Swinburne University of Technology

COS30082 - Applied Machine Learning

Assignment: Bird Species Classification

Nguyen Anh Tuan

103805949

Hanoi, Vietnam

A. Introduction

The report is included in the assignment “ Bird Species Classification” which uses the Caltech-UCSD Birds 200 (CUB-200) dataset to create a model for classifying bird species. In this assignment, I use MobileNetV2 architecture and transfer learning to classify bird species accurately. Various strategies were employed during the assignment to reduce overfitting, including regularization and data augmentation. The methodology, experimental setting, and assessment metrics are covered in this report, which also offers insights into the model's performance and difficulties that were faced along the way.

B. Methodology

1. Dataset

Using the load_data function, training and testing data are loaded from text files to create the dataset. With columns like "filename" and "class_name," each dataset is kept in a pandas DataFrame that associates photos with the corresponding labels.

```
[22] def load_data(file_path):
         with open(file_path, 'r') as file:
             return file.read().splitlines()

     train_data = load_data(train_txt_dir)
     test_data = load_data(test_txt_dir)

     training_feature = pd.DataFrame([line.split() for line in train_data], columns=['filename', 'class_name'])
     test_feature = pd.DataFrame([line.split() for line in test_data], columns=['filename', 'class_name'])
```

To improve model resilience, an ImageDataGenerator augments the training data by applying transformations including rotation, shearing, and rescaling. On the other hand, a more straightforward generator is used to generate the testing data, which merely rescales the photos to make sure they are the right size for assessment while preserving their original arrangement. Images in both datasets are resized to 224x224 pixels and a batch size of 32 is used.

```
BATCH_SIZE = 32
IMG_SIZE = (224, 224)

train_image_generator = ImageDataGenerator(
    rescale = 1.0/255.0,
    shear_range = 0.15,
    zoom_range = 0.15,
    horizontal_flip = False,
    rotation_range = 30,
)


train_dataset = train_image_generator.flow_from_dataframe(
    training_feature,
    directory = train_dir,
    x_col='filename',
    y_col='class_name',
    target_size = IMG_SIZE,
    batch_size = BATCH_SIZE,
    class_mode = 'categorical'
)
```

```
train_dataset = train_image_generator.flow_from_dataframe(
    training_feature,
    directory = train_dir,
    x_col='filename',
    y_col='class_name',
    target_size = IMG_SIZE,
    batch_size = BATCH_SIZE,
    class_mode = 'categorical'
)


test_image_generator = ImageDataGenerator(rescale=1.0/255.0)

test_dataset = test_image_generator.flow_from_dataframe(
    test_feature,
    directory = test_dir,
    x_col = 'filename',
    y_col = 'class_name',
    target_size = IMG_SIZE,
    batch_size = BATCH_SIZE,
    shuffle = False,
    class_mode = 'categorical'
)
```

2. Base model selection and configuration

a. MobileNetV2 model

MobileNetV2 was selected for this project because of its effectiveness and robust feature extraction performance, especially for tasks involving large image datasets such as bird species classification. MobileNetV2, pre-trained on ImageNet, leverages rich, generalized features to offer a strong basis for transfer learning. Its lightweight design is perfect for areas with limited resources and mobility, allowing for rapid model customization without compromising accuracy. Excluding the top layers (include_top=False)

allows the model to be optimized for particular jobs, providing computational efficiency and flexibility for this classification task.

b. Transfer learning:

Transfer learning is implemented by leveraging the pre-trained MobileNetV2 model, which has already learned visual features from the extensive ImageNet dataset. Through transfer learning, these previously acquired patterns can be reused and modified for the bird species categorization job rather than starting from scratch. The deeper layers of the model are adjusted to learn task-specific features, while the earlier layers are frozen to preserve the pre-trained weights. By expanding on strong, previously learned features, this method improves accuracy, saves time, and uses less computing power.

