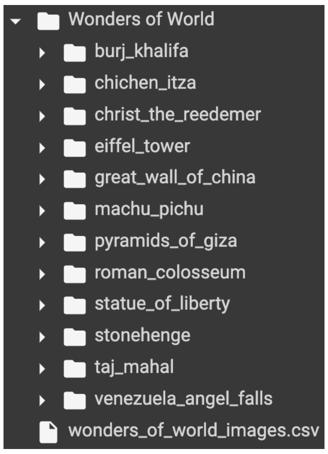
Classifying Wonders of the World Using Transfer Learning

Name: YIN Zhuohao   Student ID: 20677990   Email: zyinad@connect.ust.hk

Dataset

General description of the dataset

The dataset I chose was the Wonders of the World Image Dataset, which contains 12 classes of wonders across the world such as the Statue of Liberty, the Great Wall, etc. The dataset comes with a *wonders_of_world_images.csv* file that summarizes information about the dataset, where the first column is 'folder_name' and the second column is 'file_name'. The images are configured in a way where the data belonging to the same class are stored in one folder, whose name is the name of the class. The folder structure is shown in Fig. 1.

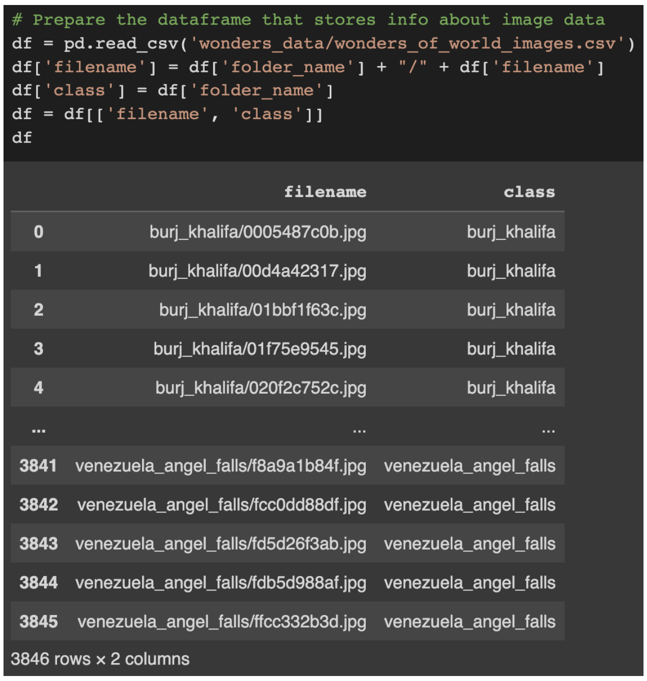


```
Wonders of World
  burj_khalifa
  chichen_itza
  christ_the_reedemer
  eiffel_tower
  great_wall_of_china
  machu_pichu
  pyramids_of_giza
  roman_colosseum
  statue_of_liberty
  stonehenge
  taj_mahal
  venezuela_angel_falls
  wonders_of_world_images.csv
```

Figure 1. File configuration of the the dataset.

Data preprocessing

After reading in the *wonders_of_world_images.csv*, I slightly modified its content for the purpose of loading image data later as shown in Fig. 2. The 'filename' column stores the relative path for each image so that later it is easier for us to load the images.



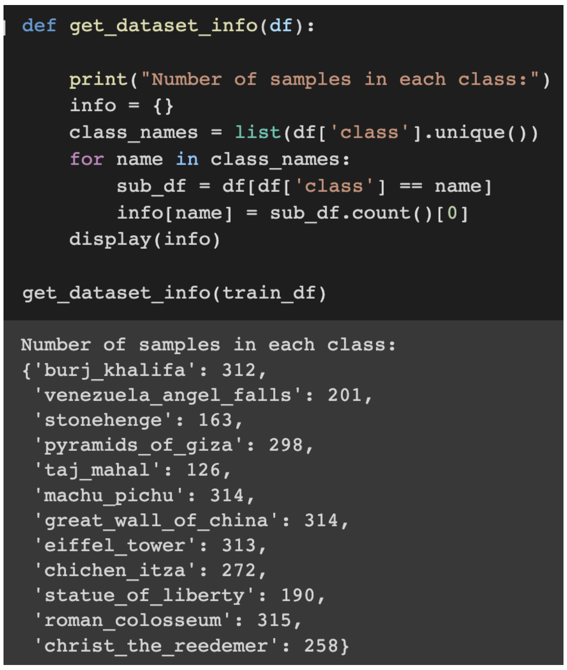
```python
# Prepare the dataframe that stores info about image data
df = pd.read_csv('wonders_data/wonders_of_world_images.csv')
df['filename'] = df['folder_name'] + "/" + df['filename']
df['class'] = df['folder_name']
df = df[['filename', 'class']]
df
```

	filename	class
0	burj_khalifa/0005487c0b.jpg	burj_khalifa
1	burj_khalifa/00d4a42317.jpg	burj_khalifa
2	burj_khalifa/01bbf1f63c.jpg	burj_khalifa
3	burj_khalifa/01f75e9545.jpg	burj_khalifa
4	burj_khalifa/020f2c752c.jpg	burj_khalifa
...	...	...
3841	venezuela_angel_falls/f8a9a1b84f.jpg	venezuela_angel_falls
3842	venezuela_angel_falls/fcc0dd88df.jpg	venezuela_angel_falls
3843	venezuela_angel_falls/fd5d26f3ab.jpg	venezuela_angel_falls
3844	venezuela_angel_falls/fdb5d988af.jpg	venezuela_angel_falls
3845	venezuela_angel_falls/ffcc332b3d.jpg	venezuela_angel_falls

3846 rows × 2 columns

Figure 2. Modifications to *wonders_of_world_images.csv* for loading data later.

