Deep Learning

Practical 4

1. From last Practicals

In the last practical, we transformed a Dense classic network into a variadic convolutional sequentialy separated residual encoder/decoder network.

The objective of this practical is to create custom layers in order to clean the code and have a taste of the power of composition of layers provided by Keras.

Let's proceed step by step.

Provide a working `.py` file for each part.

2. Primer on Custom Layers

2.1. Introduction

In TensorFlow's Keras API, creating custom layers allows for specialized and intricate behaviors not available in standard layers. This part will guide you through the steps of crafting a custom layer by extending the tf.keras.layers.Layer class.

2.2. Initialization (`__init__` method):

- Purpose: Set up any initial layer-specific parameters or sublayers.
- Implementation: Override the `__init__` method to initialize your layer. This method is called when you create an instance of your custom layer.

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Initialize layer parameters
```

2.3. Building the Layer (`build` method):

- Purpose: Create the weights of the layer, usually depending on the input shape.
- Implementation: Override the `build` method. It is invoked automatically to create the layer's weights.

```
    def build(self, input_shape):
        # Create layer weights
```

2.4. Layer Computation (`call` method):

- Purpose: Define the computation performed by the layer.
- Implementation: Override the `call` method, which is where the layer's forward computation logic resides.

```
    def call(self, inputs):
        # Layer computation logic
        return output
```

2.5. Configuration for Serialization (`get_config` method):

- Purpose: Enable the layer to be serializable. This is crucial for saving and loading models that contain this custom layer.
- Implementation: Override the `get_config` method to return a dictionary containing the layer configuration and all compile-time informations.

```
    def get_config(self):
        config = super().get_config()
        # Add layer-specific configuration
        config.update({
            ...
        })
        return config
```

2.6. Example of a Custom Layer

Here is an example illustrating how these methods come together:

```
class MyCustomLayer(tf.keras.layers.Layer):
    def __init__(self, custom_parameter=42, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.custom_parameter = custom_parameter

    def build(self, input_shape):
        self.kernel = self.add_weight("kernel", shape=inputs_shape)

    def call(self, inputs):
        # Define the forward computation using inputs and self.kernel
        return inputs + self.kernel

    def get_config(self):
        config = super().get_config()
        config.update({
            # must have the same name as the self. method
            "custom_parameter": self.custom_parameter
        })
        return config
```

In this example, the MyCustomLayer class defines a custom layer with an additional parameter custom_parameter, a weight kernel, and the necessary computation in the call method. The get_config method

includes the custom parameter in the layer's configuration, ensuring that the layer's settings can be saved and loaded correctly.

3. A Sequentialy Separated Convolutional Custom Layer

The first layer to get the custom treatment is the big `for loop` block that is repeated multiple times in the code:

```
for _ in range(3):
    hidden_layer = tf.keras.layers.Conv2D(
        filters=compression_value, kernel_size=(3, 1), padding="same"
    )(hidden_layer)
    hidden_layer = tf.keras.layers.Conv2D(
        filters=compression_value, kernel_size=(1, 3), padding="same"
    )(hidden_layer)
    hidden_layer = tf.keras.layers.Activation("relu")(hidden_layer)
    hidden_layer = tf.keras.layers.Dropout(0.2)(hidden_layer)
```

We will start with a fresh custom layer base:

```
class SequentialySeparatedConv2D(tf.keras.layers.Layer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def call(self, inputs):
        return inputs

    def get_config(self):
        config = super().get_config()
        config.update({})
        return config

    def build(self, input_shape):
        super().build(input_shape)
```

3.1. First Step, Compile-Time Data

The first question to ask is: what are the needed variables that are only known at compile time ?

- In our example, we will need the `range` n, computed as $n = \frac{\varphi - 1}{2}$, with φ the original kernel size (here 7, for a n of 3)
- Do we need the `compression_value` ? As it turn out, because the preceding layer is of size `compression_value`, we can get it for free in the `build` method.

Our `__init__` and `get_config` methods become:

```python
def __init__(self, kernel_size, *args, **kwargs):
    super().__init__(*args, **kwargs)
    self.kernel_size = kernel_size

def get_config(self):
    config = super().get_config()
    config.update({
        "kernel_size": self.kernel_size,
    })
    return config
```

3.2. Second Step, Run-Time Data

Now that we have compilation time data figured out, we can look for run-time only data

- Channel dimension: we can derive this value from the input data, as it is the length of its last dimension (`input_shape[-1]` in the `build` method)
- The convolution layers: we need to construct a series of convolution layers to use in the `call` method. As we should not build this layers in the `call` method, it leaves us we only one choice: the `build` method.

```python
def build(self, input_shape):
    super().build(input_shape)
    bs, w, h, c = input_shape
    self.layers = []
    for _ in range((self.kernel_size - 1) // 2):
        self.layers.append(
            tf.keras.layers.Conv2D(filters=c, kernel_size=(3, 1), padding="same")
        )
        self.layers.append(
            tf.keras.layers.Conv2D(filters=c, kernel_size=(1, 3), padding="same")
        )
        self.layers.append(tf.keras.layers.Activation("relu"))
        self.layers.append(tf.keras.layers.Dropout(0.2))
```

3.3. Third Step, the `call` Method

Finally, the `call` method will apply in series the layers constructed in the `build` method to the inputs data

```python
def call(self, inputs):
    for layer in self.layers:
        inputs = layer(inputs)
    return inputs
```

Write this custom layer, and replace all `for loops` in the code with this layer (it's used exactly as any other layer).

4. An Encoder/Decoder Layer

This time, we will generate the Encoder/Decoder sequentialy separated convolutional layer.

Start with a fresh custom layer code

```
class EDSequentialySeparatedConv2D(tf.keras.layers.Layer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def call(self, inputs):
        return inputs

    def get_config(self):
        config = super().get_config()
        config.update({})
        return config

    def build(self, input_shape):
        super().build(input_shape)
        bs, w, h, c = input_shape
```

4.1. `__init__` & `get_config` Methods

This layer will need two compile-time inputs:

- The kernel_size (as before)
- The compression value for the Encoder/Decoder part.

Modify the `__init__` and `get_config` methods to accomodate for the two variables.

4.2. `build` method

The `build` method will construct three layers,

- A compression layer (a Conv2D layer with kernel 1×1)
- A decompression layer (a Conv2D layer with kernel 1×1)
- A middle `SequentialySeparatedConv2D` layer from before.

See the actual code for compression and decompression layers.

Write the `build` method with the three layers into the `self` instance.

4.3. `call` method

This one is pretty easy, just apply the built layers sequentialy to the inputs.

Write the `call` method.

4.4. The Actual Model Code

Now to the code !

Rewrite the model code to accomodate for the newly created custom layer.

5. A Residual Layer

At this point, you know the drill !

Write a `ResidualEDSequentialseparatedConv2D` layer and change your model code to accomodate for it.

6. How About The Last `for loop` ?

At the end of the model, there is a still this piece of code (if you put a `7x7` kernel-size before):

```
# project data into 10 classes (but keep width and heigth)
for _ in range(3):
    hidden_layer = tf.keras.layers.Conv2D(
        filters=10, kernel_size=(3, 1), padding="same"
    )(hidden_layer)
    hidden_layer = tf.keras.layers.Conv2D(
        filters=10, kernel_size=(1, 3), padding="same"
    )(hidden_layer)
    hidden_layer = tf.keras.layers.Activation("relu")(hidden_layer)
    hidden_layer = tf.keras.layers.Dropout(0.2)(hidden_layer)
```

And it looks awfully like a `SequentialSeparatedConv2D` layer, but one that could benefit from an optional `filters` parameter !

Rewrite the `SequentialSeparatedConv2D` custom layer to accept an optional parameter and change the code to use this layer.

7. A Final Question

Give your impressions about the code, is it more or less readable than before ?