3. Fine-tuning strategy

The pre-trained MobileNetV2 model is modified for the classification of bird species in this fine-tuning approach. The earlier layers of the model are frozen up to the 100th layer, preserving the pre-learned ImageNet weights. By unfreezing layers beyond the 100th, the model can be trained to acquire task-specific attributes. To improve accuracy and performance, the deeper layers must be fine-tuned so that the model can adapt to the new dataset while maintaining the general visual patterns from ImageNet. The pre-trained weights are not disturbed by the low base learning rate, which guarantees progressive updates.

```
[ ] base_model.trainable = True
    fine_tune_at = 100  # Unfreeze from the 100th layer onward
    for layer in base_model.layers[:fine_tune_at]:
        layer.trainable = False
```

4. Output layers

The model's output layer was modified to accommodate the multi-class classification requirement of identifying bird species. The architecture, which used the pre-trained MobileNetV2 as the foundation model, has a GlobalAveragePooling2D layer, a Dense layer with 1024 units, and a ReLU activation function. A Dropout layer with a rate of 0.5 and Batch Normalization was used to prevent

overfitting. A Dense layer with 200 units and a softmax activation function made up the final output layer, which allowed the model to generate class probabilities.

```
[7] # Build the model
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(1024, activation = 'relu', kernel_regularizer=l2(0.001))(x) # Use regulizer to avoid overfitting
    x = BatchNormalization()(x)
    x = Dropout(0.5)(x)
    prediction_layer = Dense(200, activation='softmax')(x)
    model = Model(inputs = base_model.input, outputs=prediction_layer)
```

5. Loss function

The model is appropriate for multi-class classification applications and an Adam optimizer with a 0.001 initial learning rate since it employs Cross Entropy Loss as the loss function. The difference between the supposed probability and the actual labels is measured by this loss function. The model is incentivized to improve its predictions over time by penalizing it more for confidently inaccurate predictions (i.e., assigning high probabilities to incorrect classifications).

```
[10] base_learning_rate = 0.0001
     model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate = base_learning_rate),
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])
```

6. Optimizer and learning rate

Due to its flexible learning rate properties, the Adam optimizer is used in this assignment. To control the amount that the model weights are updated during training, the base learning rate is set to 0.0001. When fine-tuning a pre-trained model like MobileNetV2, this relatively modest learning rate is frequently selected to guarantee consistent convergence.

```
[10] base_learning_rate = 0.0001
     model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate = base_learning_rate),
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])
```

7. Handling Overfitting Issue

During the training and architecture phases, many strategies were used to handle the model's overfitting issue. Once the dense layer with 1024 units was in place, a Dropout layer with a rate of 0.5 was attached. During training, this layer randomly removes half of the neurons, preventing the model from becoming unduly dependent on particular features and promoting the learning of more resilient representations.

Furthermore, batch normalization was used to normalize the dense layer's outputs, which helped to maintain a uniform distribution of activations and accelerated convergence by stabilizing the learning process. To further reduce overfitting by discouraging complex models, a kernel regularizer (L2 regularization) was used on the dense layer, imposing a penalty for excessive weights during training.

```
[7] # Build the model
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(1024, activation = 'relu', kernel_regularizer=l2(0.001))(x) # Use regulizer to avoid overfitting
    x = BatchNormalization()(x)
    x = Dropout(0.5)(x)
    prediction_layer = Dense(200, activation='softmax')(x)
    model = Model(inputs = base_model.input, outputs=prediction_layer)
```

Finally, the pre-trained MobileNetV2 model was fine-tuned, beginning with the 100th layer. This method minimized the possibility of overfitting to the new dataset while allowing the model to utilize previously learned features because the generalized weights of the earlier layers were kept frozen. By integrating both techniques, the model successfully struck a compromise between retaining generalization to new data and learning from the training set.

```
[ ] base_model.trainable = True
    fine_tune_at = 100  # Unfreeze from the 100th layer onward
    for layer in base_model.layers[:fine_tune_at]:
        layer.trainable = False
```