Then, train-test split is performed to split the whole piece of DataFrame into 3 parts, `train_df`, `val_df`, `test_df`, at the ratio of 8:1:1. To get an overview of the composition of each DataFrame, I defined the `get_dataset_info(df)` function. Fig. 3 shows the number of images in each class in the training data.

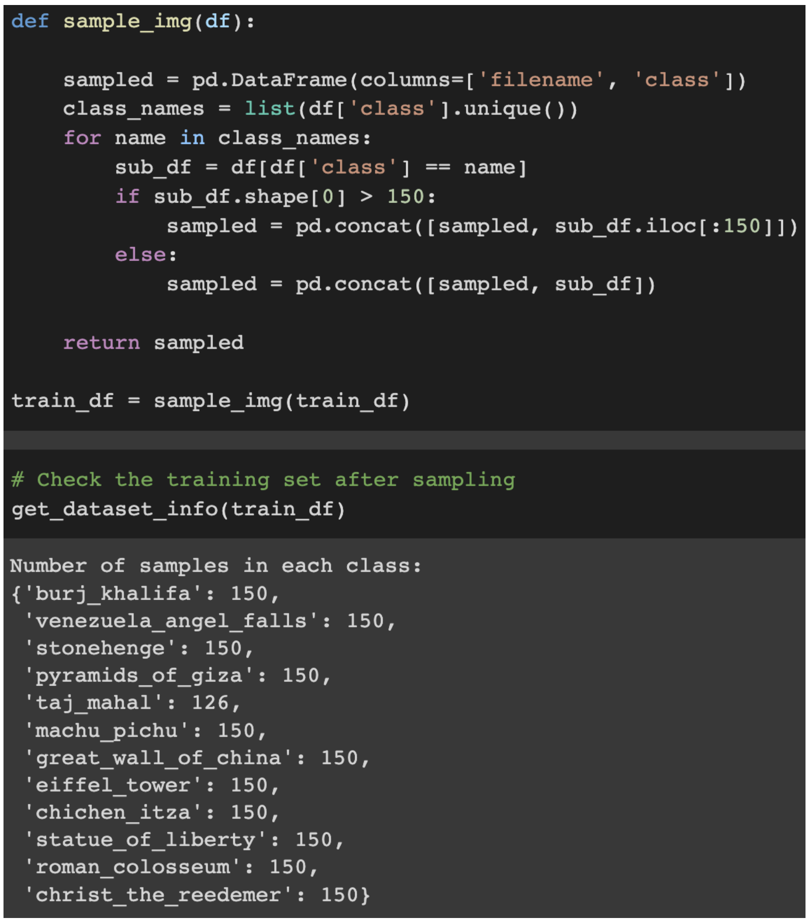


```python
def get_dataset_info(df):

    print("Number of samples in each class:")
    info = {}
    class_names = list(df['class'].unique())
    for name in class_names:
        sub_df = df[df['class'] == name]
        info[name] = sub_df.count()[0]
    display(info)

get_dataset_info(train_df)
```

```
Number of samples in each class:
{'burj_khalifa': 312,
 'venezuela_angel_falls': 201,
 'stonehenge': 163,
 'pyramids_of_giza': 298,
 'taj_mahal': 126,
 'machu_pichu': 314,
 'great_wall_of_china': 314,
 'eiffel_tower': 313,
 'chichen_itza': 272,
 'statue_of_liberty': 190,
 'roman_colosseum': 315,
 'christ_the_reedemer': 258}
```

Figure 3. Number of samples in each class in the training data.

It is noticeable that the number of training examples in each class is imbalanced. When training on the imbalanced data, the classes that have more training samples will have a higher prior probability than others, causing the model inclined to over-classify these classes, while the opposite for the minority classes. Thus, it is of great importance that we try to make the number of samples in each class approximately equal. Without loss of generality, I did this by pruning the sample size of each class down to 150, except for class ‘taj_mahal’ since it has only 126 samples to begin with. After pruning, we can check the overview about training data again and the result is shown in Fig. 4.



```
def sample_img(df):

    sampled = pd.DataFrame(columns=['filename', 'class'])
    class_names = list(df['class'].unique())
    for name in class_names:
        sub_df = df[df['class'] == name]
        if sub_df.shape[0] > 150:
            sampled = pd.concat([sampled, sub_df.iloc[:150]])
        else:
            sampled = pd.concat([sampled, sub_df])

    return sampled

train_df = sample_img(train_df)
```

```
# Check the training set after sampling
get_dataset_info(train_df)
```

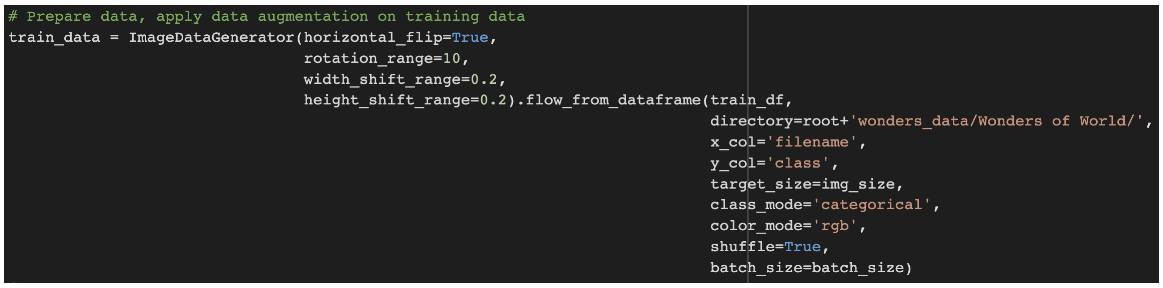
```
Number of samples in each class:
{'burj_khalifa': 150,
 'venezuela_angel_falls': 150,
 'stonehenge': 150,
 'pyramids_of_giza': 150,
 'taj_mahal': 126,
 'machu_pichu': 150,
 'great_wall_of_china': 150,
 'eiffel_tower': 150,
 'chichen_itza': 150,
 'statue_of_liberty': 150,
 'roman_colosseum': 150,
 'christ_the_reedemer': 150}
```

Figure 4. Number of samples in each class in the training data after pruning.

