Chapter 4

Writing and Structuring Deep Learning Code

In this chapter:

- What are some best practices when writing deep learning code
- How to unit test your code
- How to debug your code

This chapter is all about the development of a production-ready deep learning project. We will start off by examining deep learning coding examples from a software engineering perspective. Afterwards, we will discuss best practices that you can follow when writing code, as well as the tools to include in your arsenal to incorporate these practices.

4.1 Best practices

When referring to best practices, we usually talk about a set of "rules" one should follow. The word "rules" is in quotation marks because these aren't exactly rules. They are well-established guidelines and workflows that many teams have successfully used before and they all indirectly agree that it's the best way of doing things. However, notice that some of these practices come from my own experience and might not resonate with you. In that case, feel free to disregard them.

4.1.1 Project structure

One very important aspect when writing code is how we structure our project. A good structure should obey the "**Separation of concerns**" principle in terms that **each functionality should be a distinct component**. In this way, it can be easily modified and extended without breaking other parts of the code. Moreover, it can be reused in many places without the need to write duplicate code.

Tip: Writing the same code once is perfect, twice is kind of fine but thrice it's not. DRY (Don't Repeat Yourself) is a commonly used acronym that developers use to indicate that a piece of code or a specific functionality is not duplicated. (e.g. we should DRY this function)

The way I like to organize most of my deep learning projects is something like that:

```
$ tree -L 1
.
|------configs
|------dataloader
|------evaluation
|------executor
|------model
|------notebooks
|------ops
|------utils
```

And that, of course, is my personal preference. Feel free to play around with this until you find what suits you best.

Python modules and packages

Notice that in the project structure that I presented, each folder is a separate module that can be imported into other modules just by writing `import module`. Here, we should make a quick distinction between what python calls **module** and what **package**.

A module is simply a `.py` file containing Python code. A package, however, is like a directory that holds sub-packages and modules. In order for a package to be importable, it should contain an `__init__.py` file (even if it's empty). That is not the case for modules. In our case, each folder is a package and it contains an `__init__.py` file.

In our example, we have 8 different packages:

1. **configs**: In this module, we define everything that can be configurable and can be changed in the future. Good examples are training hyperparameters, folder paths, metrics, flags, etc.

2. **dataloader** is quite self-explanatory. All the data loading and data pre-processing classes and functions live here.

3. **evaluation** is a collection of code files that aims to evaluate the performance and accuracy of our model.

4. **executor**: in this folder, we usually have all the functions and scripts that train the model or use it for prediction in different environments. And by different environments, I mean executors for CPU-only systems, executors for GPUs, executors for distributed systems. This package is our connection with the outer world and it's what our `main.py` will use.

5. **model** contains the actual deep learning code (we are talking about Tensorflow, Pytorch, etc).

6. **notebooks** include all our Jupyter/Colab notebooks in one place that we built in the experimentation phase of the machine learning lifecycle.

7. **ops**: This one is not always needed, as it includes operations not related with machine learning such as algebraic transformations, image manipulation techniques or maybe graph operations.

8. **utils**: Utility functions that are used in more than one place. In essence, everything that doesn't belong in the pre-described categories comes here.

4.1.2 Object-oriented programming

Now that we have our project well-structured, we can begin to discover how our code should look like on a lower level.

The answer to that question is **classes** and everything that comes with this. Admittedly, **Object-Oriented Programming (OOP)** might not be the first thing that comes to mind when writing Python code (it definitely is when coding in Java or C#), but you will be surprised by how easy it is to develop software when thinking in objects.

Tip: A good way to code your way is to try and write Python the same way you would write Java.