C. Result and Discussion

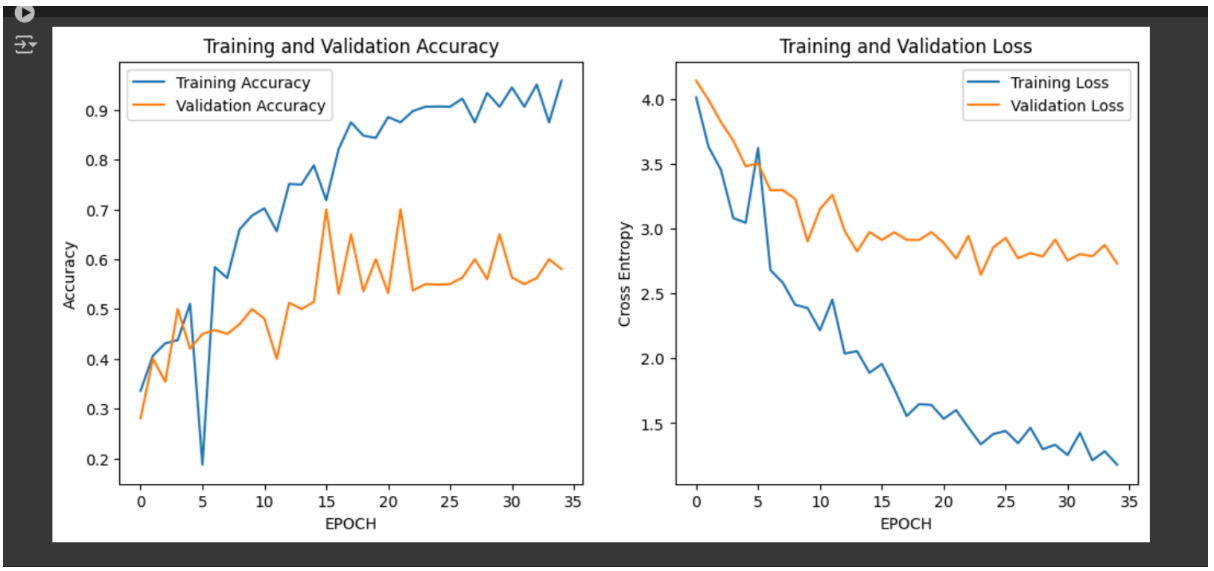
To get the result, I train the model with 35 epochs.

```
Epoch 1/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 524s 3s/step - accuracy: 0.3319 - loss: 4.0681 - val_accuracy: 0.2804 - val_loss: 4.1381
Epoch 2/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.4062 - loss: 3.6294 - val_accuracy: 0.4000 - val_loss: 3.9903
Epoch 3/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 553s 3s/step - accuracy: 0.4151 - loss: 3.4944 - val_accuracy: 0.3539 - val_loss: 3.8206
Epoch 4/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 25s 152ms/step - accuracy: 0.4375 - loss: 3.0809 - val_accuracy: 0.5000 - val_loss: 3.6762
Epoch 5/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 536s 3s/step - accuracy: 0.5078 - loss: 3.0442 - val_accuracy: 0.4206 - val_loss: 3.4787
Epoch 6/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.1875 - loss: 3.6194 - val_accuracy: 0.4500 - val_loss: 3.5008
Epoch 7/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 514s 3s/step - accuracy: 0.5879 - loss: 2.6695 - val_accuracy: 0.4578 - val_loss: 3.2950
Epoch 8/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 24s 150ms/step - accuracy: 0.5625 - loss: 2.5824 - val_accuracy: 0.4500 - val_loss: 3.2959
Epoch 9/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 513s 3s/step - accuracy: 0.6696 - loss: 2.4125 - val_accuracy: 0.4696 - val_loss: 3.2275
Epoch 10/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.6875 - loss: 2.3878 - val_accuracy: 0.5000 - val_loss: 2.9002
Epoch 11/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 558s 3s/step - accuracy: 0.7151 - loss: 2.1820 - val_accuracy: 0.4806 - val_loss: 3.1500
Epoch 12/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.6562 - loss: 2.4518 - val_accuracy: 0.4000 - val_loss: 3.2594
Epoch 13/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 552s 3s/step - accuracy: 0.7664 - loss: 1.9978 - val_accuracy: 0.5127 - val_loss: 2.9806
Epoch 14/35
```

```
Epoch 14/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.7500 - loss: 2.0548 - val_accuracy: 0.5000 - val_loss: 2.8236
Epoch 15/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 508s 3s/step - accuracy: 0.7907 - loss: 1.8867 - val_accuracy: 0.5144 - val_loss: 2.9726
Epoch 16/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.7188 - loss: 1.9566 - val_accuracy: 0.7000 - val_loss: 2.9121
Epoch 17/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 563s 3s/step - accuracy: 0.8192 - loss: 1.7622 - val_accuracy: 0.5304 - val_loss: 2.9704
Epoch 18/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.8750 - loss: 1.5554 - val_accuracy: 0.6500 - val_loss: 2.9128
Epoch 19/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 553s 3s/step - accuracy: 0.8492 - loss: 1.6304 - val_accuracy: 0.5355 - val_loss: 2.9117
Epoch 20/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.8438 - loss: 1.6411 - val_accuracy: 0.6000 - val_loss: 2.9725