Creating dataset with ImageDataGenerator( )

1. **Training data**

   First we create the training data as shown in Fig. 5:



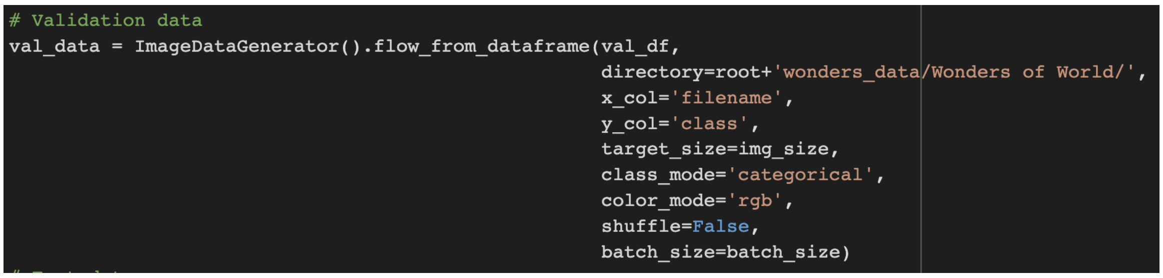
```
# Prepare data, apply data augmentation on training data
train_data = ImageDataGenerator(horizontal_flip=True,
                                rotation_range=10,
                                width_shift_range=0.2,
                                height_shift_range=0.2).flow_from_dataframe(train_df,
                                                                            directory=root+'wonders_data/Wonders of World/',
                                                                            x_col='filename',
                                                                            y_col='class',
                                                                            target_size=img_size,
                                                                            class_mode='categorical',
                                                                            color_mode='rgb',
                                                                            shuffle=True,
                                                                            batch_size=batch_size)
```

Figure 5. Initialize training data.

The `ImageDataGenerator` class enables us to apply data augmentation such as flipping, rotation, and shifting, etc, which is an important regularization technique to prevent overfitting. In my case, I applied horizontal flip, 10-degree maximum rotation, 0.2 maximum width shift and 0.2 maximum height shift. Then the `flow_from_dataframe` function is called to load images on the fly according to the directory and file names in `train_df`.

2. **Validation data**

   The validation data needs no augmentation, so we can directly create it by calling `flow_from_dataframe` as shown in Fig. 6.

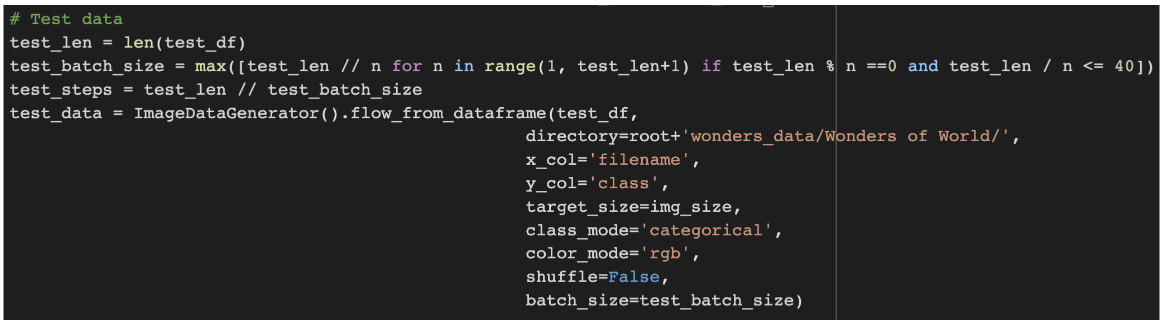


```
# Validation data
val_data = ImageDataGenerator().flow_from_dataframe(val_df,
                                              directory=root+'wonders_data/Wonders of World/',
                                              x_col='filename',
                                              y_col='class',
                                              target_size=img_size,
                                              class_mode='categorical',
                                              color_mode='rgb',
                                              shuffle=False,
                                              batch_size=batch_size)
```

Figure 6. Initialize validation data.

3. **Test data**

   The test data does not need augmentation either. However, to make sure that every test image is evaluated, we must set the batch size to be a number divisible by the size of the test data. I computed the batch size of the test data by listing all divisible factors of the length of the test data. Then, I chose the maximum value with a constraint that it is not larger than 40, which is to prevent a large batch size from consuming the RAM. Details are shown in Fig. 7.



```
# Test data
test_len = len(test_df)
test_batch_size = max([test_len // n for n in range(1, test_len+1) if test_len % n ==0 and test_len / n <= 40])
test_steps = test_len // test_batch_size
test_data = ImageDataGenerator().flow_from_dataframe(test_df,
                                                  directory=root+'wonders_data/Wonders of World/',
                                                  x_col='filename',
                                                  y_col='class',
                                                  target_size=img_size,
                                                  class_mode='categorical',
                                                  color_mode='rgb',
                                                  shuffle=False,
                                                  batch_size=test_batch_size)
```

Figure 7. Initialize the test data.

