Machine Learning Engineer Nanodegree
Capstone Project
Landmark Recognition

Fady Morris Milad[†]

December 13, 2019

[†] Email: fadymorris86@gmail.com

Contents

1 Definition 1
1.1 Project Overview 1
1.2 Problem Statement 1
1.3 Metrics 2
1.3.1 Accuracy Score 2
1.3.2 Global Average Precision Metric (GAP) 2

2 Analysis 3
2.1 Data Exploration 3
2.2 Exploratory Visualization 5
2.3 Algorithms and Techniques 5
2.4 Benchmark 6

3 Methodology 7
3.1 Data Preprocessing 7
3.2 Implementation 7
3.3 Refinement 10

4 Results 12
4.1 Model Evaluation and Validation 12
4.2 Justification 12

5 Conclusion 13
5.1 Free-Form Visualization 13
5.2 Reflection 14
5.3 Improvement 14

References 15

1 Definition

1.1 Project Overview

Computer vision algorithms include methods for acquiring, processing, analyzing and understanding digital images, and extraction of data from the real world. It is an interdisciplinary field that deals with how can computers gain a high-level understanding of digital images. It aims to mimic human vision. Convolutional neural networks are now capable of outperforming humans on some computer vision tasks, such as classifying images.

In this project, I provide a solution to the Landmark Recognition Problem. Given an input photo of a place anywhere around the world, the computer can recognize and label the landmark in which this image was taken.

The dataset I used for this project is Kaggle Google Landmark Recognition 2019 dataset and can be downloaded from Common Visual Data Foundation[1] Google Landmarks Dataset v2.

Related Academic Research : [Zhe+09] and [CCF16]

1.2 Problem Statement

Did you ever go through your vacation photos and ask yourself: What is the name of this temple I visited in China? Who created this monument I saw in France? Landmark recognition can help! This technology can predict landmark labels directly from image pixels, to help people better understand and organize their photo collections.

This problem was inspired by Google Landmark Recognition 2019 Challenge on Kaggle.

Landmark recognition is a little different from other classification problems. It contains a much larger number of classes (there are a total of 15K classes in this challenge), and the number of training examples per class may not be very large. Landmark recognition is challenging in its way.

This problem is a multi-class classification problem. In this problem, I built a classifier that can be trained using the given dataset and can be used to predict the landmark class from a given input image. I have chosen to use convolutional neural networks and transfer learning techniques as classifiers. CNNs yield better results than traditional computer vision algorithms. I trained a basic convolutional neural network, then used pre-trained VGG16 and Xception models in transfer learning to solve the Google Landmark Recognition 2019 Problem.

[1] The Common Visual Data Foundation is a 501(c)(3) non-profit organization with a mission to enable open community-driven research in computer vision.

1.3 Metrics

I used two evaluation metrics for my project. Accuracy score, which is implemented in Keras, and Global Average Precision Metric (GAP).

1.3.1 Accuracy Score

The primary evaluation metric for this problem is the accuracy score. Since our dataset is balanced (not skewed), the accuracy score can be used successfully. Accuracy is the fraction of predictions our model got right.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

$$\texttt{accuracy}(y, \hat{y}) = \frac{1}{n_\text{samples}} \sum_{i=0}^{n_\text{samples}-1} 1(\hat{y}_i = y_i)$$

1.3.2 Global Average Precision Metric (GAP)

This metric[2] is also known as micro Average Precision (microAP), as per [PLR09] works as follows:

1. For each query image (data point) in the input of N images (submission), predict landmark label (class) and corresponding confidence score (probability).
2. Sort data points in the list of predictions in descending order by their confidence scores.
3. The Global Average Precision is computed as :

$$GAP = \frac{1}{M} \sum_{i=1}^{N} P(i) rel(i)$$

where:

- N is the total number of predictions returned by the system, across all queries
- M is the total number of queries with at least one landmark from the training set visible in it (note that some queries may not depict landmarks)
- $rel(i)$ denotes the relevance of prediction i

$$rel(i) = \begin{cases} 1, & \text{if the } i\text{-th prediction is correct} \\ 0, & \text{otherwise} \end{cases}$$

- $P(i)$ is the precision at rank i,

$$P(i) = \frac{1}{i} \sum_{j=1}^{i} rel(j)$$

GAP score favors the correct predictions with higher confidence over the correct predictions at lower confidence.

The implementation of this metric was obtained from David Thaler GAP metric implementation on Kaggle.

[2] Source: Google Landmark Recognition 2019 - Evaluation

2 Analysis

2.1 Data Exploration

The original dataset I used for this project is Common Visual Data Foundation[3] Google Landmarks Dataset v2.The dataset description is in the Kaggle Google Landmark Recognition data section. This dataset was used in [Noh+17].

It currently has approximately 15K classes with 4,132,914 images. And it has some problems :

- It is a .csv file that has links to download images from.
- The dataset is very large ≈ 85 Gigabytes
- it has a large number of classes ≈ 15K
- Some classes have a low count of training examples.
- Images in the links have very large resolutions. They have to re-scaled to a reasonable resolution to save disk space and facilitate dataset sharing on the web.
- Some download links may become outdated and broken.

The original dataset index files can be downloaded from Google Landmarks Dataset v2
and they are :

- train.csv: Contains URL to download images from the web, along with their IDs and landmark category IDs.
- train_label_to_category.csv: Contains mapping between landmark IDs and landmark names.

[3] The Common Visual Data Foundation is a 501(c)(3) non-profit organization with a mission to enable open community-driven research in computer vision.