Epoch 21/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 554s 3s/step - accuracy: 0.8845 - loss: 1.5290 - val_accuracy: 0.5321 - val_loss: 2.8881
Epoch 22/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.8750 - loss: 1.6011 - val_accuracy: 0.7000 - val_loss: 2.7686
Epoch 23/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 516s 3s/step - accuracy: 0.9005 - loss: 1.4615 - val_accuracy: 0.5372 - val_loss: 2.9439
Epoch 24/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.9062 - loss: 1.3377 - val_accuracy: 0.5500 - val_loss: 2.6443
Epoch 25/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 555s 3s/step - accuracy: 0.9162 - loss: 1.3952 - val_accuracy: 0.5490 - val_loss: 2.8550
Epoch 26/35
```

```
[ ] 150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.8438 - loss: 1.6411 - val_accuracy: 0.6000 - val_loss: 2.9725
Epoch 21/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 554s 3s/step - accuracy: 0.8845 - loss: 1.5290 - val_accuracy: 0.5321 - val_loss: 2.8881
Epoch 22/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.8750 - loss: 1.6011 - val_accuracy: 0.7000 - val_loss: 2.7686
Epoch 23/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 516s 3s/step - accuracy: 0.9005 - loss: 1.4615 - val_accuracy: 0.5372 - val_loss: 2.9439
Epoch 24/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.9062 - loss: 1.3377 - val_accuracy: 0.5500 - val_loss: 2.6443
Epoch 25/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 555s 3s/step - accuracy: 0.9162 - loss: 1.3952 - val_accuracy: 0.5490 - val_loss: 2.8550
Epoch 26/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9062 - loss: 1.4402 - val_accuracy: 0.5500 - val_loss: 2.9272
Epoch 27/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 508s 3s/step - accuracy: 0.9224 - loss: 1.3378 - val_accuracy: 0.5633 - val_loss: 2.7707
Epoch 28/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 21s 128ms/step - accuracy: 0.8750 - loss: 1.4649 - val_accuracy: 0.6000 - val_loss: 2.8103
Epoch 29/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 504s 3s/step - accuracy: 0.9387 - loss: 1.3002 - val_accuracy: 0.5600 - val_loss: 2.7840
Epoch 30/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9062 - loss: 1.3342 - val_accuracy: 0.6500 - val_loss: 2.9145
Epoch 31/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 565s 3s/step - accuracy: 0.9445 - loss: 1.2553 - val_accuracy: 0.5633 - val_loss: 2.7533
Epoch 32/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9062 - loss: 1.4255 - val_accuracy: 0.5500 - val_loss: 2.8035
Epoch 33/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 560s 3s/step - accuracy: 0.9548 - loss: 1.2102 - val_accuracy: 0.5625 - val_loss: 2.7857
Epoch 34/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.8750 - loss: 1.2841 - val_accuracy: 0.6000 - val_loss: 2.8733
Epoch 35/35
150/150 ━━━━━━━━━━━━━━━━━━━━ 559s 3s/step - accuracy: 0.9592 - loss: 1.1760 - val_accuracy: 0.5802 - val_loss: 2.7305
```

