COS30082 - Applied Machine Learning

Week 5 - Lab - CNN

Huynh Trung Chien - 104848770

1. Screenshots of the three images along with their predicted classes by my model

The below photos show the predictions of my two models for these images. Beside predicting on these three, I collected 10 images belong to 10 classes and all the predictions are correct. Moreover, I also prepared a picture of a flower, which is not in the CIFAR-10 and the prediction is wrong.

Figure 1: Predictions for airplane

Figure 2: Predictions for automobiles

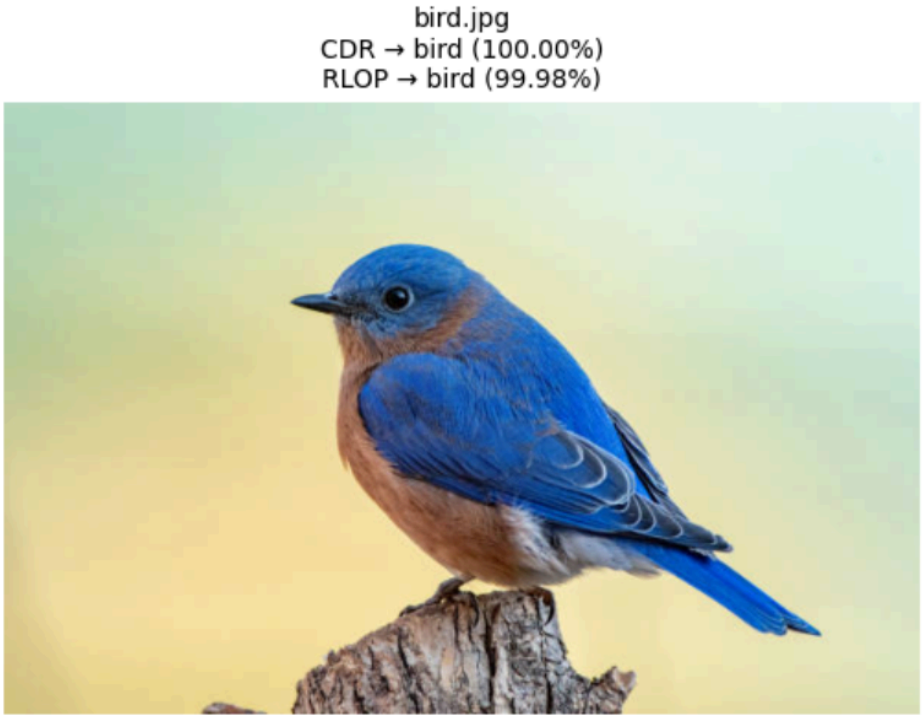

Figure 3: Predictions for bird

Figure 4: Wrong prediction for image not in CIFAR-10

2. A screenshot or description of the changes you made to your CNN and the resulting accuracy

In this lab, I created three CNN models, including my own simple model, and two ResNet models with two different learning rate decay methods. I will go through all the steps that I have done from data preprocessing, train-test split, model training, and results for each model.

a. Data Preprocessing

In order to preprocess data for training, I went through the Data augmentation Tutorial from TensorFlow (https://www.tensorflow.org/tutorials/images/data_augmentation). For the CIFAR-10 dataset, I used to preprocessing steps (rescaling and augmentation). In terms of the augmentation layer, I had tried many other augmentation functions but the results are not positive, so my choices are just to choose RandomFlip, RandomRotation, and RandomZoom with specific values (Figure 6). Those augmentation methods are simple but very effective and practical when preprocessing data.

```
# Create resizing and rescaling using tensorflow
rescale = tf.keras.Sequential([
    tf.keras.layers.Rescaling(1./255)
])
```

Figure 5: Creating Rescaling layer using TensorFlow

```python
# Create a data augmentation layer
data_augmentation = Sequential(
    [
        Input(shape=(32, 32, 3)),
        RandomFlip("horizontal"),
        RandomRotation(0.05),
        RandomZoom(0.05),
    ]
)
```

Figure 6: Create augmentation layer using TensorFlow.

