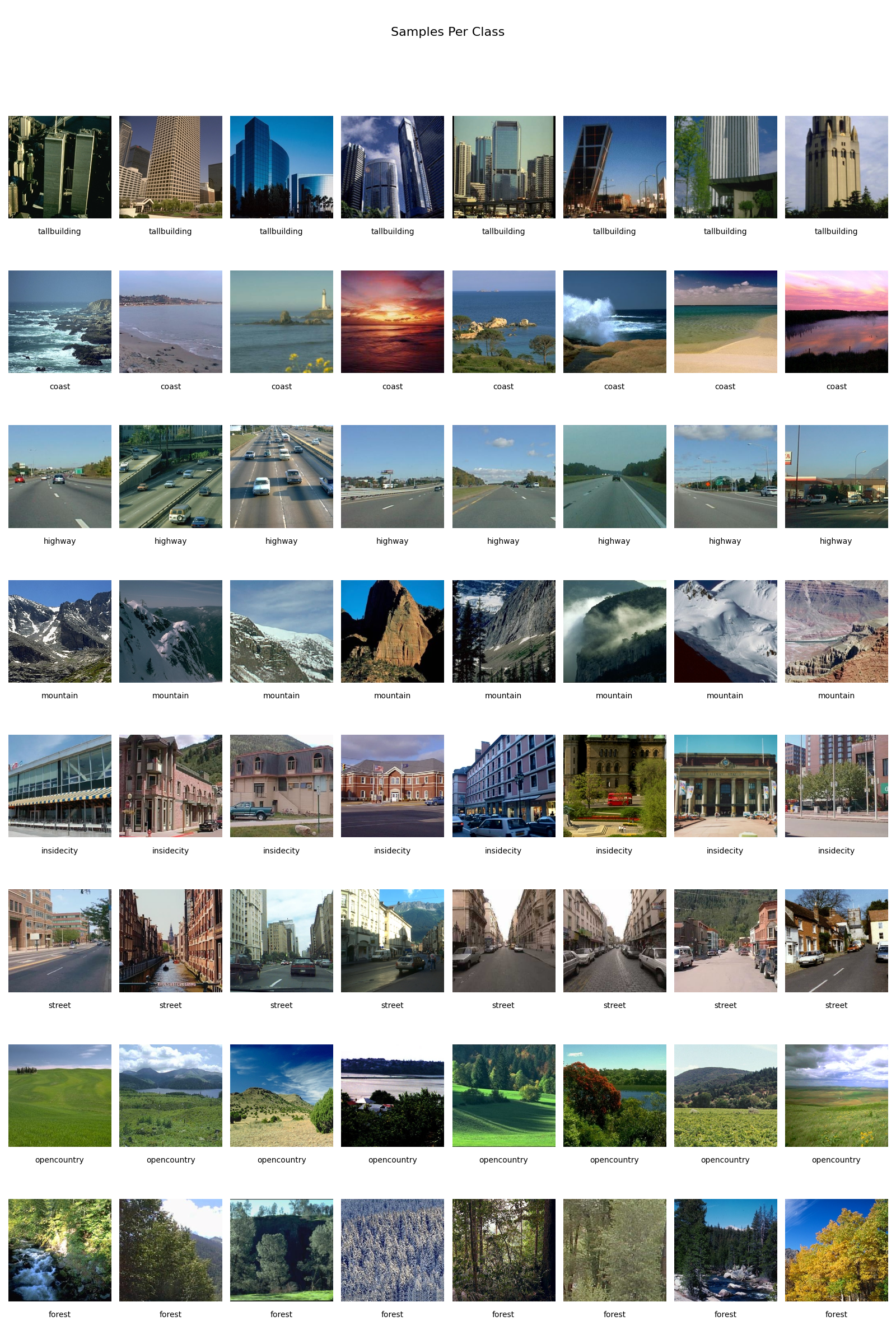
## Requested Plots:



Not much to say about this. The model, almost perfectly catch the features describing each scene, and discriminate correctly between them.