I used only a small subset of the original Google dataset. I selected 10 landmarks(classes) which have a large count of training example (> 1000) and downloaded approximately 1000 images per each class, to have a total of $\approx$ 10,000 images (See subsection 3.1 - Data Preprocessing) :

Table 1: Google landmarks dataset subset

	landmark_id	category	images_count
1	40088	Masada	1056
2	164773	Dead_Sea	1048
3	176018	Hayravank_monastery	1045
4	168098	Golden_Gate_Bridge	1037
5	165900	Mount_Arapiles	1027
6	25093	Matka_Canyon	1017
7	9070	Feroz_Shah_Kotla	978
8	127516	Burrator	976
9	56827	Kazan	975
10	147897	Kasteel_Amerongen	961

Then I divided the downloaded subset into 70% training, 15% validation, 15% test datasets:

Table 2: Training, Validation, and Testing subsets

(a) Training Data Frequencies

Landrmak	images_count
Masada	739
Dead_Sea	734
Hayravank_monastery	730
Golden_Gate_Bridge	726
Mount_Arapiles	719
Matka_Canyon	712
Feroz_Shah_Kotla	685
Burrator	683
Kazan	682
Kasteel_Amerongen	673

(b) Validation Data Frequencies

Landmark	images_count
Masada	159
Hayravank_monastery	157
Dead_Sea	157
Golden_Gate_Bridge	155
Mount_Arapiles	154
Matka_Canyon	153
Burrator	147
Feroz_Shah_Kotla	146
Kazan	146
Kasteel_Amerongen	144

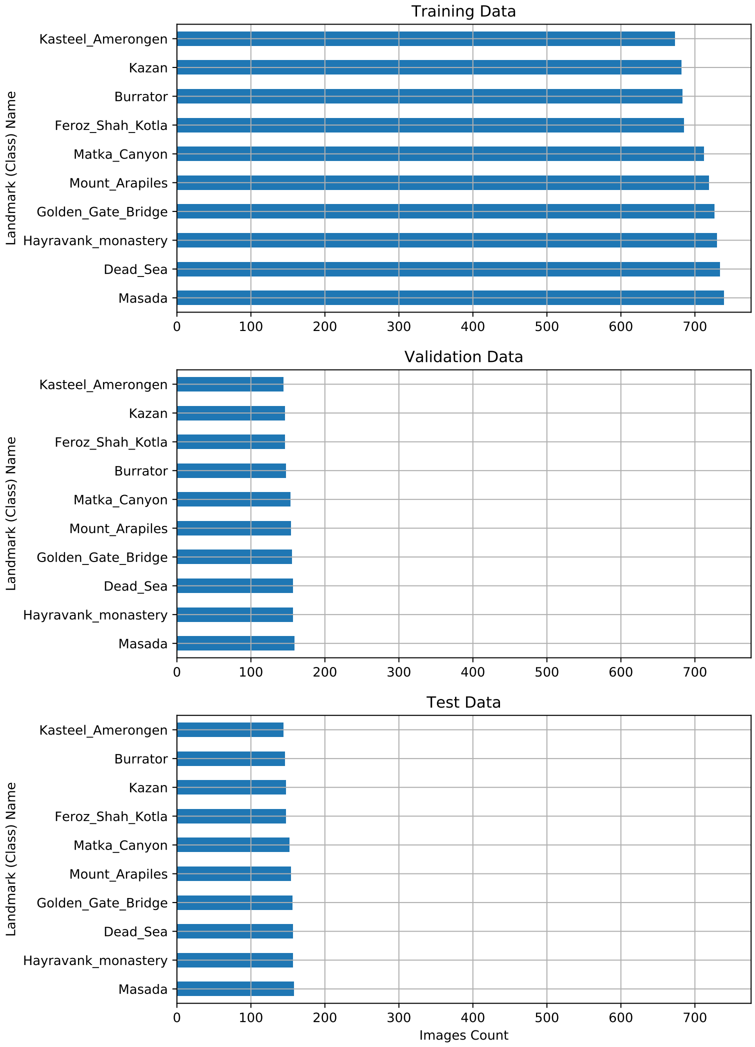
(c) Testing Data Frequencies

Landmark	images_count
Masada	158
Hayravank_monastery	157
Dead_Sea	157
Golden_Gate_Bridge	156
Mount_Arapiles	154
Matka_Canyon	152
Feroz_Shah_Kotla	147
Kazan	147
Burrator	146
Kasteel_Amerongen	144

2.2 Exploratory Visualization

The horizontal bar plots in Figure 1 show the frequency distribution of classes in training, validation and testing sets (From Table 2). The y-axis shows the landmark categories and the x-axis represents the frequencies of landmark categories. From the plot we can see that the datasets are well balanced and the image samples are almost uniformly distributed between classes:

Figure 1: Plot of training, validation and test sets



2.3 Algorithms and Techniques

I have chosen the transfer learning technique to solve the given problem. In transfer learning, we build our model to take advantage of some pre-trained model architectures that take hours for supercomputers to train. The fact that a neural network can learn different features at different layers and that a pre-trained model trained on a task can be used for a similar task is the basis of this technique.