Yes, I know that's not your usual advice, but you'll understand what I mean in time. If we reprogram the UNet model, presented in section 2.6, in an object-oriented way, the result would be something like this:

```
class UNet():

  def __init__(self, config):
    self.base_model = tf.keras.applications.MobileNetV2(
```

```
      input_shape = self.config.model.input, include_top=False
    )
    self.batch_size = self.config.train.batch_size
    . . .

  def load_data(self):
    """Loads and Preprocess data """
    self.dataset, self.info =
      DataLoader().load_data(self.config.data)
    self._preprocess_data()

  def _preprocess_data(self):
    . . .

  def _set_training_parameters(self):
    . . .

  def _normalize(self, input_image, input_mask):
    . . .

  def _load_image_train(self, datapoint):
    . . .

  def _load_image_test(self, datapoint):
    . . .

  def build(self):
    """ Builds the Keras model based """
    layer_names = [
      'block_1_expand_relu',   # 64x64
      'block_3_expand_relu',   # 32x32
      'block_6_expand_relu',   # 16x16
      'block_13_expand_relu',   # 8x8
      'block_16_project',   # 4x4
    ]
    layers =
      [self.base_model.get_layer(name).output for name in layer_names]

     . . .

    self.model = tf.keras.Model(inputs=inputs, outputs=x)
```

```
  def train(self):
    . . .

  def evaluate(self):
    . . .
```

If you compare the new code with the original one, you will start to understand what these practices accomplish.

Basically, you can see that **the model is a class, each separate functionality is encapsulated within a method, and all the common variables are declared as instance variables**.

As you can easily realize, it becomes much easier to alter the training functionality of our model, to change the layers or to flip the default of a boolean variable. On the other hand, writing spaghetti code (which is a programming slang for chaotic code) is something that should be avoided. Because it is much more difficult to find the responsibility of each function, or if a change affects other parts of the code, or how to debug the software.

As a result, we get free **maintainability, extensibility, and simplicity**.

Abstraction and Inheritance

However, using classes and objects give us a lot more than that. Abstraction and inheritance are two of the most important topics in OOP.

By using **abstraction** we can declare the desired functionalities without dealing with how they are going to be implemented. To this end, we can first think about the logic behind our code and then dive into programming every single part of it. The same pattern can easily be adopted in deep learning code. To better understand what I'm saying, have a look at the code below:

```
class BaseModel(ABC):
  """Abstract Model class that is inherited to all models"""
  def __init__(self, cfg):
    self.config = Config.from_json(cfg)

  @abstractmethod
  def load_data(self):
    pass

  @abstractmethod
  def build(self):
```

```
    pass

  @abstractmethod
  def train(self):
    pass

  @abstractmethod
  def evaluate(self):
    pass
```

One can easily observe that the **functions have no body**. They are just a declaration. In the same logic, you can think of every functionality you are going to need, declare it as an abstract method or class, and you are done. It's like having a contract of what the code should look like. That way you can decide first on the high-level implementation and then tackle each part in detail.

Consequently, that contract can now be used by other classes that will "extend" our abstract class. This is called **inheritance** . The base class will be inherited in the "child" class and it will immediately define its structure. So, the new class is obligated to have all the abstract functions as well. Of course, it can also have many other functions, not declared in the abstract class.

Look below how we pass the `BaseModel` class as an argument in the `UNet`. That's all we need. Also in our case, we need to call the `__init__` function of the parent class, which we accomplish with the `super()`. `super` is a special Python function that calls the constructor (the function that initializes the object aka the `__init__`) of the parent class. The rest of the code is normal deep learning code.

The main way to have abstraction in Python is by using the ABC library.

```
class UNet(BaseModel):
  """Unet Model class. Contains functionality for building,
  training and evaluating the model"""


  def __init__(self, config):
    super().__init__(config)
    self.base_model = tf.keras.applications.MobileNetV2(
      input_shape=self.config.model.input, include_top=False
    )

    . . .
```

```
  def load_data(self):
    self.dataset, self.info =
      DataLoader().load_data(self.config.data )
    self._preprocess_data()

    . . .

  def build(self):

    . . .

    self.model = tf.keras.Model(inputs=inputs, outputs=x)

  def train(self):
    self.model.compile(
      optimizer = self.config.train.optimizer.type
      loss = tf.keras.losses.SparseCategoricalCrossentropy(
          from_logits = True
      ),
      metrics = self.config.train.metrics
    )

    model_history = self.model.fit(
      self.train_dataset,
      epochs = self.epoches,
      steps_per_epoch = self.steps_per_epoch,
      validation_steps = self.validation_steps,
      validation_data=self.test_dataset
    )

    return model_history.history['loss'],model_history.history['val_loss']

  def evaluate(self):
    predictions = []
    for image, mask in self.dataset.take(1):
      predictions.append(self.model.predict(image))

    return predictions
```

Moreover, you can imagine that the base class can be inherited by many children (that's called **polymorphism**). That way we can have many models with the same base model as

a parent but with different logic. In another practical aspect, if a new developer joins our team, he can easily find out what his code should look like just by inheriting our abstract class.

