Intel Scene Classification Challenge Report

Author: Yash Bhalgat

Current Position: Computer Vision and Machine Learning Engineer, Voxel51 Inc.

In this report, I will describe my approach in solving this challenge, a few caveats and intricacies in the implementation, the challenges I faced and my final solution.

I ranked `3`rd on the Public Leaderboard and `6`th on the Private Leaderboard. You can find the code / implementation of my approach here: **github-intel-scene-classification**

This report is structured as follows:

1. Problem Statement

The problem posed in this challenge was to classify photographs categorically as different scenes. As opposed to object classification, context and layout/placement is quite important in this problem. In particular, there were 6 classses in this task:

1. Buildings (2628 images)
2. Forest (2745 images)
3. Glaciers (2957 images)
4. Mountain (3037 images)
5. Sea (2784 images)
6. Streets (2883 images)

Submissions were made on a test set of 7301 images, 30% of which were used to rank participants on the public leaderboard.

Note: In this document, whenever I mention "test accuracy", that implies the accuracy displayed by the solution checker on submission of my predictions. In other words, it means the accuracy on the 30% of the test set.

2. Steps and Approaches

In this section, I will describe in detail the variety of steps I took throughout the competition. Each step led to an improvement in the performance. I will explain the "why" and "how" for each of them.

2.1. Data Splitting

I split each class of the training dataset into fixed 80%-20% sets - called the `train-set` (13625 images) and the `valid-set` (3409 images). The models were generally trained for a 10-20 epochs and having a validation set helped me keep potential over-fitting in check. While the model trained, intermediate checkpoints with best (so far) validation accuracy were saved.

Note: I could have done better here if instead of having a fixed 80% as the training set, I had used cross-validation to split the entire set into 5 equal parts and then used 4 parts for training and 1 part for validation in a "round-robin" fashion. But I did not implement this in my approaches.

2.2. Initial approach

My first approach was to take a set of pretrained weights and fine-tune it on this scenes dataset. I used the weights pretrained on the ImageNet dataset for various network architectures (`AlexNet`, `VGG16/19`, `ResNet18/34/50`, etc). To set some initial baselines, I fine-tuned **only the last layer** of the networks and secured a validation accuracy of ~`90%` and a test accuracy of ~`90%` with the `ResNet34` architecture.

This gave me a baseline to compare my future models. I will now describe various steps I took in order to improve the performance on this task and at the same time, make the models more robust / generalizable.

2.3. Trainable Layers

The ImageNet weights are trained on 1000 classes, most of which are irrelevant to this problem. So, it would be better to **unlearn** the irrelevant features and fine-tune the weights for only these 6 classes. For this reason, instead of just fine-tuning the last layer of the network, I decided to train all the layers of the network starting from the pretrained checkpoint.

Broadly speaking, this was done in 2 steps (as inspired from **fast.ai**'s online deep learning course):

1. As the last layers are randomly initialized, the first step is to fine-tune them while keeping all the previous layers frozen (in PyTorch, you can *freeze* a layer by setting the parameter `is_trainable` as False for all of its parameters). After a few epochs, you see that performance saturates and we already have a good enough performance because we used the amazing ImageNet pretrained weights!
2. After this, all the previous layers are unfreezed and differential learning is implemented to train the previous layers. This basically means that as you go deeper in the model, you would want to change the layers lesser and lesser. A good practice to do this is to exponentially (with a factor of `10` in my case) decrease the learning rate as you go deeper into the network.

Note that, step 2 is about decreasing the learning rate across the layers, not with time! I also decreased the learning rate with time, but I will discuss that in a separate section below.

Just following this helped me significantly bump up the validation accuracy from 90% to 92.6% for the `ResNet34` architecture

2.4. Data Augmentation

Although the amount of data provided in this challenge was ample, we know that in deep learning, the more the data the better! I followed a number of image transformations and other data augmentation techniques to increase the **variability** of the data seen by my network. By training on these transformations, the model becomes more robust to overfitting and consequently generalizes well on images it has not "seen" before.

2.4.1. Image Transformations

The **Albumentations library** was a very helpful resource for this step which provided me a wide range of image transformation methods to plug into my training. I implemented 4 major sets of image transformations in my training procedure which are summarized as follows:

1. **Flipping and Rotation**: Since we are focusing on scene classification, only `HorizontalFlip` seemed appropriate to be used because the model will never see images with mountains flipped upside-down! I also used `RandomRotation` with a limit of 25 degrees - more than that would not have made sense.
2. **Additive Noise and Blurring**: I used additive gaussian noise as one of the distortions to make the model more robust to noisy images. This was overlayed with blurring to capture more complex distortions, again making the model more robust.
3. **Affine Distortions**: This is a very powerful augmentation tool as it changes the geometry of the images. Convolutional Neural Networks are said to learn robustness to spatial transformations very effectively, hence adding affine distortions add a lot of value in the training.
4. **Color Distortions**: One of the most commonly observed variations in scenic images are that of illumination and hue. Hence, I used contrast distortions (with CLAHE), brightness distortions and random Hue-Saturation distortions during training.

All these transformations were **probabilistic**, meaning that if the model were trained for an infinite number of iterations, it would ALWAYS see a different image in each iteration. I tweaked the values of the individual probabilities of each augmentation using practical intution of what scenes distortions usually occur in nature. For example, I used a probability of `0.5` for Horizontal Flips, but a low probability value of `0.2` for Affine Distortions.

After using these transformations for training, the performance of the `ResNet34` model went up to a test accuracy of 94%. It proved effective even with the shallower models, for example, the test accuracy with just one `AlexNet` model reached 93%.

2.4.2. CutOut and Dropout

To further tackle the overfitting problem and make the models more generalizable, I added Dropout as the **second-last** layer in all my models. In particular, I tried using Dropout with a value of `p=0.7` in my models. But Dropout did not show much effect - for some architectures, the validation and test accuracy in fact dropped down by certain amounts.

The second approach I tried was using a recently proposed approach called **CutOut**. This is actually a very useful data augmentation technique. With `num_holes=20` of height and width of `16` pixels, this method **randomly** cuts/removes "`num_holes`" number of patches of the given size from the images. A default probability value of `0.5` was used here. This method alone gave a tremendous improvement in performance. The `ResNet50` model reached a test accuracy of `95.3%`.

2.5. Learning Rate Decay

Sometimes, the optimization/training of the deep models gets stuck in a local minima which leads to poor generalization. Different **adaptive learning rates** can be used to tackle this problem. I tried out a variety of learning rate decay methods, namely

1. `exponential learning rate` - the learning rate decays exponentially. The hyperparameters are the `initial learning rate`, the `decay interval` and `rate of decay`
2. `cosine learning rate decay` - this is just another method for learning rate decay, but the results were better with this method than the above
3. `cosine learning rate with restarts` - this method proved to be the most effective. In this method, after the learning rate decays to a certain value, it is rebooted to its initial value and the decay starts again but with longer `decay_interval`. Because of this, if the model is stuck in a local

minima, the rebooting of the learning rate (to a much larger value) helps the model escape the local minima.

To be specific, the best performing model of `ResNet50` gave a validation accuracy of `95.7%` after using the "cosine learning rate decay with restarts". An initial learning rate of `0.0005` was used with initial decay interval of `2` epochs and decay rate of `0.97`.

3. Incremental Improvements

3.1. Other architectures

Once I had some basic results as described above, explored other architectures, namely `ResNet152`, `DenseNet161/121`, `InceptionResnet`, `ResNext`, `MobileNet`, `NASNet` and `SqueezeNet`. The best performing model was the `ResNet152` model with a test accuracy of `96.21%`.

3.2. Ensembling

After reaching the top 10 on the public leaderboard, the competition had started over the first decimal point of test accuracy. To bump up the accuracy, I used the good-old ensembling technique. The idea behind ensembling is that - a single "highly accurate" model has a low bias but high variance. By ensembling various "high accuracy" models, you can have a low bias with a low variance, i.e. better generalization and no overfitting.

My best submission was an emsemble of `ResNet152` + `DenseNet161` + `ResNext64` + `NASNet`, which gave me a test accuracy of `96.71%` (on the public leaderboard).

3.3. Test-time augmentation

The nail in the coffin was to use augmentation even at the test-time. Remember that I heavily used data augmentation while training. But for testing, till now, I only used the raw test image to make the predictions. In this step, I randomly applied `5` iterations of augmentations on each test image and averaged the the second-last layer output (the layer before the softmax) of the network for these `5` images before computing the softmax probabilities.

This final step gave me my best test accuracy of `96.84%` on the public leaderboard.

4. Acknowledgements

I would like to thank Analytics Vidya for hosting and organizing this challenge. I would also like to thank the people who have worked hard to make the pretrained weights openly available for so many different architectures: **pretrained-models.pytorch**. Also, thanks to the Kaggle Masters for making the **Albumentations** library openly available.

5. Five tips for future participants

In retrospect, for other people who read my code and myself, I would like to document a few directions or tips in which we can proceed differently in the future:

1. **Exploratory Data Analysis** - Before setting out to solve any data science problem, it is extremely beneficial to perform a prior analysis of the data itself. This can give you a lot of cues, for example - imbalance among classes, required cleaning/preprocessing, intra-class and inter-class variation, etc.

2. **Basic baselines** - Nowadays, because it is so easy to implement deep learning algorithms, many people directly jump there before even trying out a few simple baselines. Whenever you encounter a challenge like this one, first setting up a few basic baselines is very important for the following reasons -
    - It will give you an idea of the expected performance from the advanced models you will build later.
    - You can inspect the predictions of these simple baselines to analyze which samples are difficult to classify and get more insight into the data available.
    - Pushing yourself to reach acceptable performance with simplistic baselines will force you to build better features for the data and that is very useful.
3. **Simple tricks** - Be aware of a the tricks. This comes from experience. Cross-validation, ensembling, using different learning rates, diffrent data augmentations, different regularizations are just a few of many tricks that can prove beneficial. Apart from experience, another way to know these is to read research papers and stay up-to-date!
4. **Thorough knowledge of Machine Learning**: Many people enter Data Science competitions without having a strong foundation in Machine Learning, Linear Algebra or Statistics. And that is fine! But having atleast a basic understanding of what is happening when you use `PyTorch` or `fastai` or `sklearn` is very important. For example, enabling `cudnn` while running a `PyTorch` code makes the inference and training a 10 times faster - if you knew this, wouldn't that be awesome?
5. **Explore**: The most effective way to learn is to explore. There were a lot of things I tried that didn't work. There were atleast 20 different combinations of data augmentations I have mentioned in this report than FAILED. But while exploring these, I also learnt a lot of new things which I didn't know before. So, I encourage you to fearlessly explore new methods, even if no one else has tried them before.