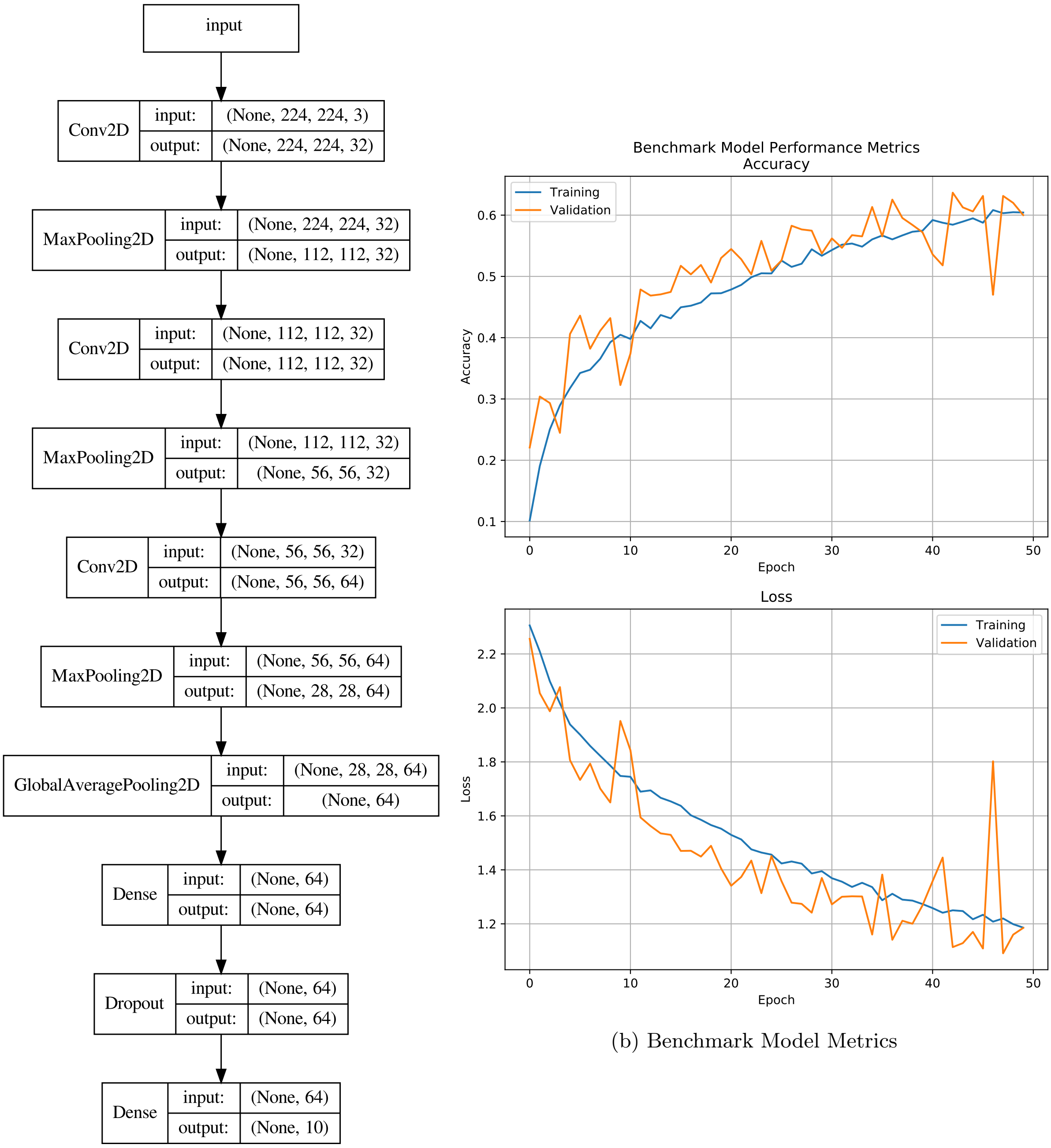
In this problem, I will use *VGG16* and *Xception* pre-trained models from Keras Applications that were trained on the ImageNet dataset without their top layers (fully connected layers) as feature extractors. I will configure my models to apply *global average pooling* to the output of their last convolutional blocks, so the output model will be a 2D tensor. Then, I will pass my training dataset through each network and save their output *bottleneck features*. Finally, I will create a *top model* for each network that is composed of fully connected layers that take such bottleneck features and output a vector corresponding to our landmark classes (here 10 classes). Transfer learning speeds up training and improves performance significantly.

I will compare my reused pre-trained models to a benchmark traditional convolutional neural network that I'll implement and train from scratch.

2.4 Benchmark

Our benchmark convolutional neural network is a simple stack of 3 convolution layers with *ReLU* activation followed by *max-pooling*, *global average pooling*, and *dense* layers (See Figure 2a). For the final output *dense* layer, I used the *SoftMax* activation function to get a vector of probabilities in the $[0, 1]$ range. This is very similar to the architectures that Yann LeCun advocated in the 1990s for image classification [Le +90] (except for ReLU). I have also added a *dropout* layer to prevent overfitting.

Figure 2b shows the performance of the benchmark model. The model was trained for 50 epochs and achieved a test *accuracy* of 62.29% and *GAP* score of 55.85% (See Metrics)


input
Conv2D | input: (None, 224, 224, 3) | output: (None, 224, 224, 32)
MaxPooling2D | input: (None, 224, 224, 32) | output: (None, 112, 112, 32)
Conv2D | input: (None, 112, 112, 32) | output: (None, 112, 112, 32)
MaxPooling2D | input: (None, 112, 112, 32) | output: (None, 56, 56, 32)
Conv2D | input: (None, 56, 56, 32) | output: (None, 56, 56, 64)
MaxPooling2D | input: (None, 56, 56, 64) | output: (None, 28, 28, 64)
GlobalAveragePooling2D | input: (None, 28, 28, 64) | output: (None, 64)
Dense | input: (None, 64) | output: (None, 64)
Dropout | input: (None, 64) | output: (None, 64)
Dense | input: (None, 64) | output: (None, 10)
Benchmark Model Performance Metrics
Accuracy
Training
Validation
0.6
0.5
0.4
0.3
0.2
0.1
Accuracy
0
10
20
30
40
50
Epoch
Loss
Training
Validation
2.2
2.0
1.8
1.6
1.4
1.2
Loss
0
10
20
30
40
50
Epoch