Static and class methods

Static and class methods are two interesting patterns that can simplify and make our code less error-prone. In OOP, we have the class and the instances. **The class is like the blueprint for creating objects. Instances are the actual objects that have the class as type**.

Class methods take as arguments the actual class and they are usually used as constructors for creating new instances of the class. You will understand what this means in the next paragraphs where we present some coding examples.

Static methods are methods that refer only to the class, not its instances, and they will not modify the class state. Static methods are mostly used for utility functions that aren't going to change.

Definition: The class state of an object is the values of all its variables.

Let's see how they can be applied in our code:

The below code snippet constructs a config from a json file. The **classmethod** `from_json` returns an actual instance of the `Config` class created from a json file by calling `Config.from_json(jso`
In other words, it is a constructor of our class. By doing this, we can have multiple constructors for the same class (we might for example want to create a config using a yaml file).

```
class Config:
  """
  Config class which contains data, train and model hyperparameters
  """

  def __init__(self, data, train, model):
    self.data = data
    self.train = train
    self.model = model

  @classmethod
  def from_json(cls, cfg):
    """Creates config from json"""
    params = json.loads(
      json.dumps(cfg), object_hook=HelperObject
```

```
    )
    return cls(params.data, params.train, params.model)
```

Static methods, on the other hand, are methods that are called on the actual object and not an instance of it. A perfect example is the `DataLoader` class, where we load the data from an external URL. Is there a reason to have a new `DataLoader` instance? Not really, because everything is stable in this functionality and nothing will ever change. When states are changing, it is better to class instances and instance methods.

```
class DataLoader:
  """Data Loader class. Loads the data as a tfds dataset"""

  @staticmethod
  def load_data(data_config):
    """Loads dataset from path"""
    return tfds.load(
       data_config.path, with_info=data_config.load_with_info
    )
```

4.1.3 Configuration

Configuration files (commonly known as config files) are files used to configure the parameters and initial settings for computer programs. They differ in syntax and format but most of them are very readable and easily modifiable.

It is generally recommended to have all settings in a single place so they can be changed seamlessly. As an example, take a look on the config file below:

```
CFG = {
   "data": {
       "path": "oxford_iiit_pet:3.*.*",
       "image_size": 128,
       "load_with_info": True
   },
   "train": {
       "batch_size": 64,
       "buffer_size": 1000,
       "epoches": 20,
       "val_subsplits": 5,
       "optimizer": {
           "type": "adam"
       },
```

```
        "metrics": ["accuracy"]
    },
    "model": {
        "input": [128, 128, 3],
        "up_stack": {
            "layer_1": 512,
            "layer_2": 256,
            "layer_3": 128,
            "layer_4": 64,
            "kernels": 3
        },
        "output": 3
    }
}
```

Whenever we want to change the batch size of our data, the optimization algorithm or the number of nodes in a layer, we can immediately come here.

4.1.4 Type checking

Another cool and useful feature, borrowed again from Java, is type checking. Type checking is the process of verifying and enforcing the constraints of types. And by types we mean whether a variable is a string, an integer or an object. To be more precise, in Python we have **type hints**. Python doesn't support type checking, because it is a dynamically typed language. But we will see how to get around that. Type checking is essential because it can acknowledge bugs and errors very early and can help us write better code overall.

It is very common when coding in Python to have moments where you wonder if a particular variable is a string or an integer. And you find yourself tracking it throughout the code trying to figure out what type it is. And that's much trickier to do when that code is implemented with Tensorflow or Pytorch.