Viewing training samples and labels

After initializing the data, I defined `show_image_samples` to see sample images and their class labels. The result is as follows.

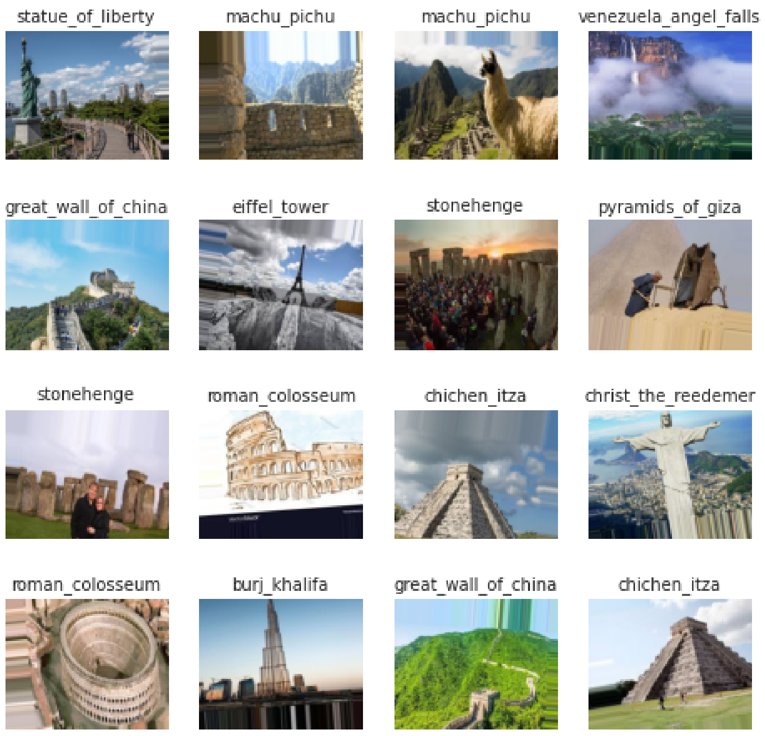


Figure 8. 16 samples of training data, annotated with labels.

Task Description

The task of this project is essentially to perform image classification for the Wonders dataset, which is a common task in the field of computer vision. However, the focus of this project lies not in the model performance, such as accuracy, F1 score, etc., but the technique of transfer learning.

Methodology

Transfer learning is the idea of reusing pre-trained models from a different problem domain. It is exceptionally useful when your problem does not have a huge amount of data. A typical procedure of transfer learning is to use a robust CNN architecture that has been pre-trained on large datasets, such as the ImageNet, as a feature extractor and change only the last softmax layer to output the desired number of class probabilities. It is often the case that all other layers except for the last softmax layer are frozen. In other words, parameters of those layers are set to non-trainable. With more flexibility, we can also stack several fully-connected layers with batch normalization and dropout on top of the backbone architecture as to achieve a larger capacity. These layers on top of the networks are usually called heads.

In this project, I will explore the performance of using different backbone models and heads on the multi-class classification task of Wonders of the World. In terms of implementation, I developed my work primarily based on this Kaggle notebook, and the saliency map implementation was modified from this blog.

Experiments

Model configurations

Since the project aims to explore different combinations of backbone models, heads and training configurations, the following are the variants and their options.

1. **Backbones**

I chose 4 well-known CNN architectures that all achieved outstanding results on the ImageNet.

- ResNet-50

  The ResNet50 model was proposed by He, et al. in 2015, which achieve an error rate of 3.57% on the ImageNet Dataset.

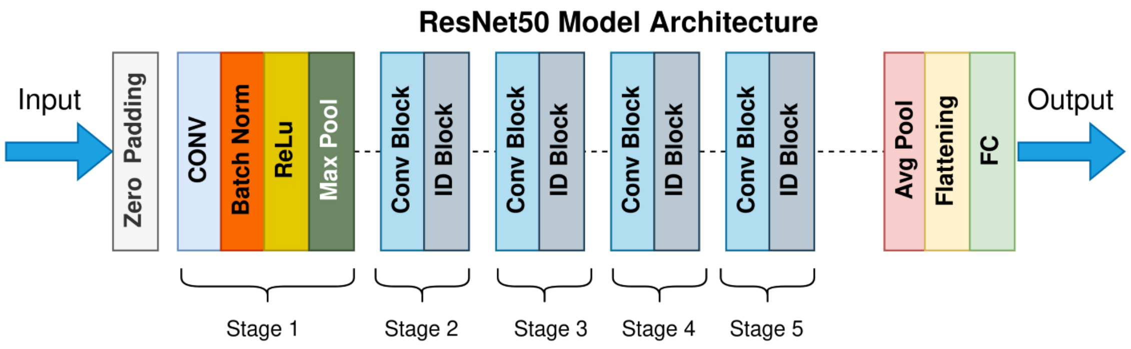


Figure 9. Model architecture of ResNet-50.

- VGG16

  VGG16 was the winner of the ImageNet in 2014. Its architecture is as follows.