Then, I visualize the result for a better overview.



I also use the Sklearn accuracy score and confusion matrix to determine the top 1 accuracy and average accuracy.

```
[ ] from sklearn.metrics import accuracy_score

    prediction = model.predict(test_dataset) # Make predictions on the test data
    predicted_labels = np.argmax(prediction, axis=1) # Convert one-hot encoded predictions to class labels
    true_labels = test_dataset.classes # Get the true labels from the test generator
    top1_acc = accuracy_score(true_labels, predicted_labels) # Calculate the Top-1 accuracy

/usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should
  self._warn_if_super_not_called()
38/38 ━━━━━━━━━━━━━━━━━━━━ 64s 2s/step

[ ] from sklearn.metrics import confusion_matrix

    cf_matrix = confusion_matrix(true_labels, predicted_labels) # Calculate confusion matrix
    class_acc = np.diag(cf_matrix) / cf_matrix.sum(axis=1)
    avg_acc_per_class = np.mean(class_acc) # Calculate the average accuracy per class

[ ] print("Top-1 Accuracy: {:.2f}%".format(top1_acc * 100))
    print("Average Accuracy per Class: {:.2f}%".format(avg_acc_per_class * 100))

Top-1 Accuracy: 58.14%
Average Accuracy per Class: 57.39%
```

The MobileNetV2 bird species classification model shows a significant variation in performance between training and validation. After 35 epochs, the training accuracy surpasses 90%, but the validation accuracy stays in the range of 60% to 70%. This suggests overfitting since it shows that the model achieves good results on the training data but has trouble generalizing to new data.
The loss curves also have the same problem. During the training process, the validation loss first fell before stabilizing, whereas the training loss decreased steadily. The model's continuous improvement of the training data without matching improvements on

the validation set is another characteristic that supports the existence of overfitting.
Despite applying methods, the overfitting issue persists. This happens because of the limited datasets which cause the model to memorize training examples and prevent it from generalizing.
To improve the performance, some measures may be considered such as increasing regularization strength or adding more dropout layers, implementing techniques like focal loss or class weighting to address potential class imbalance.
In summary, the model's training accuracy indicates potential, but it still needs work to increase its generalization capabilities. Overfitting may be lessened and overall performance improved with more testing of hyperparameters, architecture changes, and data handling strategies.

D. Reference

1. **Source code:**
   https://colab.research.google.com/drive/1rYB2kVp2wvQ56c7rh4-KmHxv5d02b55t?usp=sharing
2. **Other reference**
   Transfer learning and fine-tuning:
   https://www.tensorflow.org/tutorials/images/transfer_learning

   How to Avoid Overfitting in Machine Learning?:
   https://www.geeksforgeeks.org/how-to-avoid-overfitting-in-machine-learning/
   Overfit and underfit:
   https://www.tensorflow.org/tutorials/keras/overfit_and_underfit
   What is Batch Normalization in CNN?:
   https://www.geeksforgeeks.org/what-is-batch-normalization-in-cnn/
   Metrics and scoring: quantifying the quality of predictions:
   https://scikit-learn.org/1.5/modules/model_evaluation.html

   What is the Dropout Layer?:
   https://databasecamp.de/en/ml/dropout-layer-en