A very simple way to do type checking can be seen below:

```
def ai_summer_func(x:int) -> int:
    return x+5
```

As you can see, we declare that both x and the function's return value should be of type integer.

Note that this will not throw an error on an exception. It is just a suggestion. IDEs like PyCharm (or Python linters) will automatically discover them and show a warning. That way we can easily detect bugs and fix them as we are building our code.

If we want to catch these kinds of errors, we can use a static type checker like `Pytype` [1]. After installing it and including our type hints in our code, we can run something like below and it will show us all the type errors in our code. `Pytype` is used in many Tensorflow official codebases and it's a Google library. An example can be illustrated below:

```
$ pytype main.py

File "/home/aisummer/PycharmProjects/Deep-Learning-Production-Course
/main.py", line 19, in <module>: Function ai_summer_func was called
with the wrong arguments [wrong-arg-types]
  Expected: (x: int)
  Actually passed: (x: str)
```

One important thing that I need to mention here is that checking types in Tensorflow code is not easy. Without getting too deep into that, you can't simply define the type of x as a `tf.Tensor`. Type checking is great for simpler functions and basic data types but when it comes to Tensorflow code things can be hard. Pytype has the ability to infer some types from your code and it can resolve types such as Tensor or Module, but it doesn't always work as expected.

Linting tools such as Pylint [2] can also be great for finding type errors from our IDE. **Linting** is the automated checking of your source code for programmatic and stylistic errors. This is done using a lint tool (otherwise known as linter) and the output is usually displayed inside the code editor or in the terminal.

4.1.5 Documentation

Documenting our code is the single most important thing in this list and the thing that most of us are guilty of not doing. Writing simple comments on our code can make the life of our teammates but also of our future selves much easier. It is even more important when we write deep learning code because of the complex nature of our software. In the same sense, it's equally important to give proper and descriptive names in our classes, functions and variables. Take a look at this:

```
def n(self, ii, im):
    ii = tf.cast(ii, tf.float32) / 255.0
    im -= 1
    return ii, im
```

I'm 100% certain that you have no idea what it does.

[1] Pytype: https://github.com/google/pytype
[2] Pylint: https://pylint.org/

Now look at this:

```
def _normalize(self, input_image, input_mask):
  """ Normalise input image
  Args:
    input_image (tf.image): The input image
    input_mask (int): The image mask

  Returns:
    input_image (tf.image): The normalized input image
    input_mask (int): The new image mask
  """

  input_image = tf.cast(input_image, tf.float32) / 255.0
  input_mask -= 1
  return input_image, input_mask
```

Can it be more descriptive?

The comments you can see above are called **docstrings** and are Python's way to document the responsibility of a piece of code. I usually include a docstring at the beginning of a module (python file), indicating the purpose of a file, under every class declaration, and inside every function.

There are many ways to format the docstrings. I personally prefer the google style, which looks like the above code.

The first line always indicates what the code does, and it is the only real essential part. I personally try to always include this in my code. The other parts explain what arguments the function accepts (with types and descriptions) and what it returns. These can sometimes be ignored if they don't provide much value. Note that types provided in docstrings are different from type hints and a linter won't usually flag them.

4.2 Unit testing

Programming a deep learning model is not easy. I'm not going to lie to you. However, testing one is even harder. That's why most of the TensorFlow and Pytorch code out there does not include unit tests. But when your code is going to live in a production environment, making sure that it actually does what is intended, should be a priority. After all, machine learning is not different from any other software.

In this section, we are going to focus on how to properly test machine learning code, analyse a collection of best practices when writing unit tests, and present a number of example cases

where testing is almost a necessity. We will start on why we need them in our code, then we will do a quick catch up on the basics of testing in Python. Finally we will go over different practical real-life scenarios.

Why do we need unit testing?

When developing a neural network, most of us don't care about catching all possible exceptions, finding all corner cases, or debugging every single function. We just need to see our model fitting. And then we just need to increase its accuracy until it reaches an acceptable percentage. That's all good but what happens when the model will be deployed into a server and used in an actual public-facing application? Most likely it will crash because some users may be sending wrong data or because of a silent bug that messes up our data pre-processing pipeline. We might even discover that our model was in fact corrupted all this time.

This is where unit tests come into play. To prevent all these things before they even occur. Unit tests are tremendously useful because they:

1. Find software bugs early.
2. Debug our code.
3. Ensure that the code does what it's supposed to do.
4. Simplify the refactoring process.
5. Speed up the integration process.
6. Act as documentation.

Don't tell me that you don't want at least some of the above. Sure, testing can take up a lot of our precious time but it's 100% worth it.

But what exactly is a unit test?

4.2.1 Basics of unit testing

In simple terms, **a unit test is basically a function calling another function (or a unit) and checking if the values returned match the expected output**. Let's see an example using our UNet model to make it clearer.

We have this simple function in our data pipeline that normalizes an image by dividing all the pixels by 255.

```
def _normalize(self, input_image, input_mask):
  """ Normalise input image
  Args:
```

```
    input_image (tf.image): The input image
    input_mask (int): The image mask

  Returns:
    input_image (tf.image): The normalized input image
    input_mask (int): The new image mask
  """

  input_image = tf.cast(input_image, tf.float32) / 255.0
  input_mask -= 1
  return input_image, input_mask
```

To make sure that it does exactly what it is supposed to do, we can write another function that uses `normalize()` and check its result. It will look something like this.

```
def test_normalize(self):
  input_image = np.array([[1., 1.], [1., 1.]])
  input_mask = 1
  expected_image = np.array(
   [[0.00392157, 0.00392157], [0.00392157, 0.00392157]]
  )

  result = self.unet._normalize(input_image, input_mask)
  self.assertEquals(expected_image, result[0])
```

The `test_normalize()` function creates a fake input image, calls the function with that image as an argument, and then makes sure that the result is equal to the expected image. `assertEquals` is a special function, coming from the `unittest` package in Python and does exactly what its name suggests. It asserts that the two values are equal. Note that we can also use something like below, but using built-in functions has its advantages.

```
assert expected_image == result[0]
```

That's it. That's unit testing. Tests can be used on both very small functions and bigger, complicated functionalities across different modules. In the context of machine learning, we can test the deep learning models and all the surrounding components to make sure that the entire pipeline works as expected.

4.2.2 Unit tests in Python

Before we see more examples, I'd like to do a quick catch up on how Python supports unit testing.

The main testing framework/runner that comes into Python's standard library is `unittest`. `unittest` is pretty straightforward to use and it has only two requirements: to put your tests into a class and use its special assert functions. A simple example can be found below:

```
import unittest

class UnetTest(unittest.TestCase):

    def test_normalize(self):
        . . .

if __name__ == '__main__':
    unittest.main()
```

Some things to notice here:

1. We have our test class which includes a `test_normalize` function as a method. In general, test functions are named with `test_` as a prefix followed by the name of the function they test. (This is a convention, but it also enables `unittest`'s **autodiscovery** functionality, which is the ability of the library to automatically detect all unit tests within a project or a module. That way we won't have to run them one by one).

2. To run unit tests, we call the `unittest.main()` function which discovers all tests within the module, runs them and prints their output.

3. Our `UnetTest` class inherits the `unittest.TestCase` class. This class helps us set unique test cases with different inputs because it comes with `setUp()` and `tearDown()` methods. In `setUp()` we can define our inputs that can be accessed by all tests, and in `tearDown()` we can dissolve them (see snippet in the next section). This is helpful because all tests should run independently and usually, they can't share information. Well, now they can.

Another two powerful frameworks are `pytest` [3] and `nose` [4], which are pretty much governed by the same principles. I suggest playing with them a little before you decide what suits you best. I personally use `pytest` most of the times, because it feels a bit simpler and it supports a few nice to have things, like fixtures and test parameterization. But honestly, it doesn't have that big of a difference so you should be fine with either of them.

Sadly, unit testing in Tensorflow is not straightforward. For that reason, in the next section, I'm going to discuss another, lesser-known method.

[3] Pytest: https://docs.pytest.org/en/6.2.x/
[4] Nose: https://nose.readthedocs.io/en/latest/

4.2.3 Tests in Tensorflow

Since we use Tensorflow to program our model we can take advantage of `tf.test`, which is an extension of `unittest` but it contains assertions tailored to Tensorflow code. In that case, our code morphs into this:

```
import tensorflow as tf

class UnetTest(tf.test.TestCase):

  def setUp(self):
    super(UnetTest, self).setUp()
    . . .

  def tearDown(self):
    pass

  def test_normalize(self):
    . . .

if __name__ == '__main__':
    tf.test.main()
```

Did you notice anything familiar? Actually, it has exactly the same baselines with the caveat that we need to call the `super()` function inside `setup()`, which enables `tf.test` to do its magic. Pretty cool, right?

4.2.4 Mocking

Another super important topic we should be aware of is mocking and mock objects. Mocking classes and functions are common when writing Java but in Python they are underutilized. **Mocking makes it very easy to replace complex logic or heavy dependencies when testing code using dummy objects**. By dummy objects, we refer to simple, easy to code objects that have the same structure with our real objects but contain fake or useless data. In our image segmentation case, a dummy object might be a 4-dimensional tensor with all values equal to 1, which mimics an actual image.

Mocking also helps us control the code's behaviour and simulate expensive calls. Let's look at an example using once again our UNet model.

Let's assume that we want to make sure that the data pre-processing step is correct and that our code splits the data and creates the training and testing dataset as it should. This is a common real-life test case. Here is the code we want to test:

```
def load_data(self):
  """ Loads and Preprocess data """
  self.dataset, self.info =
  DataLoader().load_data(self.config.data)
  self.preprocess_data()

def _preprocess_data(self):
  """ Splits into training and test and set training parameters"""
  train = self.dataset['train'].map(
    self.load_image_train,
    num_parallel_calls = tf.data.experimental.AUTOTUNE
  )
  test = self.dataset['test'].map(self._load_image_test)

  self.train_dataset = train
    .cache()
    .shuffle(self.buffer_size)
    .batch(self.batch_size)
    .repeat()

  self.train_dataset = self.train_dataset.prefetch(
    buffer_size=tf.data.experimental.AUTOTUNE
  )
  self.test_dataset = test.batch(self.batch_size)

def _load_image_train(self, datapoint):
  """ Loads and preprocess  a single training image """
  input_image = tf.image.resize(
    datapoint['image'], (self.image_size, self.image_size)
  )
  input_mask = tf.image.resize(
    datapoint['segmentation_mask'],
    (self.image_size, self.image_size)
  )

  if tf.random.uniform(()) > 0.5:
    input_image = tf.image.flip_left_right(input_image)
    input_mask = tf.image.flip_left_right(input_mask)

    input_image, input_mask = self._normalize(
```

```
      input_image, input_mask
    )

    return input_image, input_mask

def _load_image_test(self, datapoint):
  """ Loads and preprocess a single test image"""

  input_image = tf.image.resize(
    datapoint['image'], (self.image_size, self.image_size)
  )
  input_mask = tf.image.resize(
    datapoint['segmentation_mask'],
    (self.image_size, self.image_size)
  )

  input_image, input_mask = self._normalize(
    input_image, input_mask
  )

  return input_image, input_mask
```

This code actually handles the splitting, shuffling, resizing, batching (grouping) of the data. We will analyze it more extensively on Chapter 5. For now, suppose we want to test this code. Everything is nice and well **except the loading function**.

```
self.dataset, self.info = DataLoader().load_data(self.config.data)
```

Are we supposed to load the entire dataset every time we run a single unit test? Absolutely not. To avoid doing that, we could mock that function to return a dummy dataset instead of calling the real one. Mocking to the rescue.

We can do that with `unittests`'s mock object package. It provides a mock class `Mock()` to create a mock object directly and a `patch()` decorator. The decorator replaces an imported module, within the module we test, with a mock object. Ok, so how do we do that?

For those who aren't familiar, the **decorator** is simply a function that wraps another function to extend its functionality.

Once we declare the wrapper function, we can annotate other functions to enhance them. See the `@patch` below? That's a decorator which wraps the `test_load_data()` with the `patch()` function.

By using the `patch()` decorator we get this:

```
@patch('model.unet.DataLoader.load_data')
def test_load_data(self, mock_data_loader):

  mock_data_loader.side_effect = dummy_load_data
  shape = tf.TensorShape(
    [None, self.unet.image_size, self.unet.image_size, 3]
  )

  self.unet.load_data()
  mock_data_loader.assert_called()

  self.assertItemsEqual(
    self.unet.train_dataset.element_spec[0].shape, shape
  )
  self.assertItemsEqual(
    self.unet.test_dataset.element_spec[0].shape, shape
  )
```

What the decorator alongside the `mock_data_loader.side_effect = ...` does, is that the `DataLoader.load_data()` is "patched" by our `dummy_load_data()` function which returns a dummy dataset.

To sum up, instead of calling the actual function, we trigger the dummy function and we save ourselves from waiting for the dataset to be loaded in every single test. Plus, we get to control exactly what our input data should look like.

We can use a handy feature from the `tensorflow_datasets` package to build a mock dataset. This will return a mock dataset instead of the real one. Then we end up having a mock dataset object inside of a mock `load_data()` function. Inception. Or maybe Mockception!

```
import tensorflow_datasets as tfds

def dummy_load_data(*args, **kwargs):
  with tfds.testing.mock_data(num_examples=1):
    return tfds.load(CFG['data']['path'], with_info=True)
```

Remember, `CFG` refers to our configuration. I can tell that you are amazed by this. Don't try to hide it.

If you're still unclear with what we gained here, let me break it down. We managed to create

a dummy object that mimics our entire dataset with a few lines of code. This object can now be used in different unit tests where the actual data are irrelevant to the functionality. We eliminated the need to load our actual dataset into memory just to perform a test.

One last thing we should mention is test coverage.

4.2.5 Test coverage

Before we see some specific testing use cases on machine learning, I would like to mention another important aspect. Coverage. By coverage, we mean how much of our code is actually tested by unit tests.

Coverage is an invaluable metric that can help us write better unit tests, discover which areas our tests don't exercise, find new test cases, and ensure the quality of our tests. We can simply check our test coverage following the steps below:

1) Install the `coverage` [5] package

```
$ conda install coverage
```

2) Run the package in our test file

```
$ coverage run -m unittest
/home/aisummer/PycharmProjects/Deep-Learning-Production-Course/
model/tests/unet_test.py
```

3) Print the results

```
$ coverage report -m
/home/aisummer/PycharmProjects/Deep-Learning-Production-Course/model/
tests/unet_test.py

Name                          Stmts   Miss  Cover   Missing
-------------------------------------------------------------
model/tests/unet_test.py      35       1       97%      52
```

This says that we cover 97% of our code. There are 35 statements in total and we missed just 1 of them. The missing info tells us which lines of code still need coverage. In this way, you can keep track of the percentage of the tested code during your project development.

[5] Coverage: https://coverage.readthedocs.io/en/6.1.1/index.html

4.2.6 Test example cases

I think it's time to explore some of the different deep learning scenarios and parts of the codebase where unit testing can be incredibly useful. Well, I'm not going to write the code for every single one of them, but I think it would be very important to outline a few use cases.

We already discussed one of them. Ensuring that our data has the right format is critical. A few others I can think of are:

Data:

- Ensure that our data has the right format (yes, I put it again here for completion).
- Ensure that the training labels are correct.
- Test our complex processing steps such as image manipulation.
- Assert data completion, quality, and errors.
- Test the distribution of the features.

Training:

- Run a training step and compare the weights before and after, to ensure that they are updated.
- Check that our loss function can be actually used on our data.

Evaluation:

- Having tests to ensure that your metrics (e.g. accuracy, precision, and recall) are above a threshold when iterating over different architectures.
- We can run speed/benchmark tests on training to catch possible overfitting.
- Of course, cross-validation can be in the form of a unit test.