(a) Benchmark Model Architecture

(b) Benchmark Model Metrics

Figure 2: Benchmark Model Architecture and Performance Metrics

3 Methodology

3.1 Data Preprocessing

I downloaded the dataset index files from Google Landmarks Dataset v2 and they are :

- train.csv : CSV with `id, url, landmark_id` fields:
  `id` is a 16-character string, `url` is a string, `landmark_id` is an integer.
  It contains URLs to download images from the web, along with their ids.

- train_label_to_category.csv : CSV with `landmark_id, category` fields:
  `landmark_id` is an integer, `category` is a Wikimedia URL referring to the class definition.

Then, I loaded `train.csv` into a Pandas Dataframe and queried it to extract 10 classes with ≈ 1000 sample images and saved them into an index subset (See Table 1 - Google landmarks dataset subset).

Next, I used Scikit-Learn `train_test_split` function twice with the `stratified` parameter to split the index subset in a stratified fashion (in equal ratios) into 70% training, 15% validation, 15% test index sets (See Table 2 - Training, Validation, and Testing subsets). I saved these sets into `index_train.csv`, `index_validation.csv` and `index_test.csv` files respectively.

Then, I used Kaggle Landmark Recognition Challenge Image Downloader Script to download the dataset image files using URLs extracted from the index files. I modified this script to re-scale the images to a resolution of 640 × 480 before saving them to my local machine to save disk space and facilitate dataset sharing on the web.

Next, I executed the following shell command to remove empty downloaded files (that correspond to broken download URLs) :

```
find . -type f -empty -exec rm {} \;
```

The total count of downloaded images is 10,146 images, totaling 1.1 GB

Finally, to use the downloaded images as input to my learning algorithms I used Keras ImageDataGenerator.`flow_from_directory()` to generate batches of tensor image data with real-time data augmentation. I set the `target_size` parameter to be 224 × 224 and re-scaled pixel images to be in the range $[0, 1]$ by setting `rescale=1/0.225`

3.2 Implementation

I implemented my transfer-learning models in Python in an object-oriented programming approach. OOP is useful for code organization and reuse. It also facilitates the creation and management of many instances of Keras pre-trained models.

I defined a DataProcessor class that wraps Keras ImageDataGenerator() and has attributes like `train_generator`, `validation_generator`, and `test_generator` that act as directory iterators. They return batches of augmented images. This class also handles image augmentation and pixel value re-scaling to the range $[0, 1]$

The following code is an empty outline of `DataProcessor` class :

Listing 1: DataProcessor

```python
class DataProcessor:
    train_datagen = keras.preprocessing.image.ImageDataGenerator()
    test_datagen = keras.preprocessing.image.ImageDataGenerator()
    train_generator = None
    validation_generator = None
    test_generator = None
    input_shape = None
    batch_size = None
    def __init__(self, input_shape, batch_size,
                 train_dir, validation_dir, test_dir):
    def init_train_generator(self, train_dir):
    def init_validation_generator(self, validation_dir):
    def init_test_generator(self, test_dir):
```

I also defined a PretrainedModel class that wraps a pre-trained transfer learning model and our defined top-model. It has methods for :

- Calculation of bottleneck features.
- Saving and loading bottleneck features to .npz files.
- Top-model creation, compiling and training.
- Saving and loading top-model history and best weights.
- Calculation of test accuracy and GAP scores (Metrics).
- Plotting learning curves.

The following code is an empty outline of `PretrainedModel` class :

Listing 2: PretrainedModel

```
class PretrainedModel:
    model_name = None
    pretrained_model = None
    top_model = None
    history = None
    data_processor = DataProcessor()
    save_path = None
    bottleneck_features_train = None
    bottleneck_features_validation = None
    bottleneck_features_test = None
    def __init__(self, model_name, pretrained_model,
                 data_processor, save_path ):
    def predict_bottleneck_features(self):
    def save_bottleneck_features(self):
    def load_bottleneck_features(self):
    def create_top_model(self, optimizer):
    def save_top_model_graph(self, figures_path):
    def train_top_model(self, epochs, batch_size):
    def save_top_model_history(self):
    def load_top_model_history(self):
    def load_top_model_weights(self):
    def test_top_model(self):
    def get_GAP(self):
    def plot_learning_curves(self, save_path):
```

Steps for creation, training, and testing of pre-trained models :

First, create an object of DataProcessor class :

```
data_processor = DataProcessor(
    input_shape,
    batch_size,
    train_dir,
    validation_dir,
    test_dir)
```

Then, For each pre-trained model follow the following steps :

1. Create an instance of PretrainedModel class, passing one of VGG16 or Xception models instances without their top layers (`include_top=False`) and with average pooling mode for feature extraction (`pooling='avg'`) and an object of DataProcessor class :

   ```
   pretrained_model = PretrainedModel("VGG16_model",
                                    applications.VGG16(include_top=False, weights='imagenet',pooling='avg'),
                                    data_processor,
                                    models_dir)
   ```