b. Train/Test split

For building the input pipeline, I went through another tutorial from TensorFlow (https://www.tensorflow.org/guide/data). This tutorial assists me in building a standard pipeline for fast, smooth TensorFlow training, with the use of *from_tensor_slices()*, *cache()*, *map()*, *prefetch()* and *tf.data.AUTOTUNE*.

```python
# Using df.data.AUTOTUNE for faster training time and efficiency
AUTOTUNE = tf.data.AUTOTUNE


# Create train and test datasets, creating a simple preprocessing pipeline
train_dataset = (
    tf.data.Dataset.from_tensor_slices((train_images, train_labels))
    .cache()
    .shuffle(10000)
    .map(lambda x, y: (rescale(x), y))
    .batch(batch_size)
    .prefetch(AUTOTUNE)
)


# Test dataset
test_dataset = (
    tf.data.Dataset.from_tensor_slices((test_images, test_labels))
    .cache()
    .map(lambda x, y: (rescale(x), y))
    .batch(batch_size)
    .prefetch(AUTOTUNE)
)
```

Figure 7: Creating train_dataset and test_dataset.

c. Simple CNN model

To create a simple CNN model, I followed a standard pattern of CNN architecture that the height, width will reduce overtime, while the number of channels will increase. Moreover, I stack 2 Conv2D layers in each stage to help the model to learn more features. At first, I decided to use 5 stages but because the training set and test set are not large, but it might cause overfitting without increasing the accuracy much, so I chose to use 3 stages. Because it is just a simple model, so I just used Sequential API to build this model.

```
# Create a simple CNN model
model = models.Sequential( # Using Sequential API for simplicity
    [
        data_augmentation, # Add data augmentation layer
        Conv2D(32, (3, 3), padding='same', activation='relu'), # padding same to keep the height and width the same
        BatchNormalization(),
        Conv2D(32, (3, 3), padding='same', activation='relu'), # padding same to keep the height and width the same
        BatchNormalization(),
        MaxPooling2D((2, 2)), # Reduce the height and width by half

        Conv2D(64, (3, 3), padding='same', activation='relu'), # (16, 16, 64)
        BatchNormalization(),
        Conv2D(64, (3, 3), padding='same', activation='relu'), # (16, 16, 64)
        BatchNormalization(),
        MaxPooling2D((2, 2)), # (8, 8, 64)

        Conv2D(128, (3, 3), padding='same', activation='relu'), # (8, 8, 128)
        BatchNormalization(),
        Conv2D(128, (3, 3), padding='same', activation='relu'),  # (8, 8, 128)
        BatchNormalization(),
        MaxPooling2D((2, 2)), # (4, 4, 128)
        Dropout(0.3), # Dropout layer for regularizations

        # Conv2D(256, (3, 3), padding='same', activation='relu'), # (4, 4, 256)
        # BatchNormalization(),
        # Conv2D(256, (3, 3), padding='same', activation='relu'), # (4, 4, 256)
        # BatchNormalization(),
        # MaxPooling2D((2, 2)), # (2, 2, 256)
        # Dropout(0.3),

        # Conv2D(512, (3, 3), padding='same', activation='relu'),
        # BatchNormalization(),
        # Conv2D(512, (3, 3), padding='same', activation='relu'),
        # BatchNormalization(),
        # Dropout(0.3)
    ]
)
```

Figure 8: Simple CNN architecture without classification layer

```
# Add Dense layers after flatten, with the final dense layers to give 10 outputs (10 classes)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10))
```

Figure 9: Adding classification layers on top

```
Total params: 847,530 (3.23 MB)

Trainable params: 846,634 (3.23 MB)

Non-trainable params: 896 (3.50 KB)
```

Figure 10: Summary of the model parameters

Eventhough the model is quite small, it achieved a quite acceptable accuracy when I trained it with *EarlyStopping*, *ReduceLROnPlateau* and *Adam* optimizers (~88% validation accuracy).

```
# Early stopping to optimize training time and reduce overfitting
early_stop = EarlyStopping(
    monitor="val_loss",    # Stop if validation loss stops improving
    patience=15,           # Allow some patience
    restore_best_weights=True,  # Roll back to the best model
    verbose=1
)
```