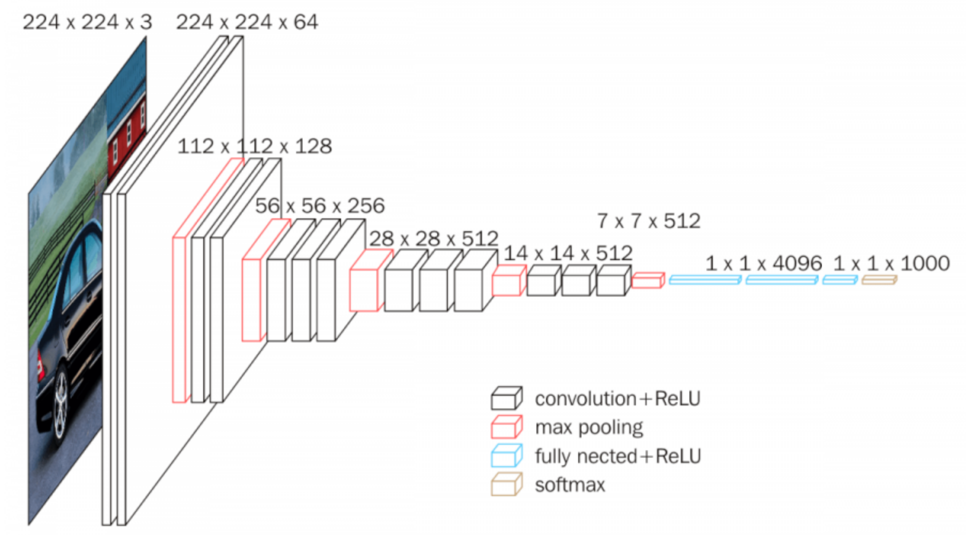


Figure 10. Model architecture of VGG16.

- Inception V3

  Inception v3 is an image recognition model that has been shown to attain greater than 78.1% accuracy on the ImageNet dataset. Its model architecture is shown below.

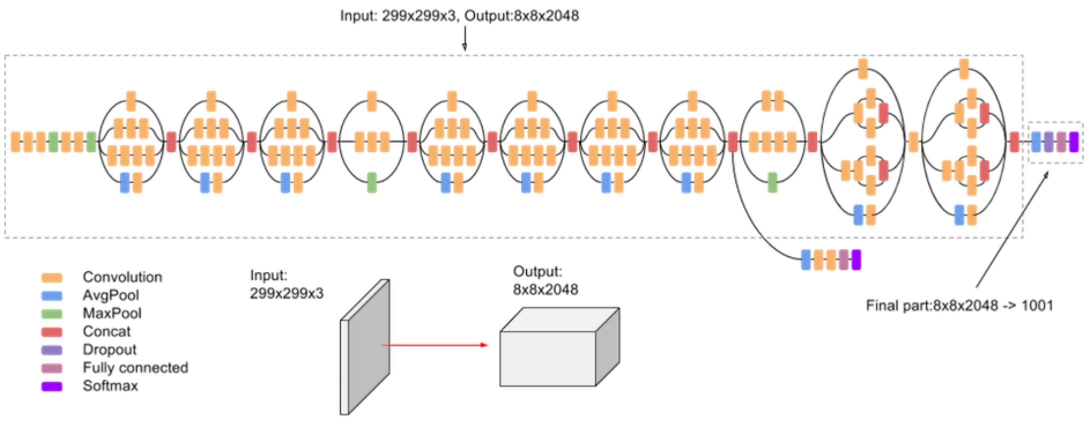


Figure 11. Model architecture of Inception V3.

- EfficientNet B3

EfficientNet uses a compound coefficient to uniformly scales network width, depth, and resolution in a principled way. Its model architecture is shown below.

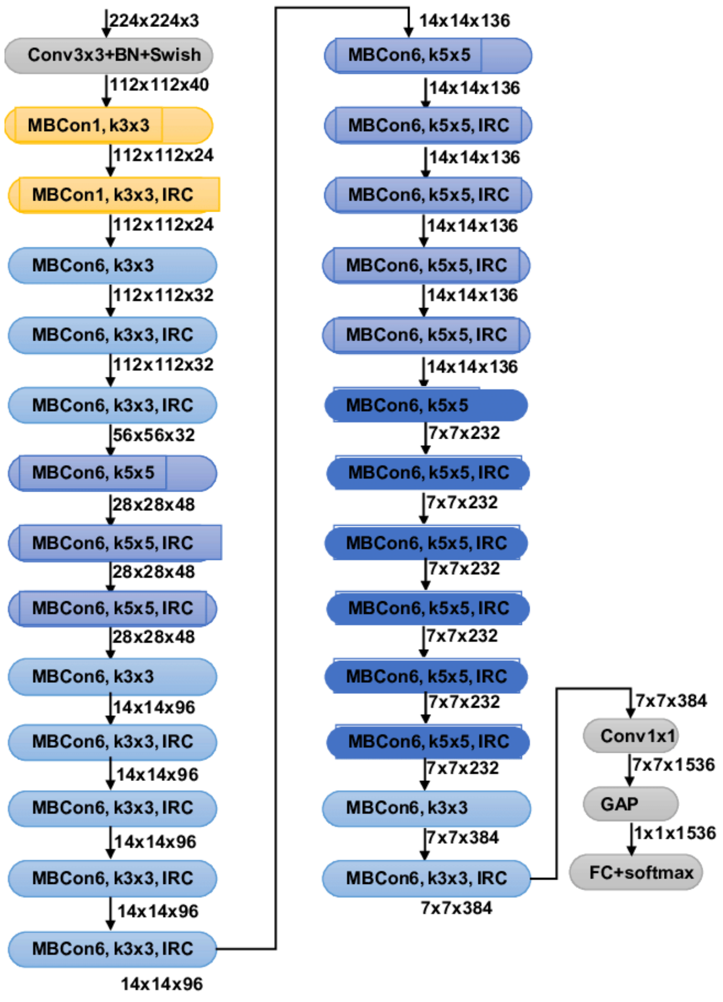
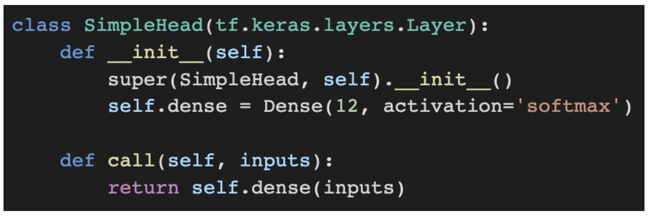


Figure 12. Model architecture of EfficientNet B3.

