C1W4_Assignment

December 23, 2020

1 Week 4 Assignment: Create a VGG network

In this exercise, you will build a class that implements a VGG network and then train it to classify images of cats and dogs. The model will look something like this:

It is primarily made up of a series of Conv2D layers followed by a softmax activated layers to classify the image. As you can see, this will be a handful and the code will look huge if you specify each layer individually. As shown in the lectures, you can instead use model subclassing to build complex architectures. You can encapsulate repeating parts of a network then reuse that code when building the final model. You will get to practice that in this exercise. Let's get started!

[1]:
```
import tensorflow as tf
import tensorflow_datasets as tfds
import utils
```

1.1 Create named-variables dynamically

In this assignment, you will see the use of the Python function `vars()`. This will allow you to use a for loop to define and set multiple variables with a similar name, such as var1, var2, var3.

Please go through the following examples to get familiar with `vars()`, as you will use it when building the VGG model. - You'll start by defining a class `MyClass` - It contains one variable `var1`. - Create an object of type `MyClass`.

[2]:
```
# Define a small class MyClass
class MyClass:
    def __init__(self):
        # One class variable 'a' is set to 1
        self.var1 = 1

# Create an object of type MyClass()
my_obj = MyClass()
```

Python classes have an attribute called `__dict__`. - `__dict__` is a Python dictionary that contains the object's instance variables and values as key value pairs.

[3]:
```
my_obj.__dict__
```

[3]:
```
{'var1': 1}
```

If you call `vars()` and pass in an object, it will call the object's `__dict__` attribute, which is a Python dictionary containing the object's instance variables and their values as ke

[4]:
```
vars(my_obj)
```

[4]:
```
{'var1': 1}
```

You may be familiar with adding new variable like this:

[5]:
```
# Add a new instance variable and give it a value
my_obj.var2 = 2

# Calls vars() again to see the object's instance variables
vars(my_obj)
```

[5]:
```
{'var1': 1, 'var2': 2}
```

Here is another way that you can add an instance variable to an object, using `vars()`. - Retrieve the Python dictionary `__dict__` of the object using vars(my_obj). - Modify this `__dict__` dictionary using square bracket notation and passing in the variable's name as a string: `['var3'] = 3`

[6]:
```
# Call vars, passing in the object.  Then access the __dict__ dictionary using␣
 ↪square brackets
vars(my_obj)['var3'] = 3

# Call vars() to see the object's instance variables
vars(my_obj)
```

[6]:
```
{'var1': 1, 'var2': 2, 'var3': 3}
```

Why this is helpful! You may be wondering why you would need another way to access an object's instance variables.
- Notice that when using `vars()`, you can now pass in the name of the variable `var3` as a string. - What if you plan to use several variables that are similarly named (`var4`, `var5` … `var9`) and wanted a convenient way to access them by incrementing a number?

Try this!

[7]:
```
# Use a for loop to increment the index 'i'
for i in range(4,10):
    # Format a string that is var
    vars(my_obj)[f'var{i}'] = 0

# View the object's instance variables!
vars(my_obj)
```

[7]:
```
{'var1': 1,
 'var2': 2,
 'var3': 3,
 'var4': 0,
 'var5': 0,
 'var6': 0,
 'var7': 0,
 'var8': 0,
 'var9': 0}
```

There are couple equivalent ways in Python to format a string. Here are two of those ways: - f-string: f“var{i}” - .format: “var{}”.format(i)

[8]:
```
# Format a string using f-string notation
i=1
print(f"var{i}")

# Format a string using .format notation
i=2
print("var{}".format(i))
```

```
var1
var2
```

You can access the variables of a class inside the class definition using `vars(self)`

[9]:
```
# Define a small class MyClass
class MyClass:
    def __init__(self):
        # Use vars(self) to access the class's dictionary of variables
        vars(self)['var1'] = 1

# Create an object of type MyClass()
my_obj = MyClass()
vars(my_obj)
```

[9]:
```
{'var1': 1}
```

You’ll see this in the upcoming code. Now you’ll start building the VGG network!

