Chapter 14: Programming Exercise CIFAR10 GoogLeNet

Exercise: Classification of CIFAR10 with GoogLeNet

In this exercise we are going to classify images of airplanes, automobiles, birds, cats, deers, dogs, frogs, horses, ships and trucks using GoogLeNet.

> Note: we will be working with a large convolutional network. Make sure to have a GPU enabled or training will be painfully slow. If you are using Google Colab, make sure to select a GPU runtime.

Step 1: Load the Data

- We start by reading the dataset we are going to use. This is a built-in dataset in Keras. You get this code in the cells below.

```
from keras.datasets import cifar10
(X_train_full, y_train_full), (X_test, y_test) = cifar10.load_data()
```

- How many examples do the training set and test set contain?
- We will use the last 10000 examples from the training set as our validation set. The other examples will be used as training set. Write code to create the validation and training data.
- The original images are 32 by 32 pixels. We will resize them to 128 by 128 pixels prior to feeding them to the network.

```
HEIGHT = 128
WIDTH = 128
```

Step 2: Data exploration

> Note: this this is the same as in the previous exercise.

- Give the number of samples for each category
- Show the first 25 examples in a 5 by 5 grid, each time showing the class (label) of each example along with the image.

```
class_names = (['airplane', 'automobile', 'bird', 'cat', 'deer',
                'dog', 'frog', 'horse', 'ship', 'truck'])
```

Step 3: Build a Model

- Implement GoogLeNet using the subclassing API to implement the Inception Module. The actual network itself is a (large) sequential stack.
- The GoogLeNet Architecture

> Note: the image is taken from the book, but has been altered at one point to fix a mistake in the original image. The first inception

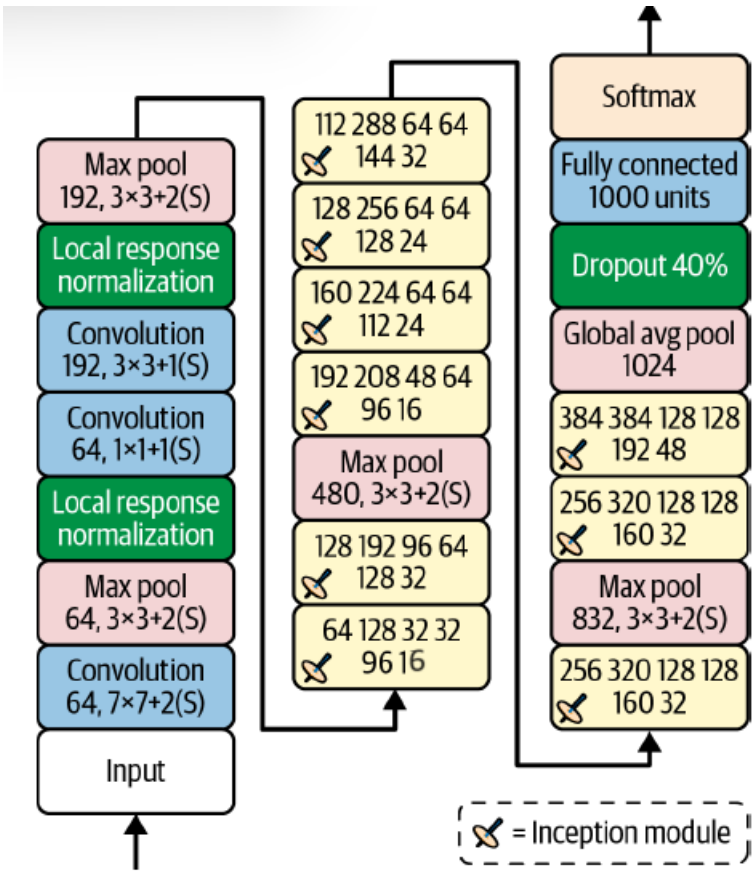

Figure 1: GoogLeNet Architecture

module doesn't have a layer with 12 filters (as in the book) but instead should use 16 filters. This is how it is documented in the original paper Going Deeper with Convolutions.