2. **Heads**

- Simple head

  The `SimpleHead()` class I defined is a keras layer that has only a softmax function.

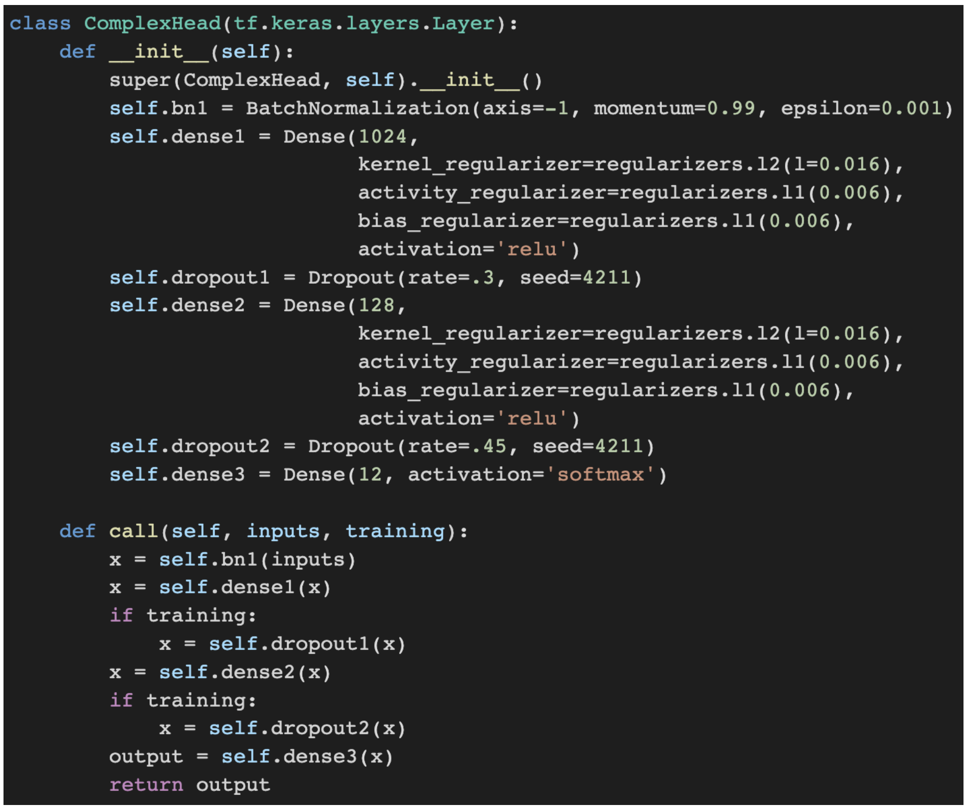


```
class SimpleHead(tf.keras.layers.Layer):
    def __init__(self):
        super(SimpleHead, self).__init__()
        self.dense = Dense(12, activation='softmax')

    def call(self, inputs):
        return self.dense(inputs)
```

Figure 13. Implementation of the simple head.

- Complex head

  The `ComplexHead()` class I defined has more complex components. Details are as follows.



```
class ComplexHead(tf.keras.layers.Layer):
    def __init__(self):
        super(ComplexHead, self).__init__()
        self.bn1 = BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001)
        self.dense1 = Dense(1024,
                            kernel_regularizer=regularizers.l2(l=0.016),
                            activity_regularizer=regularizers.l1(0.006),
                            bias_regularizer=regularizers.l1(0.006),
                            activation='relu')
        self.dropout1 = Dropout(rate=.3, seed=4211)
        self.dense2 = Dense(128,
                            kernel_regularizer=regularizers.l2(l=0.016),
                            activity_regularizer=regularizers.l1(0.006),
                            bias_regularizer=regularizers.l1(0.006),
                            activation='relu')
        self.dropout2 = Dropout(rate=.45, seed=4211)
        self.dense3 = Dense(12, activation='softmax')

    def call(self, inputs, training):
        x = self.bn1(inputs)
        x = self.dense1(x)
        if training:
            x = self.dropout1(x)
        x = self.dense2(x)
        if training:
            x = self.dropout2(x)
        output = self.dense3(x)
        return output
```

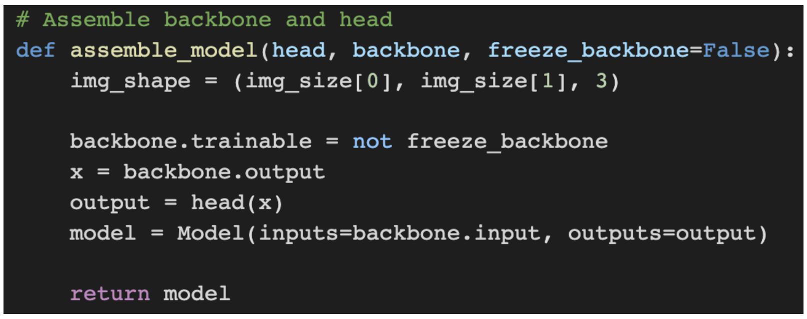
Figure 14. Implementation of the complex head.