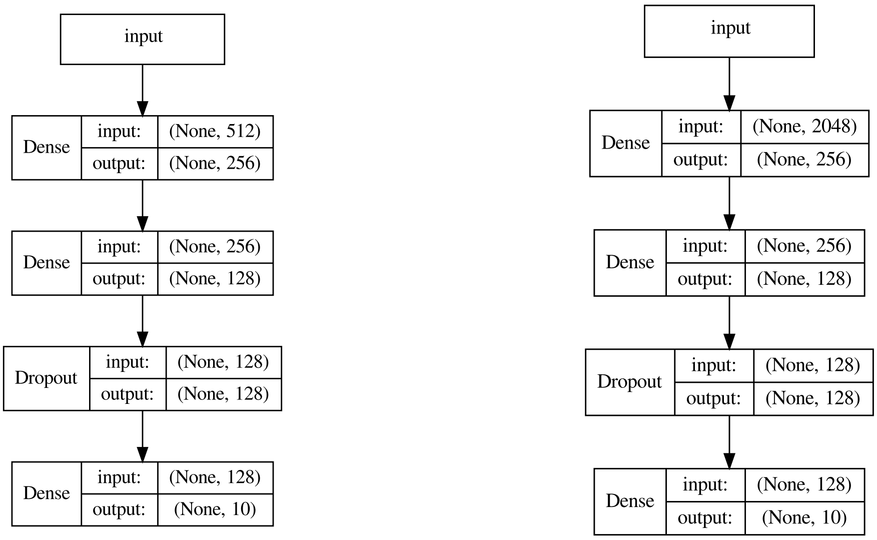
2. Calculate bottleneck features by calling `pretrained_model.predict_bottleneck_features()`.

3. Create and compile the top-model passing your optimizer of choice to `create_top_model()` function. In my initial solution I used *stochastic gradient descent* (`SGD`) optimizer :

Listing 3: Optimizer selection for top-model

```
pretrained_model.create_top_model(optimizers.SGD(lr=0.01, clipnorm=1.,momentum=0.7))
```

`create_top_model()` function defines a fully connected top model that has three *Dense* layers with input size equals to bottleneck features. it takes bottleneck features as input and outputs a vector of 10 classes corresponding to our 10 landmark categories. I also used a *Dropout* layer to reduce overfitting (See Figure 3). I used `ReLU` activation for the first and second *Dense* layers and `Softmax` activation for the last layer to create a vector of probabilities with values between 0 and 1.



(a) VGG16 top-model architecture

(b) Xception top-model architecture

Figure 3: Pretrained models top layers architecture

4. Fit (train) the model using the bottleneck features for 1000 epochs(we can choose a large number of epochs as training the dense layers is very fast, unlike our benchmark model that has convolutional layers)

```
pretrained_model.train_top_model(epochs=1000, batch_size=4096)
```

In `train_top_model()` function implementation I have employed ModelCheckpoint to save top-model best weights and EarlyStopping callback functions to stop the model if it shows no validation accuracy improvement `(monitor='val_acc')` after a set number of epochs `(patience=100)`.

5. Calculate Metrics for the model:
Get accuracy and loss by calling `pretrained_model.test_top_model()` and get Global Average Precision Metric (GAP) by calling `pretrained_model.get_GAP()` methods.

6. Plot learning curves of the top-model training history by calling `pretrained_model.plot_learning_curves()`

3.3 Refinement

For refinement, I recreated `VGG16` and `Xception` pre-trained models, but I changed the optimizer type for the top-model (see Listing 3 - Optimizer selection for top-model) from *stochastic gradient descent*(`SGD`) to `RMSProp` [first proposed by Hin12] by modifying the code to be :

```
pretrained_model.create_top_model(
                        optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0))
```

Keras' documentation of RMSprop recommends leaving the parameters of this optimizer at their default values (except the learning rate, which can be freely tuned).

The results of running the benchmark model, initial solution with `VGG16` and `Xception` using `SGD` optimizer and refinement using `RMSprop` optimizer can be summarized in Table 3 :

Table 3: Test metrics for initial and refined solutions

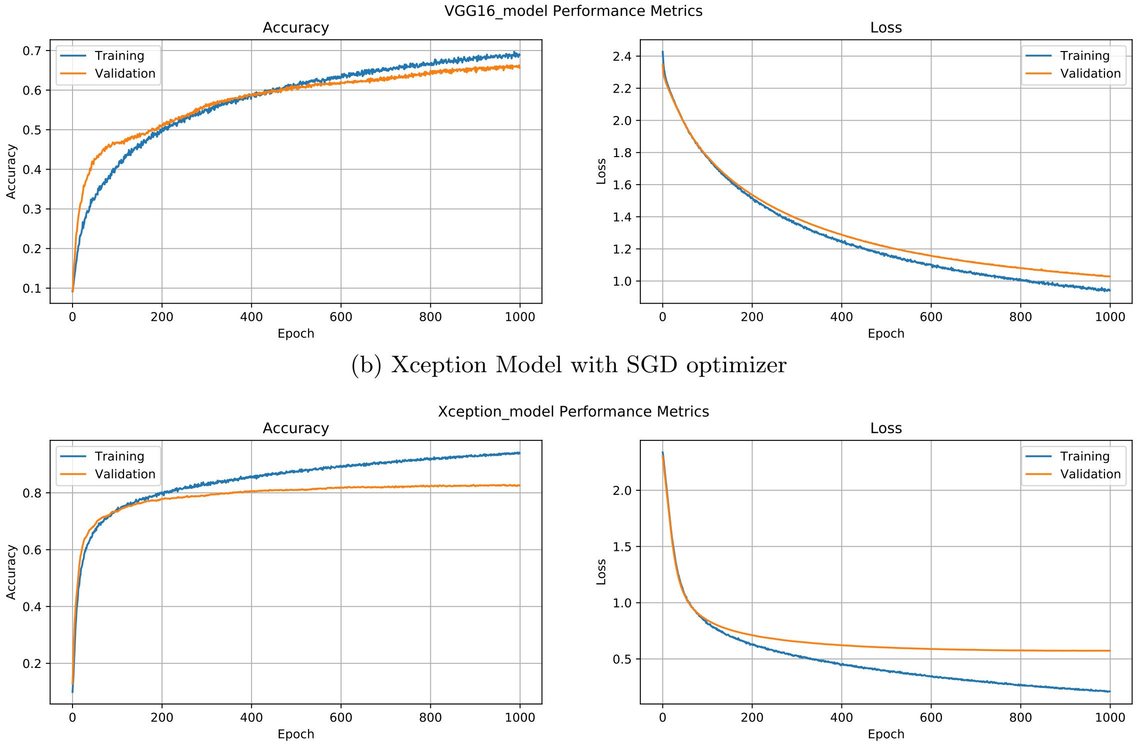
Model	Test Loss	Test Accuracy(%)	Test GAP(%)
Benchmark Model	1.1271	64.29	55.86
VGG16 Model	0.9676	69.43	62.26
Xception Model	0.5337	83.93	81.5
VGG16 Model(Refined)	0.747	77.34	73.3
Xception Model(Refined)	0.5863	84.91	82.68