Using ReduceLROnPlateau for learning rate decay, this will reduce the learning rate when the validation loss shows no improvements after a specific number of epochs.

```
# Reduce LR on Plateau
reduce_lr = ReduceLROnPlateau(
    monitor="val_loss",    # Watch validation loss
    factor=0.5,            # Reduce LR by half
    patience=5,            # Wait 5 epochs before reducing
    min_lr=1e-6,           # Don't let LR go below this
    verbose=1
)
```

Figure 11: EarlyStopping and ReduceLROnPlateau

```
# Optimizer Adam
opt = tf.keras.optimizers.Adam(learning_rate=0.001)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

model.compile(optimizer=opt, loss=loss, metrics=['accuracy'])

history = model.fit(
    train_dataset,
    epochs=200,
    validation_data=test_dataset,
    callbacks=[early_stop, reduce_lr], # Includ early_stop and reduce_lr
    verbose=1
)
```

Figure 12: Compile and train the model

```
Epoch 1/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 15s 16ms/step - accuracy: 0.4105 - loss: 1.6939 - val_accuracy: 0.3207 - val_loss: 2.1874 - learning_rate: 0.0010
Epoch 2/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.6146 - loss: 1.0786 - val_accuracy: 0.6870 - val_loss: 0.9120 - learning_rate: 0.0010
Epoch 3/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.6895 - loss: 0.8813 - val_accuracy: 0.6830 - val_loss: 0.9754 - learning_rate: 0.0010
Epoch 4/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.7301 - loss: 0.7632 - val_accuracy: 0.7369 - val_loss: 0.7775 - learning_rate: 0.0010
Epoch 5/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.7621 - loss: 0.6780 - val_accuracy: 0.7175 - val_loss: 0.8523 - learning_rate: 0.0010
Epoch 6/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.7830 - loss: 0.6249 - val_accuracy: 0.7467 - val_loss: 0.7401 - learning_rate: 0.0010
Epoch 7/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.7972 - loss: 0.5840 - val_accuracy: 0.7828 - val_loss: 0.6390 - learning_rate: 0.0010
Epoch 8/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.8115 - loss: 0.5489 - val_accuracy: 0.7828 - val_loss: 0.6517 - learning_rate: 0.0010
Epoch 9/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.8234 - loss: 0.5080 - val_accuracy: 0.7934 - val_loss: 0.6264 - learning_rate: 0.0010
Epoch 10/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.8321 - loss: 0.4839 - val_accuracy: 0.7884 - val_loss: 0.6483 - learning_rate: 0.0010
Epoch 11/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.8406 - loss: 0.4603 - val_accuracy: 0.8170 - val_loss: 0.5535 - learning_rate: 0.0010
Epoch 12/200
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.8515 - loss: 0.4348 - val_accuracy: 0.8254 - val_loss: 0.5261 - learning_rate: 0.0010
Epoch 13/200
...
Epoch 50: ReduceLROnPlateau reducing learning rate to 1.5625000742147677e-05.
391/391 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.9628 - loss: 0.1031 - val_accuracy: 0.8793 - val_loss: 0.4646 - learning_rate: 3.1250e-05
Epoch 50: early stopping
Restoring model weights from the end of the best epoch: 35.
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Figure 13: Simple model accuracy

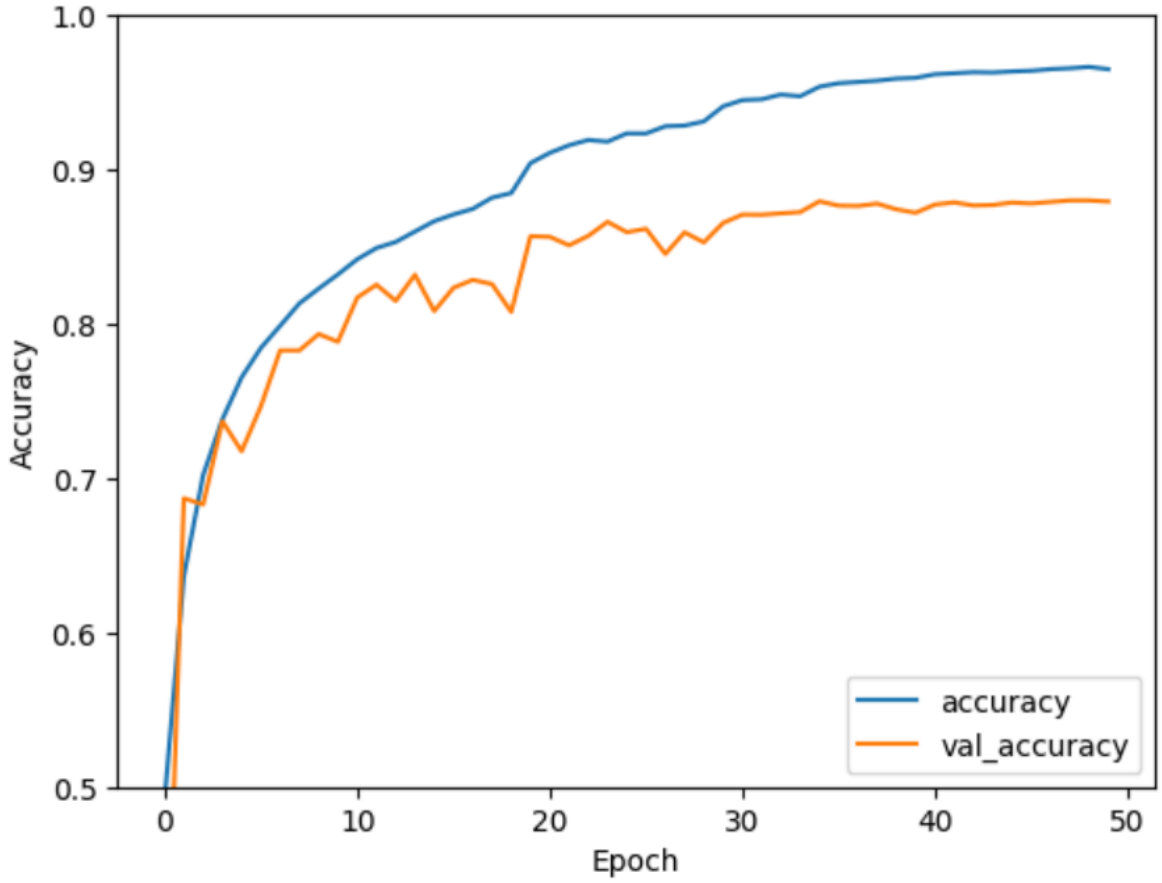

Figure 14: Learning curve of the simple model

d. ResNet

The second section of this lab will focus on buiding a *Residual Network* using TensorFlow to improve the accuracy on the CIFAR-10 dataset.

The general architecture of a `Residual Network` contains four main components:

- Identity block - keeping the height and width of the input to be the same, while increasing the number of channels. This block is used for the model to capture the features of the data.
- Convolutional block - changes the height and width of the feature maps (usually reduces by stride > 1) and increases the number of channels. This block is used to downsample the input, making the feature maps richer.
- Pooling layer - reduces the spatial dimensions (height and width) while keeping the depth (channels).
- Dense layer - maps extracted features into final outputs.

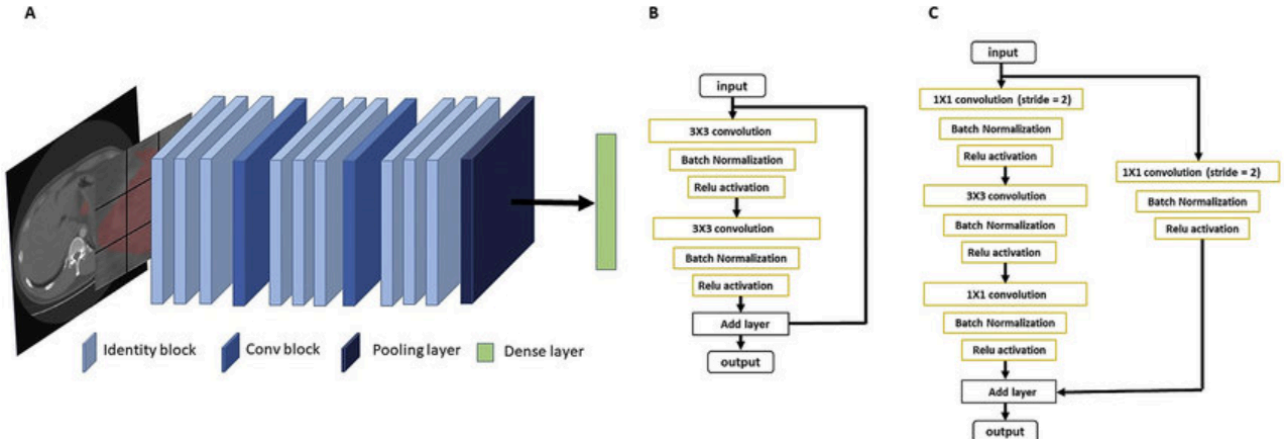

Figure 15: General architecture of Residual Network

Residual Network handles multiple inputs because it adds the shortcut at the end, so I will use Functional API to build a it instead of Sequential API. Functional API allows me to deeply control the inputs of the layers.

8. Identity Block

```python
def identity_block(X, f, filters, initializer=he_normal):
    """
    Implementation of identity block

    Arguments
    X --- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
    f --- integer, specifying the shape of the middle CONV's window for the main path
    filters --- python list of integers, defining the number of filters in the
        CONV layers of the main path
    initializer --- to set yp the initial weights of a layer. Equals to random
        uniform initializer

    Returns:
    X --- output of the identity block, tensor of shape (m, n_H, n_W, n_C)
    """

    # Get the filters
    F1, F2 = filters

    # Save the input value. For residual
    X_shortcut = X

    # First component of main path
    X = Conv2D(filters=F1, kernel_size = 1, strides=(1,1), padding='valid', kernel_initializer=initializer(seed=42),
               kernel_regularizer=tf.keras.regularizers.l2(5e-4))(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)

    # Second component of main path
    X = Conv2D(filters=F2, kernel_size=f, strides=(1,1), padding='same',kernel_initializer=initializer(seed=42),
               kernel_regularizer=tf.keras.regularizers.l2(5e-4))(X)
    X = BatchNormalization(axis=3)(X)

    # Final step: add shortcut to main path
    X = Add()([X_shortcut, X])
    X = Activation('relu')(X)

    return X
```

Figure 16: Building the Identity block

9. Convolutional Block

```python
def convolutional_block(X, f, filters, s=2, initializer=he_normal):
    """
    Convolutional block with stride s
    """
    F1, F2 = filters
    X_shortcut = X

    # Main path
    X = Conv2D(F1, (1, 1), strides=(s, s), padding='valid',
               kernel_initializer=initializer(seed=42),
               kernel_regularizer=tf.keras.regularizers.l2(5e-4))(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)

    # Main Path
    X = Conv2D(F2, (f, f), strides=(1, 1), padding='same',
               kernel_initializer=initializer(seed=42),
               kernel_regularizer=tf.keras.regularizers.l2(5e-4))(X)
    X = BatchNormalization(axis=3)(X)

    # Shortcut path
    X_shortcut = Conv2D(F2, (1, 1), strides=(s, s), padding='valid',
                        kernel_initializer=initializer(seed=42),
                        kernel_regularizer=tf.keras.regularizers.l2(5e-4))(X_shortcut)
    X_shortcut = BatchNormalization(axis=3)(X_shortcut)

    # Add
    X = Add()([X, X_shortcut])
    X = Activation('relu')(X)

    return X
```

Figure 17: Building the convolutional block

Then, I created a ResNet function, which combines the two identity and convolutional blocks with MaxPooling and Dense layers.

```python
# Create a ResNet model, which takes the input shaped (32, 32, 3) and ouput 10 class scores
def ResNet(input_shape=(32, 32, 3), classes=10):

    X_input = Input(input_shape) # Define the input shape

    X = data_augmentation(X_input) # Perform data Augmentation

    # Stage 1 - First Conv2D to capture some input features
    X = Conv2D(64, (3, 3), strides=(1, 1),
               padding='same', kernel_initializer=he_normal(seed=42),
               kernel_regularizer=tf.keras.regularizers.l2(5e-4))(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)


    # Stage 2 - Stack 1 convolutional block with 2 identity blocks
    X = convolutional_block(X, f=3, filters=[64, 64], s=1)
    X = identity_block(X, 3, [64, 64])
    X = identity_block(X, 3, [64, 64])

    # Stage 3 - Stack 1 convolutional block with 2 identity blocks
    X = convolutional_block(X, f=3, filters=[128, 128], s=2)
    X = identity_block(X, 3, [128, 128])
    X = identity_block(X, 3, [128, 128])

    # Stage 4 - Stack 1 convolutional block with 2 identity blocks
    X = convolutional_block(X, f=3, filters=[256, 256], s=2)
    X = identity_block(X, 3, [256, 256])
    X = identity_block(X, 3, [256, 256])

    # Stage 5 - Stack 1 convolutional block with 2 identity blocks
    X = convolutional_block(X, f=3, filters=[512, 512], s=2)
    X = identity_block(X, 3, [512, 512])
    X = identity_block(X, 3, [512, 512])

    # Global Average Pooling
    X = GlobalAveragePooling2D()(X)

    # Add Dropout for regularizations
    X = Dropout(0.3)(X)

    # Output
    X = Dense(classes, activation='softmax',
              kernel_initializer=he_normal(seed=42), kernel_regularizer=tf.keras.regularizers.l2(5e-4))(X)

    model = Model(inputs=X_input, outputs=X)
    return model
```

Figure 18: Building a ResNet function

- After that, I built two ResNet model, the first one using CosineDecayRestarts, which is a learning rate decay with restart schedule. The learning rate will reduce to the predefined minimum number before increasing again. This might help the model to escape from the Plateau or local minima (Figure 19).

```python
CDR_model = ResNet(input_shape=(32, 32, 3), classes=10)

# Cosine decay restarts learning rate schedule
lr_schedule = tf.keras.optimizers.schedules.CosineDecayRestarts(
    initial_learning_rate=0.1,
    first_decay_steps=200,
    t_mul=2.0,
    m_mul=0.8,
    alpha=1e-5
)

# Check for the summary of the model
CDR_model.summary()

# Optimizer with the learning rate schedule
opt = tf.keras.optimizers.SGD(learning_rate=lr_schedule, momentum=0.9, nesterov=True)

# Loss function
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# Compile the model
CDR_model.compile(optimizer=opt, loss=loss, metrics=['accuracy'])

# Create checkpoint
checkpoint = ModelCheckpoint(
    "best_resnet_CDR.h5",
    monitor="val_accuracy",
    save_best_only=True,
    mode="max",
    verbose=1
)

# === Training ===
CDR_history = CDR_model.fit(
    train_dataset,
    epochs=200,
    validation_data=test_dataset,
    callbacks=[early_stop, checkpoint],
    verbose=1
)
```

Figure 19: First ResNet model with CosineDecayRestarts (CDR_model).

```
Total params: 10,489,674 (40.01 MB)

Trainable params: 10,476,106 (39.96 MB)

Non-trainable params: 13,568 (53.00 KB)
```

Figure 20: Number of parameters for CDR_model.

Unfortunately, this model just achieved the accuracy quite as the same as the simple model, but it took more time and the accuracy fluctuated a lot.

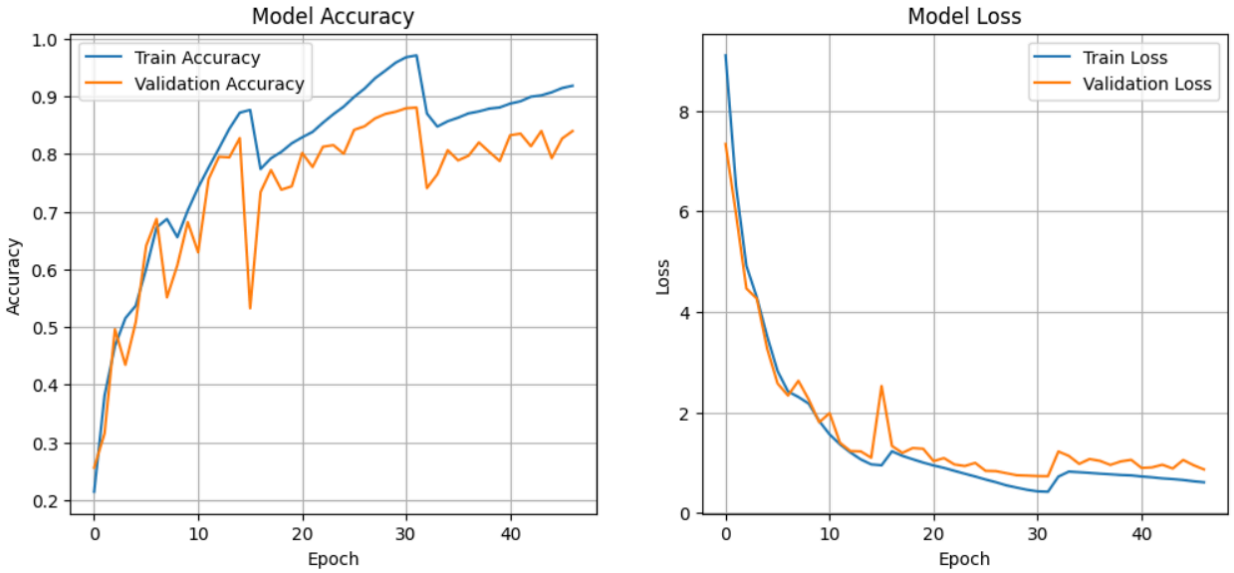

Figure 21: Learning curves for CDR_model

- The second one using ReduceLROnPlateau, which is a learning rate decay. The learning rate will reduce after a specific number of epochs when no improvements are observed in the validation accuracy (Figure 22).

```python
RLOP_model = ResNet(input_shape=(32, 32, 3), classes=10)

# Check for the summary of the model
RLOP_model.summary()

# Optimizer SGD
opt = SGD(learning_rate=0.1, momentum=0.9, nesterov=True)

# Loss function
loss = SparseCategoricalCrossentropy(from_logits=False)

# Compile model
RLOP_model.compile(optimizer=opt, loss=loss, metrics=['accuracy'])

# Creat check point
checkpoint = ModelCheckpoint(
    "best_resnet_RLOP.h5",
    monitor="val_accuracy",
    save_best_only=True,
    mode="max",
    verbose=1
)

# === Training ===
RLOP_history = RLOP_model.fit(
    train_dataset,
    epochs=200,
    validation_data=test_dataset,
    callbacks=[early_stop, checkpoint, reduce_lr],
    verbose=1
)
```

Figure 22: First ResNet model with ReduceLROnPlateau (RLOP_model)

This model achieved a much higher accuracy compared to the previous two models, at approximately 91.5 -> 93% validation accuracy. I have run this model previously on another notebook and the accuracy ~ 92.5% (Figure 24) but for this notebook the accuracy is just 91.5% (Figure 23).

One problem I encountered was that the model achieved the highest accuracy at around 100 epochs and eventhough I set EarlyStopping but it continued to run for a huge number of epochs without stopping.

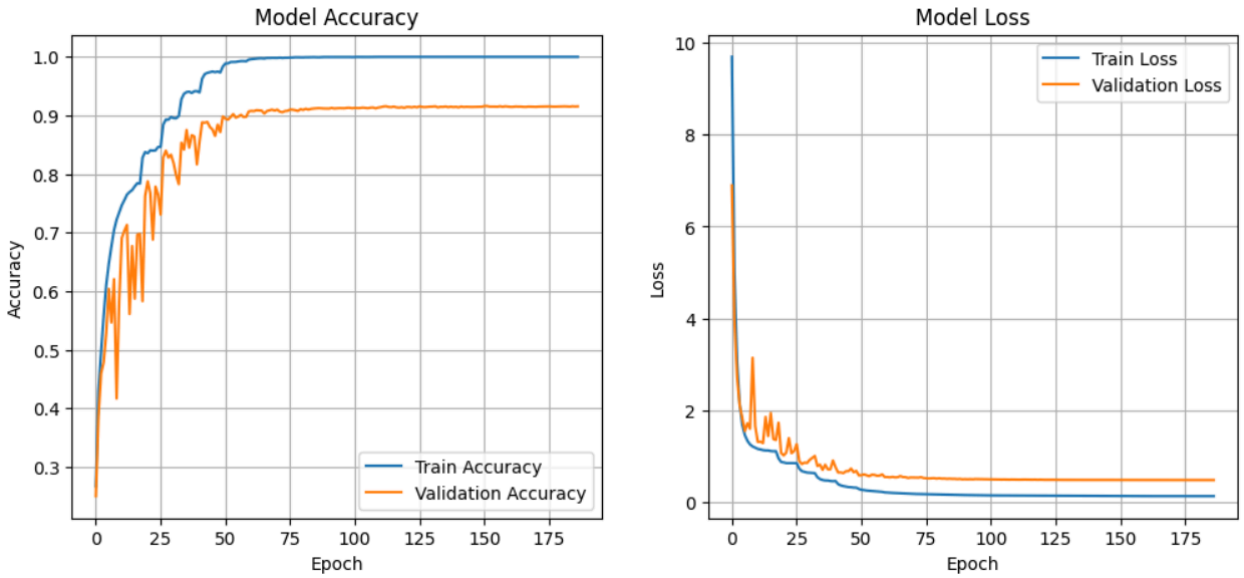

Figure 23: Model achieved 91.5% accuracy

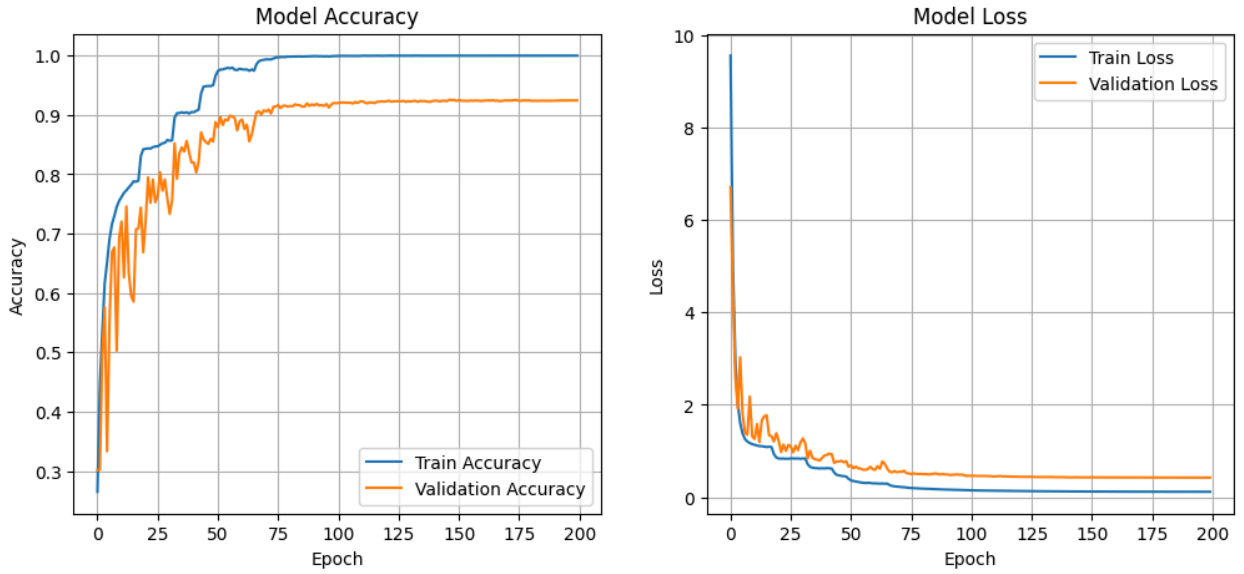

Figure 24: Model achieved 92.5% accuracy