3. **Freezing the backbone**

   Although it is a common practice to freeze all layers of the backbone model and just optimize parameters in the head, there are also suggestions saying not freezing them yields a more flexible model and better results. This is why I have set the `freeze_backbone_options = [True, False]` to explore different settings.

Training

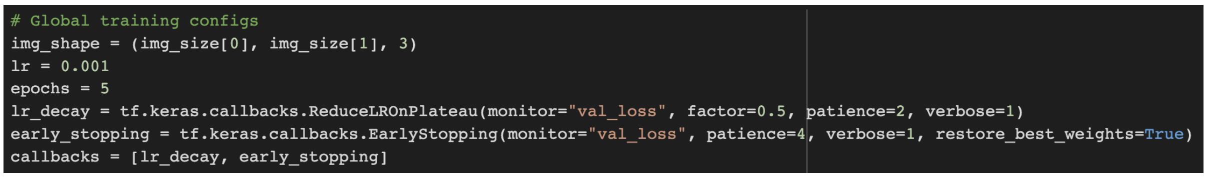
Prior to the training process, the model must be assembled first given a head and a backbone. Notice that the `trainable` attribute of the backbone model is set to the negation of `freeze_backbone`.



```python
# Assemble backbone and head
def assemble_model(head, backbone, freeze_backbone=False):
    img_shape = (img_size[0], img_size[1], 3)

    backbone.trainable = not freeze_backbone
    x = backbone.output
    output = head(x)
    model = Model(inputs=backbone.input, outputs=output)

    return model
```

Figure 15. Implementation of the function that assembles models.

Once the model is assembled, it will be trained using the following training configurations. Specifically, the image size is pre-determined, the learning rate is set to 0.001, and the number of epochs to train is set to 5. Moreover, callbacks are enabled to schedule learning rate decay and early stopping.



```python
# Global training configs
img_shape = (img_size[0], img_size[1], 3)
lr = 0.001
epochs = 5
lr_decay = tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=2, verbose=1)
early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=4, verbose=1, restore_best_weights=True)
callbacks = [lr_decay, early_stopping]
```

Figure 16. Training configurations.

In favor of conciseness, the implementation of the train and test function will not be shown here.

During training, the training loss, training accuracy, validation loss, and validation accuracy are monitored. After training, the loss curve and the validation curve are produced. An example is shown below.

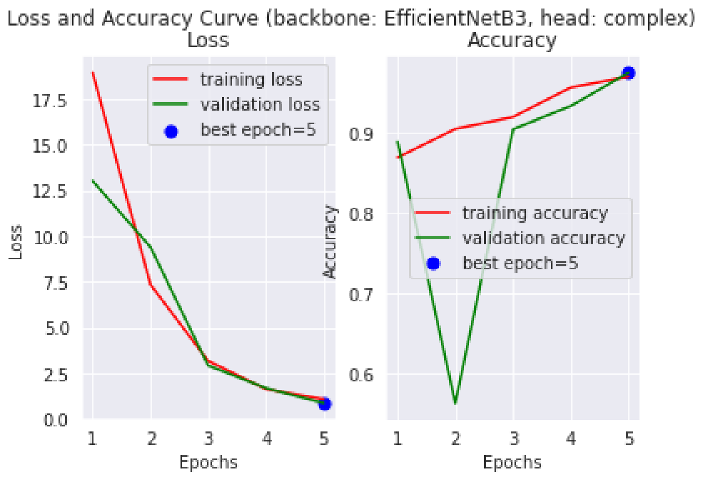


Figure 17. An example of the loss and accuracy curve.

Evaluation and comparison

- Evaluation metrics

As for evaluation metrics, I included accuracy, as well as the confusion matrix. The accuracy results are presented in the tables below. However, the confusion matrices are produced as figures, and only an example is shown here. The full results are available in the outputs of the Jupyter notebook.

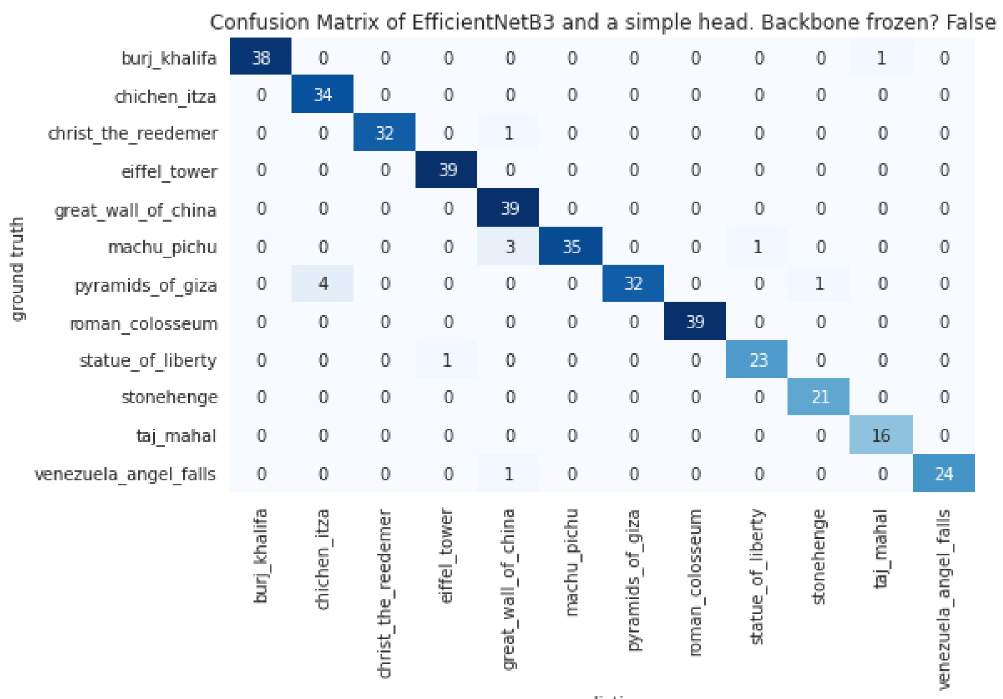


Figure 18. An example of the confusion matrix.