From Table 3 we can see that the best performance metrics were obtained from the refined `Xception` model with `RMSprop` optimizer.

The Learning curves of the initial solution with SGD optimizer are shown in Figure 4 and the Learning curves of the refinement solution with RMSprop optimizer are shown in Figure 5.

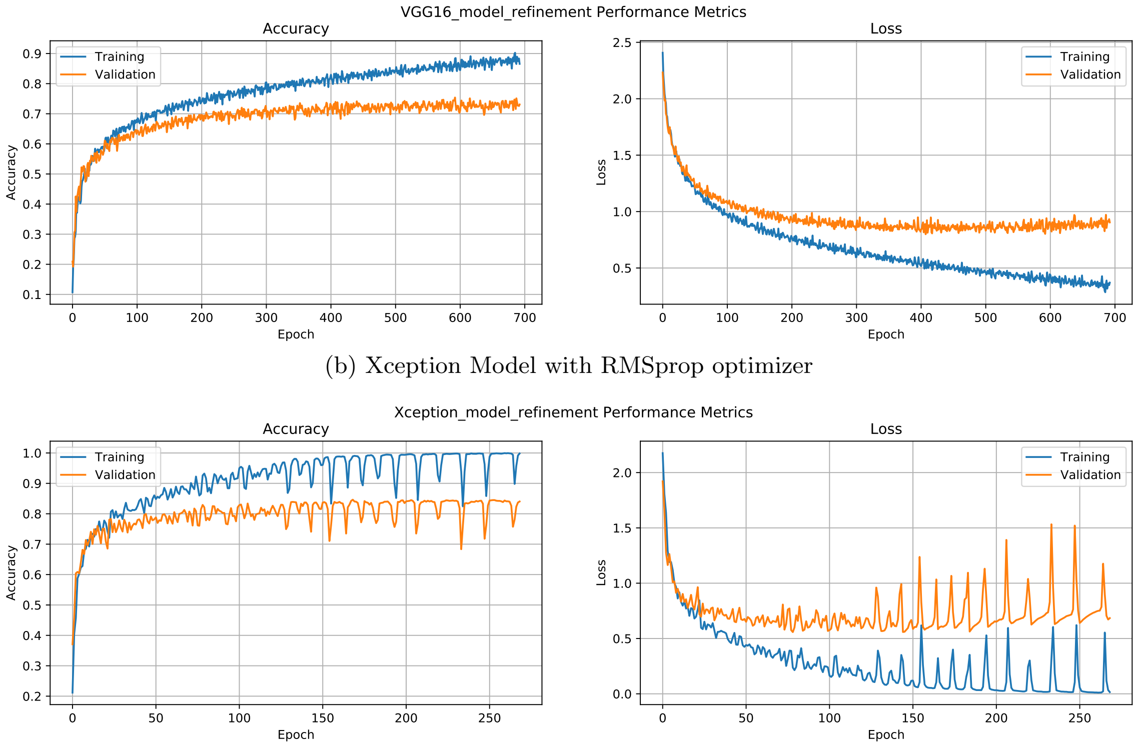
Figure 4: Learning curves of the initial solution with SGD optimizer

(a) VGG16 Model with SGD optimizer



(b) Xception Model with SGD optimizer

Figure 5: Learning curves of the refinement solution with RMSprop optimizer



(a) VGG16 Model with RMSprop optimizer

(b) Xception Model with RMSprop optimizer

4 Results

4.1 Model Evaluation and Validation

During model development, I used a validation set to evaluate the model while training. The validation set accuracy and loss scores were the basis for selecting the best top-model weights.