1.2 Create a generic VGG block (TODO)

The VGG Network has blocks of layers, where each block has a varied number of layers. - In order to create blocks of layers that have a customizable number of conv2D layers, you’ll define a class `Block`, which can generate a customizable block of layers

1.2.1 `__init__`

In the constructor `__init__`, store the conv2D parameters and also define the number of conv2D layers using the parameters passed into `__init__`. - Store the filters, kernel_size, and repetitions as class variables so that they can be used later in the `call` function. - Using a for loop, define a number of Conv2D Conv2D layers, based on the number of `repetitions` desired for this block. - You can define each conv2D layer using `vars` and string formatting to create conv2D_0, conv2D_1, conv2D_3 etc. - Set these four parameters of Conv2D: - filters - kernel_size - activation: set this to 'relu' - padding: set this to 'same' (default pading is 'valid').

- Define the MaxPool2D layer that follows these Conv2D layers.
  - Set the following parameters for MaxPool2D:
    - pool_size: this will be a tuple with two values.
    - strides: this will also be a tuple with two values.

1.2.2 `call`

In `call`, you will connect the layers together. - The 0-th conv2D layer, `conv2D_0`, immediately follows the `inputs`. - For conv2D layers 1,2 and onward, you can use a for loop to connect conv2D_1 to conv2D_0, and connect conv2D_2 to conv2D_1, and so on. - After connecting all of the conv2D_i layers, add connect the max_pool layer and return the max_pool layer.

[64]:
```
## Please uncomment all lines in this cell and replace those marked with `# YOUR CODE HERE`.
## You can select all lines in this code cell with Ctrl+A (Windows/Linux) or Cmd+A (Mac), then press Ctrl+/ (Windows/Linux) or Cmd+/ (Mac) to uncomment.



class Block(tf.keras.Model):

    def __init__(self, filters, kernel_size, repetitions, pool_size=2, strides=2):
        super(Block, self).__init__()
        self.filters = filters
        self.kernel_size =kernel_size
        self.repetitions = repetitions


#         # Define a conv2D_0, conv2D_1, etc based on the number of repetitions
        for i in range(repetitions):
```

```
            vars(self)[f'conv2D_{i}'] =tf.keras.layers.Conv2D(self.filters,␣
↪self.kernel_size, padding='same',
                                                              activation='relu')
            #self.act = tf.keras.layers.Activation('relu')


#         # Define the max pool layer that will be added after the Conv2D blocks
        self.max_pool =tf.keras.layers.MaxPool2D(pool_size, strides=strides)

    def call(self, inputs):


#         # access the class's conv2D_0 layerconv2D_0 = # YOUR CODE HERE

#         # Connect the conv2D_0 layer to inputs
        x =self.conv2D_0(inputs)

#         # for the remaining conv2D_i layers from 1 to `repetitions` they will␣
↪be connected to the previous layer
        for i in range(1 , self.repetitions):

#             # access conv2D_i by formatting the integer `i`. (hint: check how␣
↪these were saved using `vars()` earlier)
            conv2D_i =vars(self)[f'conv2D_{i}']

#             # Use the conv2D_i and connect it to the previous layer
            x = conv2D_i(x)

#         # Finally, add the max_pool layer
        max_pool =self.max_pool(x)

        return max_pool
```

[65]:

[ ]:

[66]:
```
utils.test_block_class(Block)
```

```
All public tests passed
```

[ ]:

1.3 Create the Custom VGG network (TODO)

This model stack has a series of VGG blocks, which can be created using the `Block` class that you defined earlier.