- Results and interpretations

My experiments trained and evaluated 16 models that have different structures or configurations. The results can be summarized in the following table.

Backbone	Head	Backbone frozen?	Best training accuracy	Test accuracy
ResNet50	simple	True	0.8829	0.9065
ResNet50	simple	False	0.7866	0.7351
ResNet50	complex	True	0.7827	0.8078
ResNet50	complex	False	0.6025	0.3325
VGG16	simple	True	0.7748	0.8260
VGG16	simple	False	0.0952	0.1299
VGG16	complex	True	0.1154	0.1117
VGG16	complex	False	0.0805	0.0649
InceptionV3	simple	True	0.2872	0.2961
InceptionV3	simple	False	0.9229	0.8234
InceptionV3	complex	True	0.8891	0.8779
InceptionV3	complex	False	0.8243	0.4026
EfficientNetB3	simple	True	0.8581	0.9091
EfficientNetB3	simple	False	0.9792	0.9662
EfficientNetB3	complex	True	0.9859	0.9766
EfficientNetB3	complex	False	0.9690	0.9299

The results are quite interesting. It is noticeable that the model performance tends to be better when backbones are frozen, namely not updated. My interpretation of this phenomenon is that deeper networks are harder to optimize, and thus it is unlikely for them to achieve satisfactory performance within only 5 epochs. It can also be seen that simple heads tend to work better than complex heads. This may seem counter-intuitive as complex models should have a better capacity. I proposed 2 possible explanations for it:

1. Deeper networks are harder to optimize. I wonder whether the performance will be boosted with more epochs of training for these rather complex models.
2. Not freezing backbone layers cause damage on the information or features extracted during the pre-training phase, thus a lower accuracy.

To verify my interpretation, I decided to conduct more experiments to test the hypothesis. I then trained two models for 10 epochs using ResNet50 as backbone, simple heads, and one freezes the backbone while the other does not. The results are as follows.

Backbone	Epochs	Head	Backbone frozen?	Best training accuracy	Test accuracy
ResNet50	5	simple	True	0.8829	0.9065
ResNet50	5	simple	False	0.7866	0.7351
ResNet50	10	simple	True	0.9572	0.9195
ResNet50	10	simple	False	0.9499	0.9013

The results show that even though we train for more epochs, the model with backbone frozen still outperforms that with backbone not frozen, which is a sign that the poor performance was not a result of lack of training epochs. Instead, the reason we should freeze the backbone layers is that unfreezing the whole architecture may lead to potential corruption of the features that have already been extracted. Corrupted features intuitively yield worse classification results.

- Visualization: Saliency maps

Saliency maps are widely used for network visualizations and interpretations. Its mechanism is to compute the output class scores's gradient with respect to each input pixel. A larger gradient signifies greater contribution to the class scores, which are typically the critical factors based on which the model made its classification decision. I define the `saliency_map` function that takes in a model and a path of an image and outputs the prediction result, the original image, and its saliency map. Fig. 19 (a) & (b) show two examples. We can see that in (a), the subject pixels, those of the Christ statue, have higher gradients, meaning that they contribute more to the final classification decision. Similarly in (b), the Great Wall pixels have higher gradients than the pixels of the greenery.

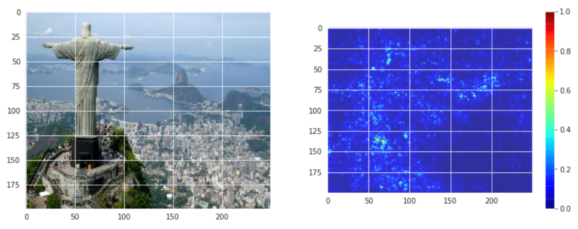


Figure 19 (a). The saliency map for an example in class christ_the_reedemer.

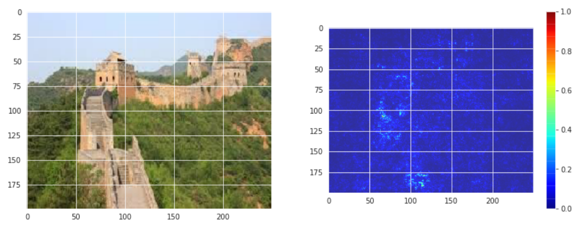


Figure 19 (b). The saliency map for an example in class great_wall_of_china.

Conclusion

In this project, I have explored a handful of models using different backbones and heads, as well as the option to freeze the backbone parameters or not. Among all models, the best test accuracy reached 0.9766. The model achieved this result has **EfficientNetB3** as the backbone, and a **complex** head, with backbone **frozen** and trained for 5 epochs. Experiments showed that for most combinations, freezing the backbone parameters yielded a better result when keeping all other variants the same. My proposed explanation for this is that unfreezing the whole architecture induces risks of corrupting the information or features captured by the backbone layers, leading to a poor performance. Apart from exploring this topic, I also produced saliency maps for one of the models so that I can better interpret the networks.