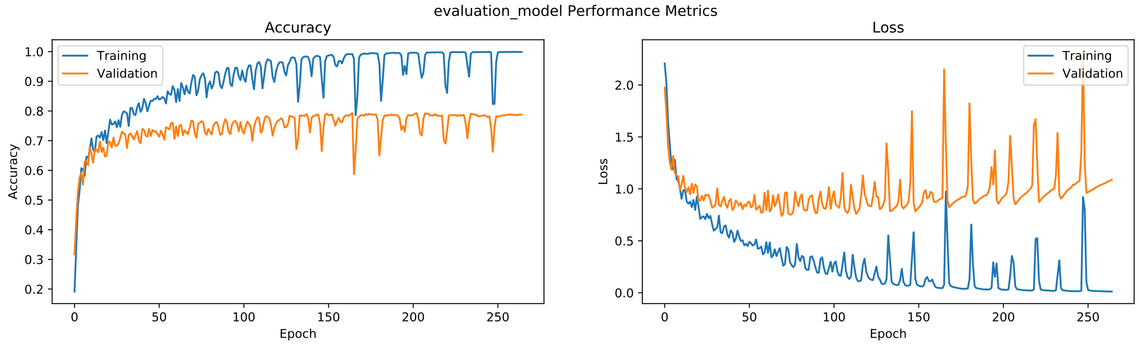
The final solution model chosen is `Xception Model` with the `RMSprop` optimizer. It was selected because it had the best performance on test accuracy and GAP scores (See Table 3). This model's performance is satisfactory given the nature and difficulty of the landmark recognition problem.
The following list describes our final solution model :

- pre-trained Xception model without its top layers (`include_top=False`) and with average pooling mode for feature extraction (`pooling='avg'`)
- The input image size to the model is 240 × 240 pixels.
- A fully connected top model that has three *Dense* layers. For a complete description of the top-model architecture refer to Figure 3b - Xception top-model architecture. For a description of the layers and activation functions refer to step 3 in Implementation.
- `RMSProp` optimizer. with learning rate `lr=0.001`

Sensitivity Analysis

To validate the robustness of the solution model, I tried training the model with a different input image size of 150 × 150. The test loss score was 0.8985, the accuracy score was 77.93%, and the GAP score was 74.18%. The performance is slightly less but not very far from using an input size of 240 × 240. Figure 6 shows the Learning curves of the evaluation model.

Figure 6: Learning curves of the evaluation model



4.2 Justification

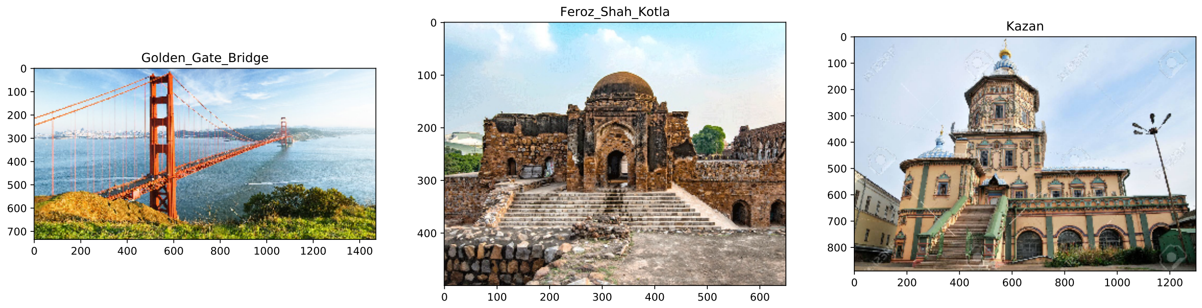
The final solution model (summarized in subsection 4.1 - Model Evaluation and Validation) has 84.91% test accuracy score and 82.68% GAP score. Its performance is significantly higher than the benchmark model (described in subsection 2.4) which had 62.29% test accuracy score and 55.85% GAP score.

5 Conclusion

5.1 Free-Form Visualization

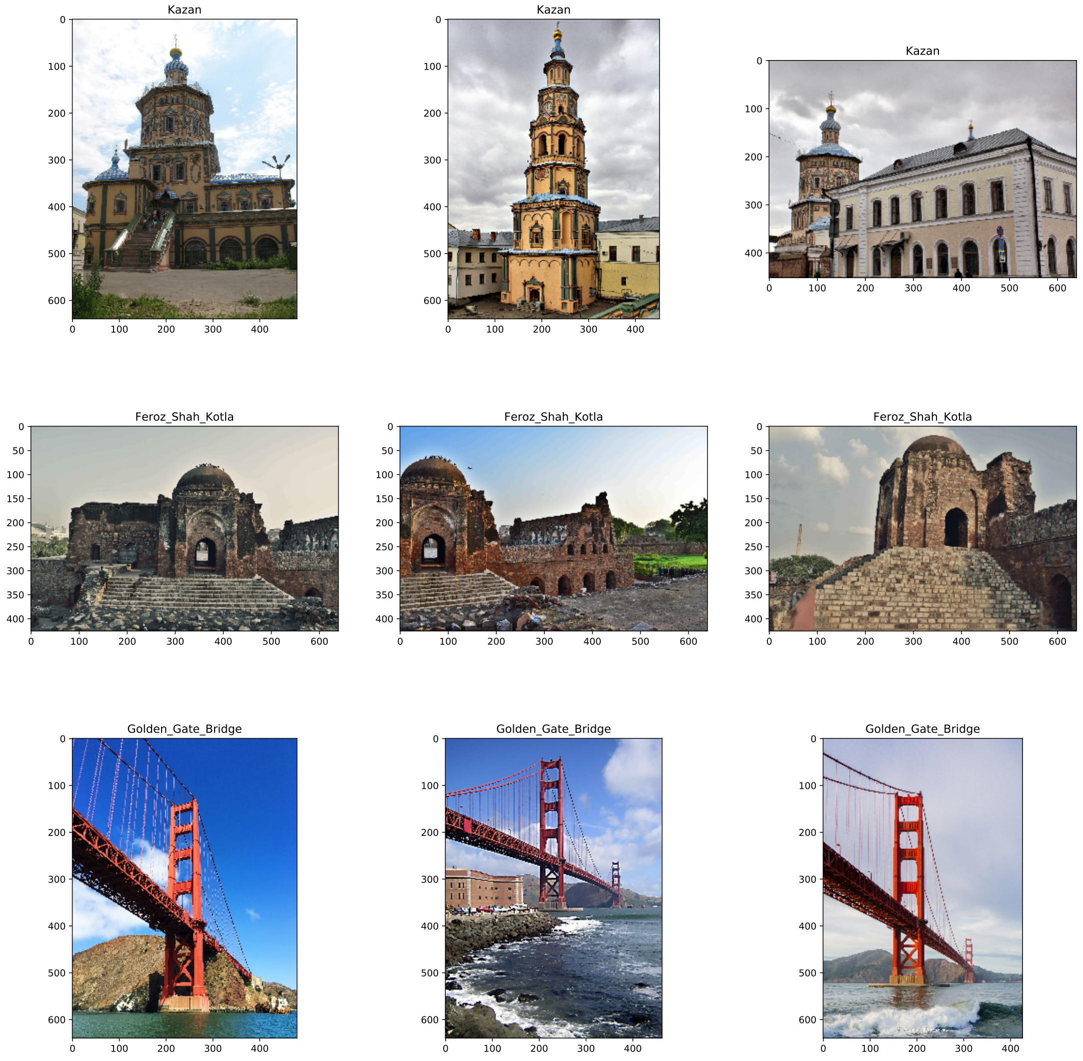
I tested my final solution model on three unseen landmark images randomly downloaded from the web for Jami Masjid in *Feroz Shah Kolta*, Cathedral of the Apostles Peter and Paul in *Kazan* and Golden Gate Bridge and the predictions are shown in Figure 7. It shows that the solution model did a good job classifying the three images correctly.