1.3.1 `__init__`

- Recall that the `__init__` constructor of `Block` takes several function parameters,
  - filters, kernel_size, repetitions: you'll set these.
  - kernel_size and strides: you can use the default values.
- For blocks a through e, build the blocks according to the following specifications:
- block_a: 64 filters, kernel_size 3, repetitions 2
- block_b: 128 filters, kernel_size 3, repetitions 2
- block_c: 256 filters, kernel_size 3, repetitions 3
- block_d: 512 filters, kernel_size 3, repetitions 3
- block_e: 512 filters, kernel_size 3, repetitions 3

After block 'e', add the following layers: - flatten: use Flatten. - fc: create a fully connected layer using Dense. Give this 256 units, and a `'relu'` activation. - classifier: create the classifier using a Dense layer. The number of units equals the number of classes. For multi-class classification, use a `'softmax'` activation.

1.3.2 `call`

Connect these layers together using the functional API syntax: - inputs - block_a - block_b - block_c - block_d - block_e - flatten - fc - classifier

Return the classifier layer.

[67]:
```
class MyVGG(tf.keras.Model):

    def __init__(self, num_classes):
        super(MyVGG, self).__init__()

        # Creating blocks of VGG with the following
        # (filters, kernel_size, repetitions) configurations
        self.block_a = Block(filters=64, kernel_size=3, repetitions=2)
        self.block_b = Block(filters=128, kernel_size=3, repetitions=2)
        self.block_c = Block(filters=256, kernel_size=3, repetitions=3)
        self.block_d = Block(filters=512, kernel_size=3, repetitions=3)
        self.block_e = Block(filters=512, kernel_size=3, repetitions=3)

        # Classification head
        # Define a Flatten layer
        self.flatten = tf.keras.layers.Flatten()
        # Create a Dense layer with 256 units and ReLU as the activation␣
↪function
```

```
        self.fc = tf.keras.layers.Dense(256, activation='relu')
        # Finally add the softmax classifier using a Dense layer
        self.classifier =tf.keras.layers.Dense(num_classes,␣
↪activation='softmax')

    def call(self, inputs):
        # Chain all the layers one after the other
        x = self.block_a(inputs)
        x = self.block_b(x)
        x = self.block_c(x)
        x = self.block_d(x)
        x = self.block_e(x)
        x = self.flatten(x)
        x = self.fc(x)
        x = self.classifier(x)
        return x
```

[68]:
```
utils.test_myvgg_class(MyVGG, Block)
```

```
All public tests passed
```

1.3.3 Load data and train the VGG network (Optional)

You can now load the dataset and proceed to train your VGG network. - This will take a few minutes to complete and is **not required to complete the assignment**. - You can submit your work before starting the training.

