Udacity Machine Learning Nanodegree Capstone

Using deep learning for Scene Classification in Images

Shibin Menachery

September, 2019

1. Definition

1.1. Overview

The amount of visual data has increased exponentially over the past few years. Running ML models to extract certain features like pedestrians, cars etc. without knowing if the image is an eligible candidate can result in sending large amounts of images without the required background and hence wasting a lot of computing resources. For example, we have a ML model that detects restaurants and shops in images, however we keep sending crowdsourced images of nature with no buildings in it. Hence to overcome this, in this proposal I intend to create a scene classification ML model that given a predefined set of scene categories, will classify any unseen image into one of the scene categories.

In this study, a deep learning model based on CNNs is proposed for the Intel Image Classification Challenge from Kaggle[1]. There are other approaches other than the CNN based ones like the bag of words[2] model.

My personal motivation to solve this problem is due to one of the scenarios I am tackling at work where I am encountering a large influx of images which needs to be processed to extract customer specific attributes from them. I found that a lot of my compute time is wasted working on images which are not related to the attributes what the customer specifies. If I can pre assign images with scene labels then this will help me reduce the number images to be sent forward for further processing thus saving on costs and time.

1.2. Problem Statement

Due to an ever-increasing amount of visual data coming from a multitude of devices, it becomes essential to have some form of image classification in place to channelize the right images downstream for further area specific classification/detection solutions. It is not possible for a human annotator to sift through this huge influx of image and categorize them appropriately to be used further. In order to automate this process, we investigate deep learning techniques CNN and transfer learning that can classify scenes.

1.3. Evaluation Metric

The overall evaluation metric that will be used for the benchmark model and the solution model is accuracy:

accuracy = Number of correctly predicted class/Total number of predictions

Then we will use a confusion matrix to evaluate our model to see if the accuracy is skewed to a particular class or not as we know that our dataset is imbalanced. If in the confusion

[1] Kaggle Intel scene classification dataset

[2] Scene classification with low-dimensional semantic spaces and weak supervision paper

matrix we see that classes with less data are indeed having less scores then we will take steps such as under sampling, oversampling or creating PR curves to evaluate our model.

1. Under sampling by removing samples from over-represented classes.
2. Oversampling: SMOTE(Synthetic Minority Over-sampling Technique) to create synthetic samples of the minority class.
3. PR(Precision-Recall) curves, if we are keeping the dataset as it is. High area under the curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate.

2. Analysis

2.1. Data Exploration

The dataset is provided by the Intel scene classification challenge. The dataset contains:
Image Size: 150x150
Training Set: 17034 images (To be split into training, validation and test) **Pred Set**: 7032 images
Categories:

Scene	Label	Image Distribution
buildings	0	2628
forest	1	2745
glacier	2	2957
mountain	3	3037
sea	4	2784
street	5	2883

The images for each label in training is not evenly distributed and class distribution varies from 2628 to 3037. This demonstrates that the data is imbalanced and the data need to be balanced in order to get the best results.

Also while sampling some images I saw it had a combination of scenes but the label was there for only one . It would be interesting to see how this plays out during training.

There are 3 files provided: train.zip which contains all images, train.csv and test.csv which has the following structure:

Key	Value
Image_name	Name of the image in the dataset
Label	Category of natural scene

Source: https://www.kaggle.com/nitishabharathi/scene-classification

2.2. Algorithms and Techniques

2.2.1 Convolutional Neural Network (CNN)

A Convolutional Neural Network is an algorithm that uses sparsely connected layers and accepts matrices as input making it ideal for most image processing tasks, including classification. As it needs large amounts of data compared to other approaches, we address this problem using image augmentation, since we have few images to train it adequately. The role of the ConvNet is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction.

A CNN mainly consists of the following types of layers:

1. Convolution layer: Convolutional layer consists of a set of learnable “filters”. The filters take a subset of the input data at a time but are applied across the full input (by sweeping over the input). The operations performed by this layer are still

linear/matrix multiplications, but they go through an activation function at the output, which is usually a non-linear operation.
The following parameters are used to initialize this layer:

- Number of learnable filters
- Size of the filter
- Padding
- Stride (number of columns by which the filter is moved)
- Activation

2. Pooling layer: Pooling layer replaces the output of the network at certain locations with the summary statistic of the nearby outputs. This helps in reducing the spatial size of the representation, which decreases the required amount of computation and weights.

3. Dropout layer: Dropout layer helps prevent overfitting by randomly disconnecting inputs from the preceding layer to the next layer

4. Flatten : Flattens the output of the convolution layers to feed into the fully connected layers.

5. Fully connected layer(Dense): Fully Connected layer has full connectivity with all neurons in the preceding and succeeding layer and is computed by a matrix multiplication and a bias effect.

Activation functions : Activation layers apply a non-linear operation to the output of the other layers such as convolutional layers or dense layers.

- ReLU Activation : ReLU or Rectified Linear Unit computes the function f(x)=max(0,x) to threshold the activation at 0.
- Softmax Activation : Softmax function is applied to the output layer to convert the scores into probabilities that sum to 1.

Loss function: Loss functions describes how far off the result the network produced is from the expected result - it indicates the magnitude of error the model made on its prediction. The error can be taken and backpropagated it through the model, adjusting its weights and making it get closer to the truth the next time around.

- Categorical Cross Entropy: Categorical cross entropy is a loss function that is used for single label categorization. It is used for multi-class classification

Optimizers : Optimizers update the weight parameters to minimize the loss function.

- RMSprop: Instead of letting all of the gradients accumulate for momentum, RMSprop only accumulates gradients in a fixed window.
- Adam: Adam (Adaptive moment estimation) is an update to RMSProp optimizer in which the running average of both the gradients and their magnitude is used. In practice Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp.

In my experiments, I have used both RMSprop as well as Adam with different learning rates and found Adam to be a better choice.

A few parameters can be tuned for training the network:

- Training length (number of epochs)
- Batch size (images per training step)
- Validation split (training and validation data split)
- Callbacks (Set of functions to be used during training to monitor output)

2.2.2 Transfer Learning

Transfer learning refers to the process of using the weights a pretrained network trained on a large dataset applied to a different dataset (either as a feature extractor or by finetuning the network). Finetuning refers to the process of training the last few or more layers of the pretrained network on the new dataset to adjust the weight. Transfer learning is very popular in practice as collecting data is often costly and training a large network is computationally expensive.

Depending on both:

- the size of the new data set, and
- the similarity of the new data set to the original data set

the approach for using transfer learning will be different. There are four main cases:

- new data set is small, new data is similar to original training data
- new data set is small, new data is different from original training data
- new data set is large, new data is similar to original training data
- new data set is large, new data is different from original training data

As our data set is small and we will:

- slice off the end of the neural network
- add a new fully connected layer that matches the number of classes in the new data set
- randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network
- train the network to update the weights of the new fully connected layer

The following pretrained models will be evaluated:

- VGG16 and ImageNet[3]
- VGG16 and Places365[4]
- Resnet101 and ImageNet[5]
- Resnet152 and Places365[6]

2.3. Benchmark

A vanilla CNN consisting of 2 each of convolutional, dense layers, max pooling and dropouts was used. It used categorical cross entropy for loss, RMSProp for loss and accuracy was

[3] VGG16 model, with weights pre-trained on ImageNet Keras documentation
[4] VGG16 model, with weights pre-trained on Places365 GitHub
[5] Resnet101 model, with weights pre-trained on ImageNet download link
[6] Resnet152 model, with weights pre-trained on ImageNet download link

selected as the metric. The dataset was split into training(10901 images), validation(2726 images) and testing(3407 images) sets. The images were shuffled and passed onto for training in batches of 32. The model was trained for 10 epochs. It achieved the accuracy of 0.75.

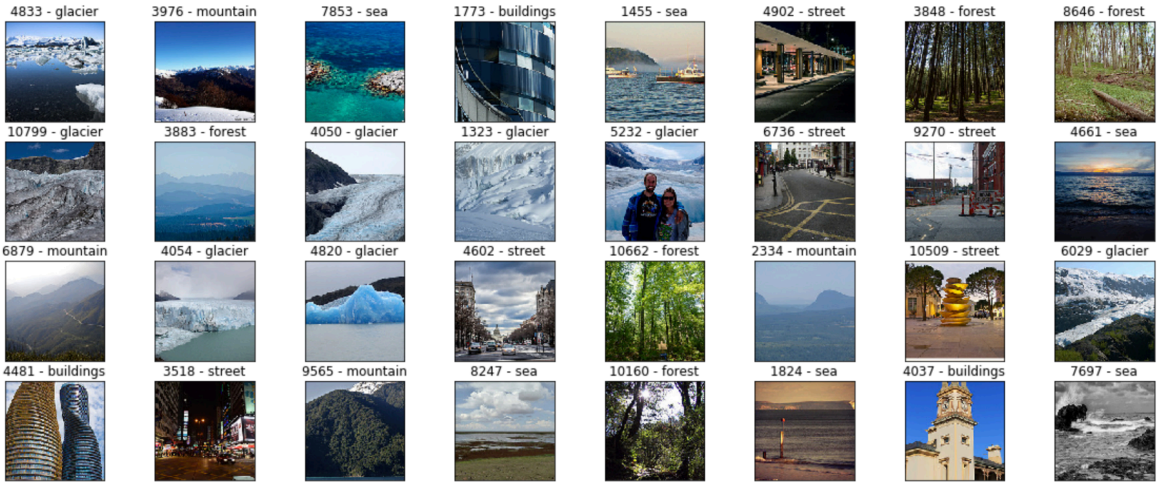
However a training/ validation losses plot showed divergence indicating overfitting, which can be addressed by intensifying the weight decay, adding dropout, or reducing the model complexity (e.g. reducing the number of layers), among other techniques

3. Methodology

3.1. Data Pre-processing

The following steps are taken to pre-process the data:

1. The six categories represented by numerical are assigned labels
2. The images are divided into training set and validation set
3. Rescale the image by dividing every pixel by 255
4. To optimize memory usage, we train with batches of 100
5. Image Data Generators generate training data from the directories/NumPy arrays in batches and processes them with their labels. Training data was also shuffled during training the model, while validation data was used to get the validation accuracy and validation loss during training.



3.2. Implementation

The CNN implemented using Keras with TensorFlow as backend was used for image classification. It took some troubleshooting to get the jupyter notebook use the GPU due to non-compatibility between libraries and TensorFlow. The troubleshooting was mainly done using nvidia-smi and gpu-stat. Once the GPU compatibility was resolved, the following were explored for each step of CNN model development:

- Number of layers: Convolutional, fully connected, pooling and dropout layers. After a few experiments I found out a good fit would be 4 convolutional layers and 3 dense layers were good enough.
- Next to evaluate were the parameters for the convolutional layers such as learnable filters, kernel size and padding
- For activation function, ReLU as it is non-linear and hence can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function
- Number of max pooling layers were added and removed between the convolutional layer with its impact tested. I ended up adding only 2 maxpooling layers one after 2 and 4 conv layers respectively
- Dropout layers were only added once before the final layer as no major improvements were seen after adding it.
- Loss function: sparse categorical cross entropy was used as classes were mutually exclusive

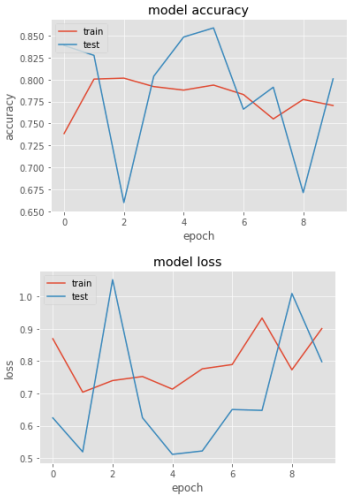
- Optimizer: Adam with a learning rate of 0.0001 was used after experimenting with RMSProp
- The model was checkpointed based on the accuracy metric
- Different number of epochs were tried out based on the results to avoid overfitting or underfitting

Once the above CNN model was evaluated, I decided to explore transfer learning methods to check if there were any improvements in accuracy using a model trained on large datasets such as ImageNet[7] and Places365[8].

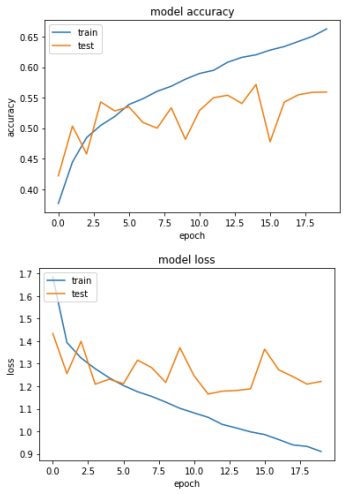
3.3. Refinement

The initial architecture was faced with the problem of overfitting. To resolve overfitting, image augmentation steps were added using ImageDataGenerator. This solved the problem of overfitting but the I could not get any improvements in accuracy.

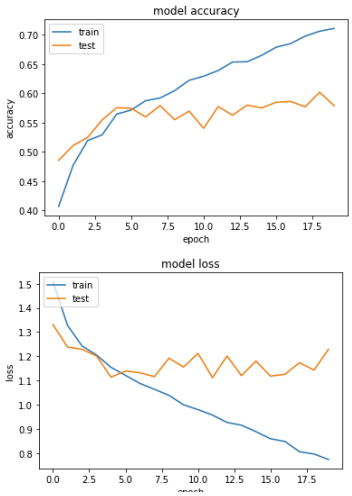
Next we decided to explore transfer learning using pretrained CNN models. We experimented with VGG16, Resnet101 and Resnet152 trained on ImageNet and VGG16 trained on places365 dataset by dropping the final fully connected layer and adding custom fully connected layers. The accuracy hovered around 55% which is close to random guessing and was prone to overfitting with the model unable to learn at all.



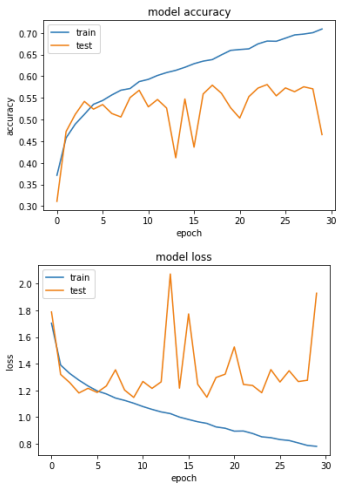
VGG 16 (ImageNet)



VGG 16 (places365)



Resnet101 (ImageNet)



Resnet152 (ImageNet)

As the initial model had better results compared to transfer learning I decided to refine the same to improve accuracy and reduce loss. With overfitting under control I adding layers till I started overfitting, gradually increasing the conv layers from 2 to 6. I used early stopping to know the correct number of epochs to run. I increased the kernel size from 2 to 3 and number of learnable filters from 16 to 200. The loss function was changed from categorical cross entropy to sparse categorical cross entropy and the optimizer was changed from RMSProp to Adam with a lower learning rate. The max pooling layers were reduced to 2 after increasing conv layers as suggested by this paper[9].

---

[7] ImageNet Classification with Deep Convolutional Neural Networks paper
[8] Places: A 10 million Image Database for Scene Recognition paper
[9] Striving for Simplicity: The All Convolutional Net paper

4. **Results**

4.1. Model Evaluation and Validation

During the evaluation, a validation set was used to evaluate the model. For experiment the instead of defining the number of epochs it should run, early stopping callback was used. Modelcheckpoint callback was used to save weights only when an improvement was used.

The final model had 6 convolutional layers followed by 4 dense layers. Only 2 max pooling layer and 1 dropout layer is applied. All activation functions were ReLU except for the last layer where softmax was used. Sparse categorical cross-entropy was used as the loss function. Adam was used as the optimization algorithm. The architecture is as shown below:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_13 (Conv2D)           (None, 148, 148, 300)     8400
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 146, 146, 200)     540200
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 29, 29, 200)       0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 27, 27, 200)       360200
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 25, 25, 150)       270150
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 23, 23, 150)       202650
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 21, 21, 100)       135100
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 4, 4, 100)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1600)              0
_________________________________________________________________
dense_1 (Dense)              (None, 200)               320200
_________________________________________________________________
dense_2 (Dense)              (None, 150)               30150
_________________________________________________________________
dense_3 (Dense)              (None, 100)               15100
_________________________________________________________________
dense_4 (Dense)              (None, 50)                5050
_________________________________________________________________
dropout_1 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_5 (Dense)              (None, 6)                 306
=================================================================
Total params: 1,887,506
Trainable params: 1,887,506
Non-trainable params: 0
_________________________________________________________________
```

Below are the results of the various experiments before finalizing on the final model:

Model	Training loss	Training accuracy	Validation loss	Validation accuracy
CNN(Benchmark model)	0.615	0.983	1.35	0.766
VGG-16 (ImageNet)	0.90	0.771	0.798	0.801
VGG-16 (Places365)	0.91	0.663	1.221	0.559
Resnet101 (ImageNet)	0.767	0.71	1.39	0.546
Resnet152 (ImageNet)	0.78	0.709	1.93	0.465
CNN (Final model)	0.42	0.867	0.407	0.856

In order to evaluate the robustness of the model, cross validation technique was used by dividing the internal dataset into 64% for training, 20% for validation and 16% for testing. To ensure equal distribution and to avoid overfitting it was ensured that all the images from different classes were randomly shuffled and redistributed.

4.2. Justification

The benchmark model had accuracy which was close to 100% for the training data indicating overfitting. The final CNN model had an accuracy and loss in which both the training and validation results are in sync with an accuracy of (85.6%) and loss of (0.407).

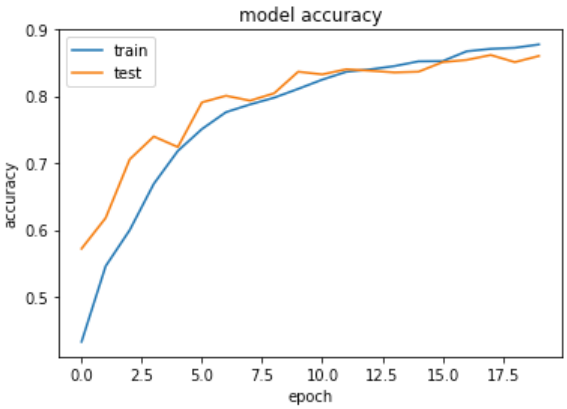
On a p2.xlarge(single GPU with 11GiB RAM) machine, the model I was able to classify 2000 images in 20 seconds.

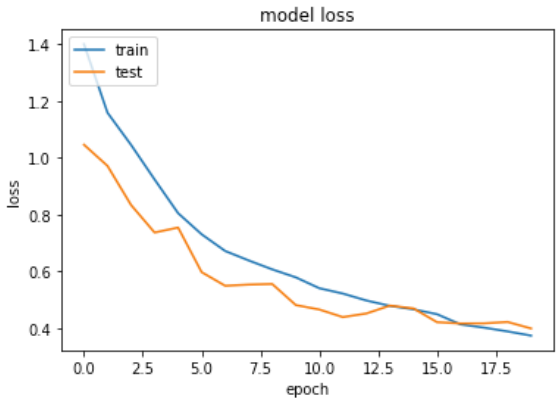
With the achieved accuracy and speed, the final model can be used as an initial model to classify large batches of data for scene classification before it is sent for downstream processing.

5. Conclusion

5.1. Free-From Visualization

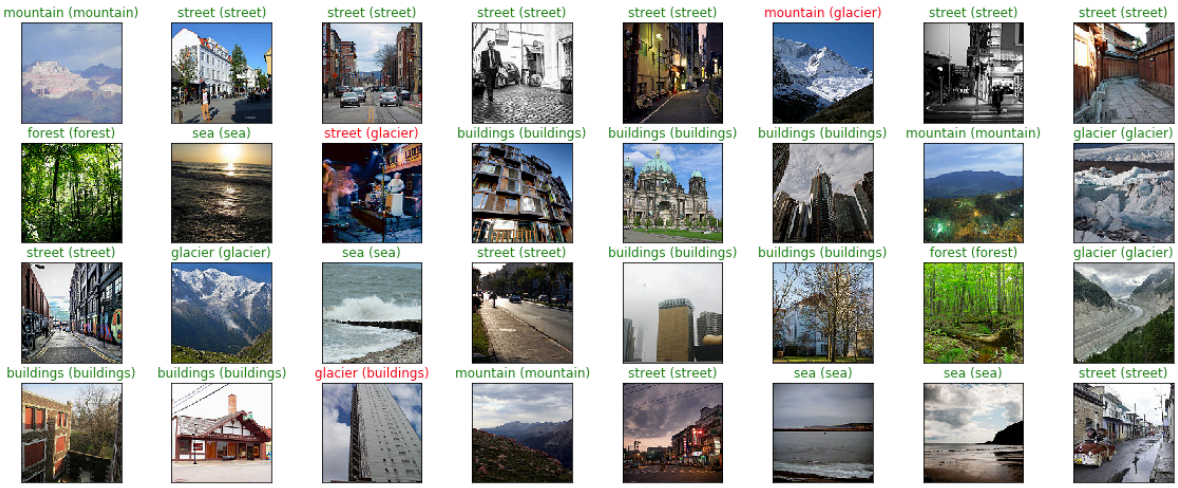
Below is the accuracy and loss plots for training and validation data of the model per epoch.





The training accuracy is near to 90% and the loss is around 0.4 at the end of 20 epochs. We see the trend that the training and validation loss are still going down together indicating a few more epochs may be helpful in getting a higher accuracy without overfitting.

Prediction for 32 test images:



5.2. Reflection

I started off by improving the CNN model I used for the benchmark. I started off by augmenting the images as I felt that increasing the number of images might improve the accuracy of the model. In terms of accuracy the model did not show much improvement but compared to the benchmark model the problem of overfitting was solved as the accuracy and loss function plots started to improve together. As I was eager to try transfer learning as I felt it will be able to solve my problem due to the similarity of the images on the datasets the VGG16, Resnet101, Resnet 152 were trained on. I used the models that had been trained on places365 dataset from GitHub. However I had overestimated how easily transferable, transfer learning is. I was getting accuracy in the range of 65% which was far lower than my benchmark model. Maybe by increasing epochs I might have increased the accuracy and reduced the loss. So I returned

back to tweaking my CNN model. After experimenting with multiple layers, I was finally able to receive satisfactory results.

The most difficult part for me was getting the entire setup running. I started with my local machine but due to lack of GPU the training was taking forever. I then started using a GPU instance from AWS. I tried to setup a machine using an AMI with an older TensorFlow version. This resulted in many of my existing code not running due to older version of other libraries such as NumPy. Upgrading these libraries would result in GPU not being used and only CPU being utilized due to incompatibility between older CUDA libraries. After consulting a few other ML engineers I decided to use the latest AMI available. Due to cost and time constraints, I could run only a certain number of experiments but it was good enough to get decent results.

The final model with an accuracy of around 85%, can be used as a base model to reduce the number of false positive images that are used for downstream processing. The downstream models need to be equipped to handle the fallouts due to the errors.

5.3. Improvement

Due to the time and computation cost it was not possible to run experiments with different known architectures such as InceptionV3 as well as run for higher number of epochs to see if it effects the learning for this dataset.

As the classes were heavily imbalanced, if I generate augmented dataset for the classes that have less data than the others, the dataset can be more balanced and will lead to improved training.

In some of the images in the dataset, the image can be categorized into more than one class. For example in the below image it can be seen that there are both building and street in the same image leading to confusion as to which category it belongs.

Building and street

Further removing such images will help improve the model accuracy.