- GoogLeNet uses something called the Inception Module
  - The notation `3 × 3 + 1(S)` means that this convolutional layer uses a 3×3 kernel, stride 1, and "same" padding.
  - All convolutional layers use the ReLU activation function.
  - Every single layer uses a stride of 1 and "same" padding (even the max pooling layer).
  - The concatenation layer can be implemented using Keras's `Concatenate` layer, using the default `axis=-1`.

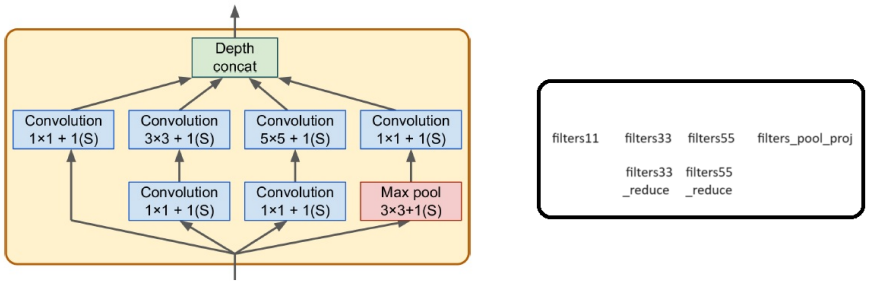

Figure 2: inception module

- Use the subclassing API to implement a Keras `Layer` that performs the operations described in the Inception Module
  - An example of the use of the subclassing API can be found in the book on page 336 (and on page 515).

```
DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, strides=1,
                        padding="same", kernel_initializer="he_normal", use_bias=False)

class ResidualUnit(tf.keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = tf.keras.activations.get(activation)
        self.main_layers = [
            DefaultConv2D(filters, strides=strides),
            tf.keras.layers.BatchNormalization(),
            self.activation,
            DefaultConv2D(filters),
            tf.keras.layers.BatchNormalization()
        ]
        self.skip_layers = []
        if strides > 1:
```

```
            self.skip_layers = [
                DefaultConv2D(filters, kernel_size=1, strides=strides),
                tf.keras.layers.BatchNormalization()
            ]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)
```

- We want to write similar code to the `ResidualUnit` to implement the Inception Module
- We are going to add a boolean `use_batch_norm` to be able to easily train and test the entire model with or without using BatchNormalization at the end of the Inception Module.
- Use the following code to start from:

```
import functools

DefaultConv = functools.partial(
  tf.keras.layers.Conv2D, kernel_size=(1, 1), strides=(1, 1),
                          padding="same", activation="relu")

class InceptionModule(tf.keras.layers.Layer):
    def __init__(self, filters11, filters33_reduce, filters33,
                       filters55_reduce, filters55, filters_pool_proj,
                       use_batch_norm=True, **kwargs):
        super().__init__(**kwargs)
        self.conv11 = DefaultConv(filters=filters11)

        # YOUR CODE HERE

    def call(self, inputs):
        path1 = self.conv11(inputs)

        # YOUR CODE HERE
```

- Complete the `__init__` method:
  - Create the Convolutional Layer at the left. Use `DefaultConv2D` and `filters11`. (already present in the previous example code)
  - Create the Convolutional Layer at the left middle, at the bottom. Use `DefaultConv2D` and `filters33_reduce`.
  - Create the Convolutional Layer at the left middle, at the top. Use

`DefaultConv2D` and `filters33`.
  - Create the Convolutional Layer at the right middle, at the bottom. Use `DefaultConv2D` and `filters55_reduce`.
  - Create the Convolutional Layer at the right middle, at the top. Use `DefaultConv2D` and `filters55`.
  - Create the MaxPooling Layer at the right.
  - Create the Convolutional Layer at the right. Use `DefaultConv2D` and `filters_pool_proj`.

  - if `use_batch_norm` is `True`: Create a BatchNormalization Layer
- Complete the `__call__` method:
  - Use of the Convolutional Layer at the left (already present in the previous example code)
  - Use of the Convolutional Layer at the left middle, at the bottom
  - Use of the Convolutional Layer at the left middle, at the top
  - Use of the Convolutional Layer at the right middle, at the bottom
  - Use of the Convolutional Layer at the right middle, at the top
  - Use of the MaxPooling Layer at the right
  - Use of the Convolutional Layer at the right
  - Concatenate the outputs
  - if `use_batch_norm` is `True`: Use a BatchNormalization Layer
- Create an instance of the InceptionModule (e.g. the first Inception Module in GoogLeNet).
  - What is the shape?
  - How many weights (parameters) does it have? Does this match your expectations?
- Implement the entire model.
  - Use the following code to start from

  Note: we are not going to bother putting in the Local Response Normalization. We will use BatchNormalization instead.

```
DefaultMaxPool = functools.partial(
    tf.keras.layers.MaxPool2D,
    pool_size=(3,3), strides=(2,2), padding='same'
)
def get_googlenet_model(input_shape, num_classes, use_batch_norm=True, **kwargs):

  model = tf.keras.Sequential(**kwargs)

  # YOUR CODE HERE
```

- Complete the code above. Use the image of the GoogLeNet architecture
  - add the first Convolutional Layer
  - if `use_batch_norm`: add a BatchNormalization Layer
  - add the MaxPooling Layer
  - add the second and third Convolutional Layer

  - if `use_batch_norm`: add a BatchNormalization Layer
  - add the MaxPooling Layer
  - add the 2 Inception Modules
  - add the MaxPooling Layer
  - add the 5 Inception Modules
  - add the MaxPooling Layer
  - add the 2 Inception Modules
  - add the GlobalAveragePooling Layer
  - add the Dropout Layer
  - add the output Layer

  Note: if possible try to avoid copy-pasting code for the Inception Modules. Try to use a loop instead.

- Before we can finally build the model, we are going to add a preprocessing layer, to resize the images to `WIDTH = HEIGHT = 128` and to rescale the pixel values between -1 and 1. Use the following Python code. Make sure you understand the Python code

- Before we can finally build the model, we are going to add a preprocessing layer, to resize the images to WIDTH = HEIGHT = 128 and to rescale the pixel values between -1 and 1. Use the following Python code. Make sure you understand the Python code

```
preprocess_layer = tf.keras.Sequential([
    tf.keras.layers.Resizing(HEIGHT, WIDTH, input_shape=(32,32,3)),
    tf.keras.layers.Rescaling(scale=1/127.5, offset=-1)
])

tf.keras.backend.clear_session()
model = get_googlenet_model(input_shape=(128,128,3), num_classes=10,
                            use_batch_norm=False, name="GoogLeNet")
model.summary()
full_model = tf.keras.Sequential([preprocess_layer, model])
```

  Note: why do we pass the name "GoogLeNet" to the function `get_googlenet_model`? What happens if you don't do this?

Step 4: Compile the Model

Next, compile the model.

- Use `adam` as the optimizer with learning_rate = 0.001.
- Specify the correct loss for this classification problem.
- Track the `accuracy` metric.

Step 5: Train the Model

- Train the model.
  - Train it for a large number epochs.

  - Use early stopping to prevent (too much) overfitting.
  - Use a batch size of 128.
  - Be sure to use the validation data.

Step 6: Evaluate the Model

- Use the `evaluate` method to evaluate the model on the test set.
  - What is the performance of the model on the test set?

Step 7: Improve the Model

- Introduce BatchNormalization. How are you going to do that in a very simple way?
- Compile, train and evaluate the model again.
- We want to add Data Augmentation to get better results
  - Create a `tf.keras.Sequential` object to add `RandomFlip`, `RandomRotation` and `RandomContrast`. You can use code from the book for this.
  - What are you going to do to add the data augmentation before the preprocessing?
- Compile, train and evaluate the model again.
- Reduce the learning rate by 1 / 10 once the model has stopped training. Compile and train the model further. Do **not** recreate the model, you want to continue training from where you left off.
- Reduce the learning rate again by 1 / 10. Compile and train the model. Do not recreate the model. This probably doesn't yield any further improvement.
- Use Monte Carlo Dropout (see book page 397)
  - Don't use the entire test set in one time. Loop through the test set in batches of 500 examples
  - Restrict to 20 samples for speed

  Note: if you call the model on the entire test set you will most probably get an out of memory error (at least on Google Colab).

With all the techniques mentioned above, you should be able to get a test accuracy around 90% or so.