[69]:
```
dataset = tfds.load('cats_vs_dogs', split=tfds.Split.TRAIN, data_dir='data/')

# Initialize VGG with the number of classes
vgg = MyVGG(num_classes=2)

# Compile with losses and metrics
vgg.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
↪metrics=['accuracy'])

# Define preprocessing function
def preprocess(features):
    # Resize and normalize
    image = tf.image.resize(features['image'], (224, 224))
    return tf.cast(image, tf.float32) / 255., features['label']

# Apply transformations to dataset
dataset = dataset.map(preprocess).batch(32)

# Train the custom VGG model
```

```
vgg.fit(dataset, epochs=10)
```

```
Epoch 1/10
      2/Unknown - 35s 18s/step - loss: 0.6942 - accuracy: 0.4062


      ␣
↪---------------------------------------------------------------------------

       KeyboardInterrupt                         Traceback (most recent call␣
↪last)

       <ipython-input-69-a97d1c255550> in <module>
        17
        18 # Train the custom VGG model
   ---> 19 vgg.fit(dataset, epochs=10)


       /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/keras/
↪engine/training.py in fit(self, x, y, batch_size, epochs, verbose, callbacks,␣
↪validation_split, validation_data, shuffle, class_weight, sample_weight,␣
↪initial_epoch, steps_per_epoch, validation_steps, validation_freq,␣
↪max_queue_size, workers, use_multiprocessing, **kwargs)
       817         max_queue_size=max_queue_size,
       818         workers=workers,
   --> 819         use_multiprocessing=use_multiprocessing)
       820
       821   def evaluate(self,


       /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/keras/
↪engine/training_v2.py in fit(self, model, x, y, batch_size, epochs, verbose,␣
↪callbacks, validation_split, validation_data, shuffle, class_weight,␣
↪sample_weight, initial_epoch, steps_per_epoch, validation_steps,␣
↪validation_freq, max_queue_size, workers, use_multiprocessing, **kwargs)
       340                 mode=ModeKeys.TRAIN,
       341                 training_context=training_context,
   --> 342                 total_epochs=epochs)
       343             cbks.make_logs(model, epoch_logs, training_result,␣
↪ModeKeys.TRAIN)
       344


       /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/keras/
↪engine/training_v2.py in run_one_epoch(model, iterator, execution_function,␣
↪dataset_size, batch_size, strategy, steps_per_epoch, num_samples, mode,␣
↪training_context, total_epochs)
```

```
      126         step=step, mode=mode, size=current_batch_size) as batch_logs:
      127       try:
  --> 128         batch_outs = execution_function(iterator)
      129       except (StopIteration, errors.OutOfRangeError):
      130         # TODO(kaftan): File bug about tf function and errors.
↪OutOfRangeError?


      /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/keras/
↪engine/training_v2_utils.py in execution_function(input_fn)
       96     # `numpy` translates Tensors to values in Eager mode.
       97     return nest.map_structure(_non_none_constant_value,
  ---> 98                               distributed_function(input_fn))
       99
      100   return execution_function


      /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/eager/
↪def_function.py in __call__(self, *args, **kwds)
      566         xla_context.Exit()
      567     else:
  --> 568       result = self._call(*args, **kwds)
      569
      570     if tracing_count == self._get_tracing_count():


      /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/eager/
↪def_function.py in _call(self, *args, **kwds)
      597       # In this case we have created variables on the first call, so␣
↪we run the
      598       # defunned version which is guaranteed to never create␣
↪variables.
  --> 599       return self._stateless_fn(*args, **kwds)  # pylint:␣
↪disable=not-callable
      600     elif self._stateful_fn is not None:
      601       # Release the lock early so that multiple threads can perform␣
↪the call


      /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/eager/
↪function.py in __call__(self, *args, **kwargs)
     2361     with self._lock:
     2362       graph_function, args, kwargs = self.
↪_maybe_define_function(args, kwargs)
  -> 2363     return graph_function._filtered_call(args, kwargs)  # pylint:␣
↪disable=protected-access
     2364
```

```
      2365   @property


       /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/eager/
↪function.py in _filtered_call(self, args, kwargs)
      1609          if isinstance(t, (ops.Tensor,
      1610                            resource_variable_ops.
↪BaseResourceVariable))),
   -> 1611         self.captured_inputs)
      1612
      1613   def _call_flat(self, args, captured_inputs,␣
↪cancellation_manager=None):


       /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/eager/
↪function.py in _call_flat(self, args, captured_inputs, cancellation_manager)
      1690       # No tape is watching; skip to running the function.
      1691       return self._build_call_outputs(self._inference_function.call(
   -> 1692           ctx, args, cancellation_manager=cancellation_manager))
      1693     forward_backward = self._select_forward_and_backward_functions(
      1694         args,


       /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/eager/
↪function.py in call(self, ctx, args, cancellation_manager)
       543               inputs=args,
       544               attrs=("executor_type", executor_type, "config_proto",␣
↪config),
   --> 545               ctx=ctx)
       546         else:
       547           outputs = execute.execute_with_cancellation(


       /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/eager/
↪execute.py in quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
        59     tensors = pywrap_tensorflow.TFE_Py_Execute(ctx._handle,␣
↪device_name,
        60                                                op_name, inputs,␣
↪attrs,
   ---> 61                                                num_outputs)
        62   except core._NotOkStatusException as e:
        63     if name is not None:


       KeyboardInterrupt:
```

[ ]: