Scene Classification Competition for Kaggle

Discussion of Methodology and Results

Leo Woiceshyn and Farzad Niroui

1 Considered Approaches

When considering the task of image classification, we discovered that researchers have been improving at the task at an extremely fast rate. We approached the problem as if we were tasked at a job to produce the best possible results using the best available resources. Thus, in order for us to achieve the best possible results, we wanted to test and implement some of the state-of-the-art methods. The first thing we did was look up some of the past Kaggle competitions for image classification, and see what types of approaches the winners took, and see how they achieved their results [1] [2]. One of the commonalities we noticed in both these approaches, as well as in much of the literature, was the use of a pre-trained convolutional neural network for feature extraction. We also identified a method where you take the pre-trained network, and re-classify the network to adapt to the classes in our dataset, referred to as transfer learning [3].

This pre-trained network is typically trained on millions of images, which can take multiple weeks on a series of high end GPUs. Feature extraction is done by removing the last fully-connected layer, feeding our images forward through the network, and taking the outputs from the second-last layer in the network as our new features. Thus, our feature space gets reduced to a much smaller amount, meaning we can try different types of classifiers quickly on the new dataset without worrying about issues relating to computational cost. Some of the latest state-of-the-art convolutional networks that are used most prominently in recent literatures were Facebook's ResNet [4] and Google's Inception [5]. We noticed that many of the online resources and literature mentioned that using a pre-trained convolutional neural network with just the penultimate layer for feature extraction might lead to overfitting to the ImageNet classes, and that using the antepenultimate layer, although having much more features than the penultimate layer, might be beneficial for achieving better results. For the transfer learning approach, the first step of procedure is to replace the output layer of the original network, which consisted of replacing the classes they used for ImageNet with our own classes for our dataset. Then, continue backpropagating the already-trained network with a low learning rate, using our own dataset and class types, until we get a final model which is adapted to our dataset.

For the first approach, once the feature extraction was complete, we decided that it would be beneficial to try a number of different classification methods as well as ensembles of them, and see what gave us the best results on the public test set. Thus, based on the combination of feature extraction network and classifier type, whichever combination gave us the best result on the public test set, was the one that we decided to go with for our Kaggle submission. If the models given by training our own version of the deep residual network gave us good results, we could also try ensembling those with the results from transfer learning.

2 Preliminary Data Exploration

In addition to exploring different approaches for the image classification task, we decided to also do a bit of analysis on our data, and see if there was any preprocessing that we should do to improve our results. From looking at the class distribution, we can see that the distribution is completely uneven, with the Structures class having a staggering 2114 training examples, while the Food, Sea, and Car classes have 91, 48, and 19 training examples respectively, 1. Thus, we decided to research some methods for dealing with this type of class imbalance, specifically within the realm of image classification. We discovered that there are three common ways of combating this imbalance within the class distribution: oversampling the classes that are underrepresented, undersampling the classes that are overrepresented, and creating synthetic training examples. As described in the approach section, we decided to use the synthetic image creation approach to remedy this problem with the data.

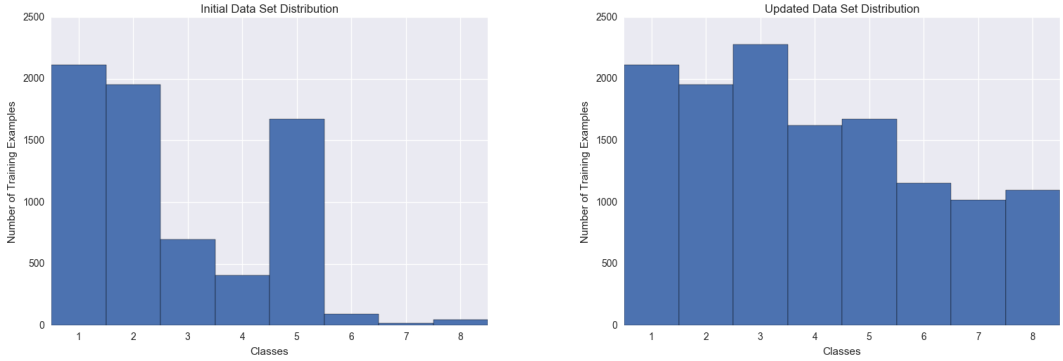

Figure 1: Number of training examples for each class, initial numbers (left) and after generated images (right).

3 Implementing our Approaches

We decided to try both the feature extraction approach and the transfer learning approach and compare the results, as well as potentially combine them using bagging.

3.1 Data Preprocessing

As mentioned above, the uneven class distribution made us worry about the results. In order to remedy this, we chose to expand the given dataset by creating additional synthetic images as training examples for the five classes that were lacking training examples(Animals, People, Car, Food, and Sea). This was done by taking the existing images in the dataset for each of these classes, and applying image augmentation in order to create new synthetic images. A pre-existing Python script [9] was modified to work with our images, and used

to create synthetic examples, by blurring, changing brightness and contrast, rotating, and adding Gaussian noise. Figure 2 shows an example of one of the original images, along with two synthetic images created from it using the image augmentation script:

Figure 2: Original car image (left) and generated car synthetic images (middle and right).

3.2 Transfer Learning with Fine-Tuning

For implementing our transfer learning approach, we decided to use the code provided by Facebook [8], implemented in Torch [9], a scientific computing framework. We chose the Resnet-200 network architecture to train on, since it is the latest release with the best proven results. We trained many different models in order to try different learning rates, batch sizes, number of epochs, and forms of our dataset (original vs synthetic, etc.) until we got the models which produced the best results on our validation set.

3.3 Feature Extraction

For our alternative approach, following the methodologies used by the Kaggle image classification winners, we decided to try the feature extraction approach. We used the latest InceptionV4 [10] network by Google, pre-trained on the ImageNet dataset, and taking the outputs of the penultimate layer of the network as our features. This gave us a 2048-dimensional feature vector for each of our training examples and test set examples, which we used to try a number of different classifiers on, using classifiers that were available in Scikit-Learn. After plenty of trial and error, the best results that we were able to achieve were by using a Support Vector Machine. Multiple different kernels were tested, and the best results from an individual classifier achieved with a 4-degree polynomial kernel. The hyperparameters, kernel, regularization constant C and polynomial coefficient, were tuned using the StratifiedKFold and GridSearchCV methods from the Scikit-Learn package. Our ten-fold cross-validation accuracy on the training set, using this method, was 0.83, and our test set prediction accuracy was 0.81, which was slightly worse than our transfer learning approach.

3.4 Tuning Hyperparameters

For the transfer learning approach, the only hyperparameters to be tuned were the learning rate, batch size, and number of epochs to stop at. It was hard to optimize these fast, since training took such a long time on the deep network. We used a simple 80-20 cross validation split to pick the optimal learning rate and batch size. The computer used for training was limited to a maximum batch size of 16 due to GPU memory. This was a restriction on the upper bound for this hyperparameter for us. In the end, we decided to train two different models using batch sizes of 10 and 16, and the learning rate of 0.0001 was used. Another important consideration was the number of epochs we chose to train the network for. Due to overfitting concerns, we found that a lower number of epochs typically performed better on the validation set. The optimal epoch was chosen by looking at the validation set error trends and convergence, Figure 4. It was found that for batch size of 10 the optimal epoch was 12 and for batch size of 16 it was 15.

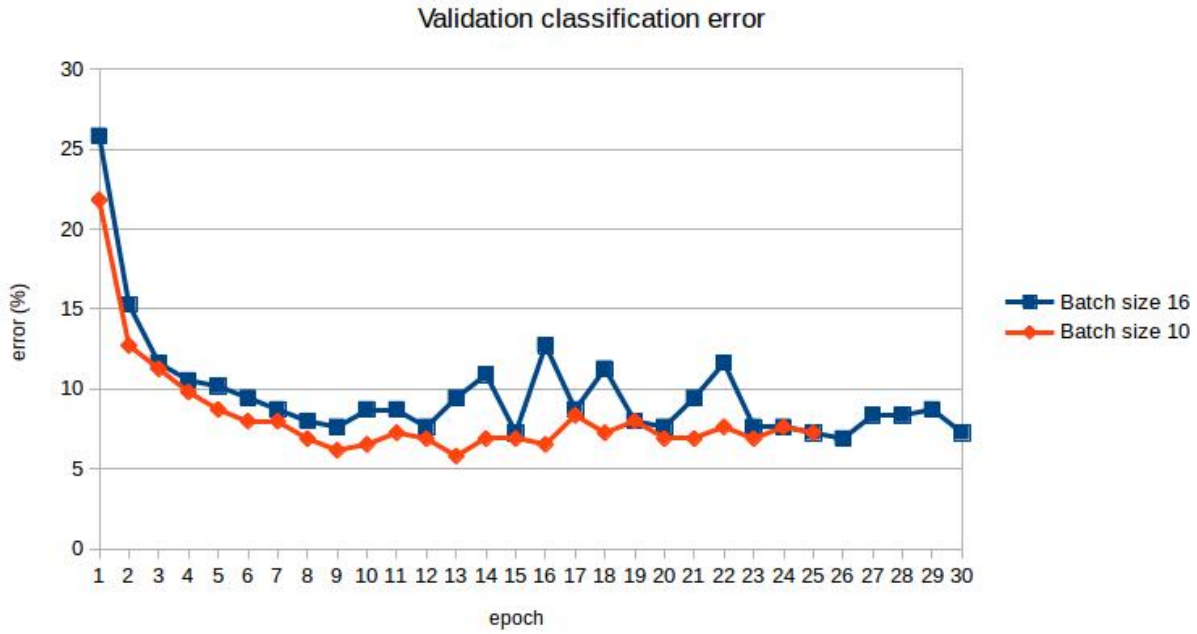

Figure 3: Error on the validation set at different epochs for both batch sizes of 10 and 16.

For the feature extraction approach, we used the best practice way of choosing the hyperparameters for each of the models which was a 10-fold cross-validation using a large selection of possible hyperparameter values. Since the 2048-feature representation was still fairly slow for classifiers such as Adaboost, XGBoost, and SVM, we further reduced the number of components using the principal component analysis class in Scikit-Learn to 50 components to choose the best hyperparameters for each of these classifiers.

4 Comments on our Methods and Our Results

Since our approach was focused on achieving high performance, we believe that we both succeeded as well as exceeded our initial expectations. From the results on the public test set, our team is going into the final evaluation with the best accuracy of 85.464%. Thus, the methods we used were optimal in terms of performance in the competition. We feel that our methodology of researching the state-of-the-art in the field, trying multiple of these approaches and many different algorithms within these approaches, and with our synthetic data creation, were the three main factors in our success. Both of the two main approaches we used performed very well on the dataset, with both of them achieving over 80% accuracy on the public test set with their best models. We did some analysis of which types image classes our best-performing model classified incorrectly, and noticed that it still had the most difficulty with the three lowest-represented classes which we created synthetic images for. This was likely due to the lack of variation in training examples for these classes, as even the augmented images were still very similar to their source images.

4.1 Transfer Learning with Fine-Tuning

For transfer learning one of the pre-trained models, we initially trained on the original dataset of 7000 images with batch size of 10 as a baseline indicator, and were shocked at how well this performed as a baseline, with a Kaggle score of 0.845.

Eventually, after fine tuning with our larger dataset, using the synthetically generated images for the Food, Car, Sea, Animals, and People labels, and using batch size of 16, we trained the network again, which got us our best result of 0.855.

4.2 Feature Extraction

For the feature extraction, we tried multiple different classifiers in Scikit-Learn, and evaluated their performance by using the 10-fold cross-validation accuracy on the best hyperparameters for each model. The following table shows the cross validation accuracies achieved for each different type of classifier used:

In addition to these, we also took a simple ensemble of 9 of these classifiers, using a python script which combined multiple Kaggle submission files and did best-of voting. Unfortunately, this method was not able to outperform the transfer learning approach in the end, although it came quite close, with a 0.83 accuracy on the public test test.

