Fine Tune Image Classification using CNN

In this project, we will revisit the `CIFAR-10` classification project and apply some of the improvement techniques to increase the accuracy from ~66% to ~85%.

We will accomplish the project by following these steps:

1. Import the dependencies.
2. Get the data ready for training:
   - Download the data from the `Keras` library.
   - Split it into train, validate, and test datasets.
   - Normalize the data.
   - One-hot encode the labels.
   - Data augmentation
3. Build the model architecture. In addition to simple convolutional and pooling layers, we add the following layers to our architecture:
   - Deeper neural network to increase learning capacity
   - Dropout layers
   - L2 regularization to our convolutional layers
   - Batch normalization layers
4. Train the model.
5. Plot the learning curve.
6. Evaluate the model.

Remark: This is assumed that you can run the entire program in one shot. If not, you will need to save and reload the model or weights to continue training your data and keep the log if you want to plot the learning curve.

Step 1: Import Dependencies

Here's the `Keras` code to import the needed dependencies:

```
#import dependencies
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Dense, Activation, Flatten,
Dropout, BatchNormalization
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.datasets import cifar10
```

```
from tensorflow.keras import regularizers, optimizers
import numpy as np
from matplotlib import pyplot
```

Step 2: Get the Data Ready for Training

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

1. Normalize the data

Normalizing the pixel values of our images is done by subtracting the mean from each pixel and then dividing the result by the standard deviation:

```
# Normalize the data to speed up training
mean = np.mean(x_train)
std = np.std(x_train)
x_train = (x_train-mean)/(std+1e-7)
x_test = (x_test-mean)/(std+1e-7)
```

2. One-hot encode the labels

```
num_classes = 10
y_train = to_categorical(y_train,num_classes)
y_test = to_categorical(y_test,num_classes)
```

3. Data augmentation

For augmentation techniques, we will arbitrarily go with the following transformations: rotation, width and height shift, and horizontal flip.

```
# data augmentation
datagen = ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    vertical_flip=False
    )

# compute the data augmentation on the training set
datagen.fit(x_train)
```

Remark: When you are working on problems, view the images that the network missed or provided poor detections for and try to understand why it is not performing well on them. Then create your hypothesis and experiment with it. For example, if the missed images were of shapes that are rotated, you might want to try the rotation augmentation. You would apply that, experiment, evaluate, and repeat. You will come to your decisions purely from analyzing your data and understanding the network performance.

Step 3: Build the Model Architecture

In this project, we will build a deeper network for increased learning capacity (6 CONV + 1 FC). The network has the following configuration:

- Instead of adding a pooling layer after each convolutional layer, we will add one after every two convolutional layers. This idea was inspired by VGGNet, a popular neural network architecture developed by the Visual Geometry Group (University of Oxford). VGGNet will be explained later.
- Inspired by VGGNet, we will set the `kernel_size` of our convolutional layers to 3×3 and the `pool_size` of the pooling layer to 2×2.
- We will add dropout layers every other convolutional layer, with (p) ranges from 0.2 and 0.4.
- A batch normalization layer will be added after each convolutional layer to normalize the input for the following layer.
- In `Keras`, L2 regularization is added to the convolutional layer code.

```
# build the model

base_hidden_units = 32

# l2 regularization hyperparameter
weight_decay = 1e-4

model = Sequential()

# CONV1
model.add(Conv2D(base_hidden_units, (3,3), padding='same',
kernel_regularizer = regularizers.l2(weight_decay), input_shape =
x_train.shape[1:]))
model.add(Activation('relu'))
model.add(BatchNormalization())

# CONV2
model.add(Conv2D(base_hidden_units, (3,3), padding='same',
kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())
```

```
# POOL + Dropout
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.2))

# CONV3
model.add(Conv2D(2*base_hidden_units, (3,3), padding='same',
kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# CONV4
model.add(Conv2D(2*base_hidden_units, (3,3), padding='same',
kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# POOL + Dropout
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.3))

# CONV5
model.add(Conv2D(4*base_hidden_units, (3,3), padding='same',
kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# CONV6
model.add(Conv2D(4*base_hidden_units, (3,3), padding='same',
kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# POOL + Dropout
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.4))

# FC7
model.add(Flatten())
model.add(Dense(num_classes, activation='softmax'))
```

Step 4: Train the Model

Before we jump into the training code, let's discuss the strategy behind some of the hyperparameter settings:

- `batch_size` — This is the mini-batch hyperparameter that we covered. The higher the `batch_size`, the faster your algorithm learns. You can start with a mini-batch of 64 and double this value to speed up training.
- `epochs`—I started with 50 training iterations and found that the network was still improving. So I kept adding more epochs and observing the training results. In this project, I was able to achieve

87% accuracy of validation after 125 epochs. As you will see soon, there is still room for improvement if you let it train longer.

- Optimizer—I used the `Adam` optimizer.

Remark: you will need to use GPU for training.

```
# training
batch_size = 128
epochs = 125

from tensorflow.keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath='model.125epochs.hdf5',
               verbose=1, save_best_only=True)
optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001,decay=1e-6)
model.compile(loss='categorical_crossentropy', optimizer=optimizer,
              metrics=['accuracy'])
history = model.fit(datagen.flow(x_train, y_train, batch_size =
          batch_size), callbacks=[checkpointer],
          steps_per_epoch=x_train.shape[0] // batch_size,
          epochs = epochs,verbose = 2,
          validation_data=(x_valid,y_valid))
```

When you run this code, you will see the verbose output of the network training for each epoch. Keep your eyes on the `loss` and `val_loss` values to analyze the network and diagnose bottlenecks. Figure below shows the verbose output of epochs 115 to 117. The `val_loss` does not improve after epoch 115.

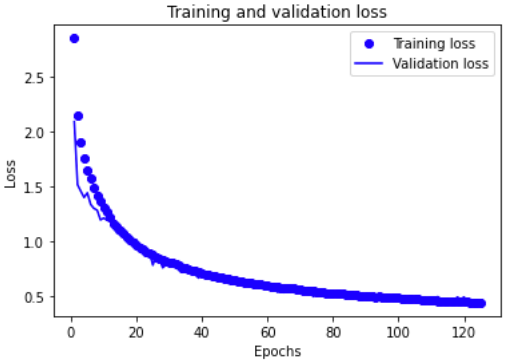
```
Epoch 115/125

Epoch 00115: val_loss improved from 0.45393 to 0.44453, saving model to
model.125epochs.hdf5
351/351 - 24s - loss: 0.4558 - accuracy: 0.8537 - val_loss: 0.4445 -
val_accuracy: 0.8674 - 24s/epoch - 69ms/step
Epoch 116/125

Epoch 00116: val_loss did not improve from 0.44453
351/351 - 24s - loss: 0.4527 - accuracy: 0.8534 - val_loss: 0.4668 -
val_accuracy: 0.8570 - 24s/epoch - 68ms/step
Epoch 117/125

Epoch 00117: val_loss did not improve from 0.44453
351/351 - 24s - loss: 0.4540 - accuracy: 0.8549 - val_loss: 0.4653 -
val_accuracy: 0.8596 - 24s/epoch - 68ms/step
```

Step 5: Plot Learning Curve

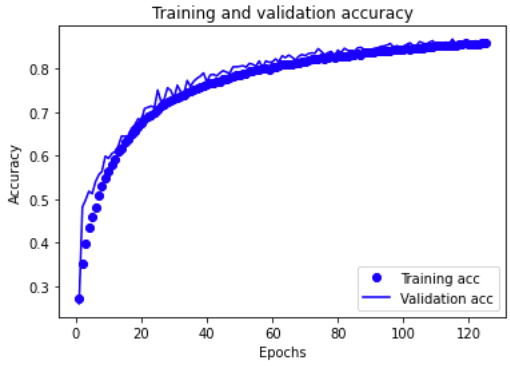
You should always plot the learning curves to analyze the training performance and diagnose

overfitting and underfitting.

```
import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```




```
plt.clf()   # clear figure
acc_values = history.history['accuracy']
val_acc_values = history.history['val_accuracy']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



Fun Time: Judging from these figures, do you further performance improvement can be achieved if we increase more epochs? (1) Yes (2) No.

Step 6: Final Verdict: Evaluate Your Model Based on New Data

```
# evaluating the model
scores = model.evaluate(x_test, y_test, batch_size=128, verbose=1)
print('\nTest result: %.3f loss: %.3f' % (scores[1]*100,scores[0]))
```

And we reach the accuracy of 84.85%.

You can download the above code `CNN_Cifar10_FineTune.ipynb` from the course website.

Further Improvement

Accuracy of 85% is not bad, but you can still improve further. Here are some ideas you can experiment with:

- More training epochs—Notice that the network was improving until epoch 115. You can increase the number of epochs to 200 or 300 and let the network train longer.
- Deeper network—Try adding more layers to increase the model complexity, which increases the learning capacity.
- Lower learning rate—Decrease the learning rate (you should train longer if you do so).
- Different CNN architecture—Try something like Inception or ResNet (explained later).
- Transfer learning—In follow-up lecture, we will explore the technique of using a pre-trained network on your dataset to get higher results with almost no cost of training time.