### Q3

Almost all previous processes remain the same:

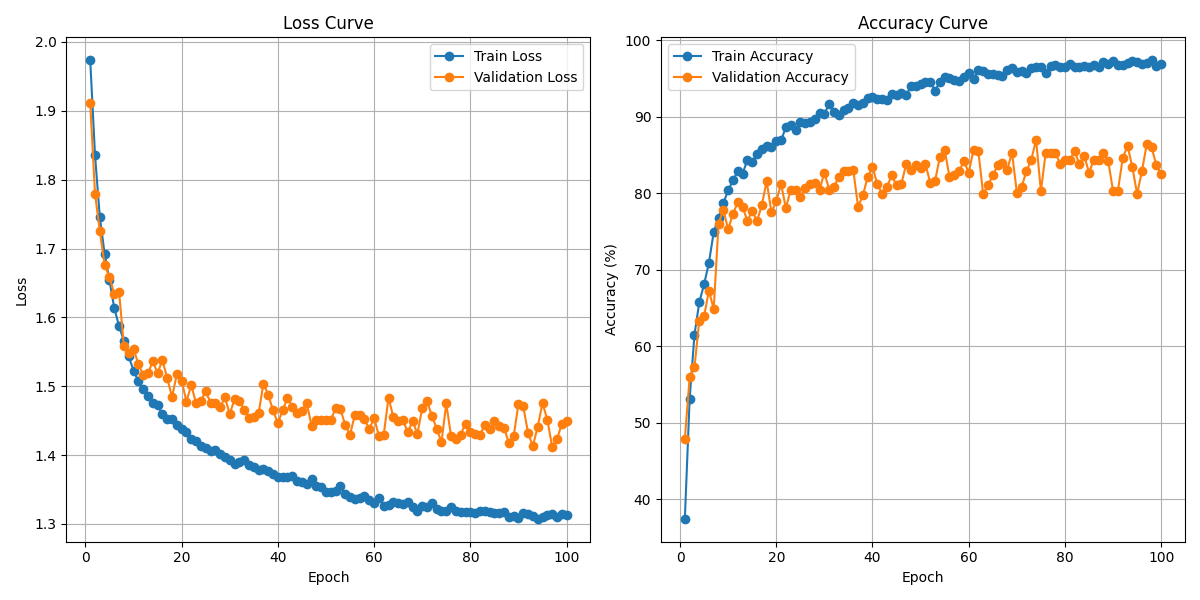
1. Data Loder is defined in code.  
   
2. The neural network I’ve constructed is highly inspired by VGG16.

The changes I’ve made there are the following:

* I’ve added batch normalization after each conv layer.
* Instead of flattening the last CNN layer (8\*8\*512) and build a few fully connected layers, which introduce approx. 32000\*4000 additional weights to tweak, I’ve used average pooling, which reduce forward and backward data flow over the network (especially the backward prop where we need to derive and update weights)
* This tremendously reduce computation time as I tried something similar to the original architecture proposed, while kept a good training error reduction from epoch to epoch.
* Big time saver. (and money if you pay for computing)
* The network called FastCNN
* Basically the network, starting from 32 kernels after the basic 3, at each cnn, increase number of kernals by factor of 2, while reduce “image” dimentionality by a factor of 2.
* Strides and padding are 1 and 1 so that the original image shape will remain.
* Use of maxpool (2x2) to cut in half the dimension of image.
* ADD SUMMARY OF MODEL

I’ve tried a few other different networks, nothing interesting worth noting. You can see in the code in the nn\_architectures.py module what they look like.

Running with 100 epocs, Loss, Accuracy metrics:



As for evaluating over the test set:

Test Accuracy: 87.17%

Test ROC AUC (macro): 0.9823

To improve this, at the effortless way possible, I will run more epochs over a train/validation split, while keep the test separated (using random state=1337).

This didn’t get me to 90%

To improve I would (but I won’t be due to lack of time):

1. Get a deeper network or / and
2. Get a different architecture, as we know ResNets perform better.
3. Augment the data. We have a good example of how to augment random crop 224\*224 (which eventually yields the known 7\*7\*512 layer we know)
4. Fiddle with the optimizer. I’ve chosen 0.0001 at first with Adam and a few other parameters including weight decay to avoid overfit.
5. Keep running more and more epochs until something good will happen. (As long as your network is highly regularized, you’ll probably avoid overfitting and sometime succeed in converging into a better local minima.
6. If nothing else works, just inject the testing data into the training and keep it a secret.

# Requested Plots:

A graph with different colored lines

Description automatically generated

# Overfitting Building Example:

* For this example, I’ve built a faster CNN which called: OverFitFastCNN
* I’ve removed all regularization methods, removed batch normalization, set weight decay to 0 to avoid regularization, removed dropout layers. Removed part of the layers to make the code run faster...
* It didn’t overfit so easily.. need to run many more epocs which just wastes my time, so here is an example of how it would look like:
* 