Figure 7: Predictions of unseen landmark images from the web



Visually similar images from the training dataset that the model was trained on are shown in Figure 8.

Figure 8: Visually similar images from the training dataset



5.2 Reflection

The entire end-to-end problem solution can be summarized as the following :

1. A challenge problem and an available public dataset were found.
2. A suitable metric was found and implemented.
3. The data was downloaded and split into training, validation and testing sets.
4. The data was prepared and preprocessed to be used as input for the classification model.
5. A simple convolutional neural network benchmark model was implemented and tested.
6. Transfer-learning solution models `VGG16` and `Xception` were implemented, refined and tested. The extraction of bottleneck features was done for each model. Then, a solution model was selected based on performance metrics.
7. Sensitivity analysis was conducted on the solution model using a different resolution of input images.
8. The solution model classifier was tested on unseen images downloaded randomly from the web.

An interesting aspect of this project was that transfer-learning models achieved great performance in short training time. Their performance was better than the benchmark model that took a much longer time to train.

The challenging aspect of the project was the selection of classes and the extraction of a balanced subset dataset for the project from the large publicly available dataset. Another challenge was the implementation of the GAP performance metric and obtaining the correct input vectors.

Convolutional neural networks in general and transfer-learning techniques, in particular, are the best for image classification problems (up to current date) as we have seen in this project implementation. They are highly recommended for such problems and similar problems.

5.3 Improvement

For the improvement of our solution model, my suggestions are the following :

1. Allowing the model to train for longer times on more powerful hardware with capable GPUs. That would speed up experimenting with and testing different solution models in shorter times.
2. Another possible improvement is collecting a larger dataset of landmarks, training the model on a larger count of images and augmenting the available data to increase the count of images per each class.
3. In this project, we froze the original pre-trained networks' weights and focused on training the top-model that we created (which took bottleneck features as input). We can try fine-tuning each of the original pre-trained models' weights to achieve a better feature extraction from the input.
4. Proper classification of non-landmark images and the ability to recognize them should be implemented.

References

[CCF16] Jiuwen Cao, Tao Chen, and Jiayuan Fan. "Landmark recognition with compact BoW histogram and ensemble ELM". In: *Multimedia Tools and Applications* 75.5 (Mar. 2016), pp. 2839–2857. ISSN: 1573-7721. DOI: 10.1007/s11042-014-2424-1. URL: https://doi.org/10.1007/s11042-014-2424-1.

[Hin12] Geoffrey Hinton. *Neural Networks for Machine Learning - Lecture 6a - Overview of mini-batch gradient descent.* Lecture 6 of the online course "Neural Networks for Machine Learning" on Coursera. 2012. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[Le +90] Y. Le Cun et al. "Handwritten zip code recognition with multilayer networks". In: *[1990] Proceedings. 10th International Conference on Pattern Recognition.* Vol. ii. June 1990, 35–40 vol.2. DOI: 10.1109/ICPR.1990.119325.

[Noh+17] Hyeonwoo Noh et al. "Large-Scale Image Retrieval with Attentive Deep Local Features". In: Oct. 2017, pp. 3476–3485. DOI: 10.1109/ICCV.2017.374.

[PLR09] F. Perronnin, Y. Liu, and J. Renders. "A family of contextual measures of similarity between distributions with application to image retrieval". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition.* June 2009, pp. 2358–2365. DOI: 10.1109/CVPR.2009.5206505.

[Zhe+09] Yantao Zheng et al. "Tour the World: Building a web-scale landmark recognition engine". In: June 2009, pp. 1085–1092. DOI: 10.1109/CVPRW.2009.5206749.