5 Multi Label Classification

For this multi-label classification problem, our approach was as follows. We implemented one of the simplest methods for multiple-instance problems: binary relevance. This approach involves training a one-vs-all classifier for each separate label, predicting the probability of each label for each training example, and choosing a threshold value for which to accept the label as belonging to that training example. Although we understand that this approach has its drawbacks for our case where labels have dependencies, we chose it due to its simplicity in terms of implementation, as other methods such as ensemble of classifier chains were significantly more complex.

Our approach was divided into two stages: preprocessing and classifying. The preprocessing stage involved firstly converting the image labels into a binary matrix of labels, with 1s indicating that this image had the label and 0 indicating that it did not have the label. This was done through the Scikit-Learn package's MultilabelBinarizer class. Next, we did a simple 80-20 training-validation split of the labeled training examples in order to evaluate our approach, and hopefully get some intuition into how well it performs given our label independence assumption. Next, since training would be extremely slow using the raw pixel intensity values, we used our transfer learning feature extraction approach from the single-label classification problem in order to speed up the training time for our binary classifiers. We then used PCA on our 2048 dimensional feature space to convert it to 128 dimensions to speed up our training. For the classifying stage, we chose a linear SVM as our classifier, trained each binary classifier on our dataset, and made predictions on our validation set. Lastly, we applied the evaluation metric from the assignment handout. All of these were done using the following python script:

Which resulted in the accuracy of 0.9551.

The total number of unique labels in the dataset is 25. We split the training set 80-20 for validation, so we had 5600 training examples and 1400 test examples, giving us a 1400 by 25 matrix for our predictions and ground truth labels. Calculating the error using the formula in the handout gave us an accuracy of 0.956 (33,477 labels guessed correctly out of 35,000 total labels), meaning that although we assumed independence between each label, the model was still able to predict the correct labels for the validation set with extremely high accuracy. This was likely due moreso to the fact that our transfer learning feature extraction approach yielded very good results. If we had modeled in the conditional dependencies from the co-occurrence and hierarchical groupings of labels, we would have had an even better accuracy, but given how well we already did, it seems this wasn't very crucial to achieving good results compared to using an optimal method for feature extraction.

```
import pandas as pd
import numpy as np
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.decomposition import PCA


labels = pd.read_excel('train_bonus.xls', header=None)
data = np.load('data_batch_train.npz')
y_train = labels.as_matrix()
y_train_multi = MultiLabelBinarizer().fit_transform(y_train)

y_train = y_train_multi[:5600,:]
y_test = y_train_multi[5600:,:]

pca = PCA(n_components=128)
X_train = data['representations']
pca.fit(X_train)
X_train = pca.transform(X_train)
x_train = X_train[:5600,:]
x_test = X_train[5600:7000,:]
x_test_predicted = OneVsRestClassifier(LinearSVC(random_state=0)).fit(x_train, y_train).predict(x_test)

n = float(y_test.shape[0]*y_test.shape[1])
diff = abs(x_test_predicted - y_test)
sum = np.sum(diff)
acc = float((n-sum)/n)
print 'Accuracy:' + str(acc)
```

Figure 4: Python script for multi-label classification.

References

1:https://goo.gl/WrfC0w
2:https://goo.gl/ZQeDnw
3:http://cs231n.github.io/transfer-learning/
4:https://github.com/facebook/fb.resnet.torch
5:https://github.com/google/inception
6:https://arxiv.org/pdf/1603.05027v3.pdf
7:https://arxiv.org/pdf/1512.03385v1.pdf
8:https://github.com/facebook/fb.resnet.torch
9:http://torch.ch/
10:https://github.com/huvers/img_augmentation
11:https://arxiv.org/pdf/1602.07261v2.pdf