Model Architecture:

- The model's layers are stacking.
- The model's output has the correct shape.

On second thought, let's program the last one to prove to you how simple it is:

```
def test_ouput_size(self):
    shape = (1, self.unet.image_size, self.unet.image_size, 3)
    image = tf.ones(shape)
```

```
    self.unet.build()
    self.assertEqual(self.unet.model.predict(image).shape, shape)
```

That's it. Define the expected shape, construct a dummy input, build the model, and run a prediction is all it takes. Not so bad for such a useful test, right? You see unit tests don't have to be complex. **Sometimes a few lines of code can save us from a lot of trouble**. Trust me. At the same time though, we shouldn't go on the other side and test every single thing imaginable. This is a huge time sink. As always, we need to find a balance.

I am confident that you can come up with many more test scenarios when developing your own models. Now that you have a rough but clear idea what tests are, you can find the ones that suit best to your work.

4.2.7 Integration / acceptance tests

Something that I deliberately avoided mentioning is **Integration and Acceptance Tests (ATs)** . These kinds of tests are very powerful tools and **aim to test how well our system integrates with other systems**. If you have an application with many services or a client/server interaction, acceptance tests are the go-to functionality to make sure that everything works as expected at a higher level.

Later throughout the book, when we deploy our model in a server, we will need to write some acceptance tests as we want to be certain that the model returns what the user/client expects in the form that they expect it. As we iterate over our application while it is live and is served to users, we can't have a failure due to a minor bug. These are the kinds of things that acceptance tests help us avoid.

Unfortunately, you will have to wait for the last chapters to see how we can build an acceptance test. It's quite straightforward and has the form of a unit test, but it requires us to have two separate systems. In our example, we will use a server containing our model and a client. More on that in Chapter 7.

4.3 Debugging

Have you ever been stuck on an error for way too long? I remember once when I spent over 2 weeks on a small typo that didn't crash the program but returned inexplicable results. I literally couldn't sleep because of this. I'm 100% certain that this has happened to you as well, therefore now we will be focusing on how to debug deep learning code and how to use logging to catch bugs. We will of course use Tensorflow to showcase some examples, following our image segmentation project, but the exact same principles apply to Pytorch or other AI frameworks.

As I said at the beginning of the book, **machine learning is ordinary software and should always be treated like one**. And one of the most essential parts of the software development lifecycle is debugging. Proper debugging can help eliminate future pains when our algorithms are been used by real users. It can make our system as robust and reliable as our users expect it to be.

4.3.1 How to a debug deep learning project?

Deep learning debugging is more difficult than normal software because of multiple reasons:

- Poor model performance does not imply bugs in the code.
- The iteration cycle (building the model, training, and testing) is quite long.
- Training/testing data can also have errors and anomalies.
- Hyperparameters affect the final accuracy.
- It's not always deterministic (e.g. probabilistic machine learning).
- Static computation graphs (e.g. Tensorflow 1.0 and CNTK) prevent line by line execution of the code.

Based on the above, the best way to start thinking about debugging is to **simplify the ML model development process as much as possible**. By simplifying, I mean to a ridiculous level. In general, when experimenting with our model, the best practice is to start from a simple algorithm. It is also common to utilize only a handful of features and gradually keep expanding by adding features and tuning hyperparameters while keeping the model simple. Once we find a satisfactory set of features, we can start increasing our model's complexity, keep track of the metrics, and continue incrementally until the results are satisfactory for our application.

In the image segmentation case, we don't really have a choice but to use the image. We can however start with a simple U-shaped convolutional network. There are tons of variations of Unet for image segmentation, but the standard baseline will work just fine as a first step. Furthermore, research papers are too much focused on the modelling part for fixed datasets. This is rarely helpful in a production environment. Now we will mostly care about our data and our model lifecycle.

But even in these case, bugs and anomalies might occur. In fact, they will definitely occur. When they do, our next step is to take advantage of Python's debugging capabilities.

4.3.2 Python's debugger

Python debugger (Pdb) is part of the Python standard library. **The debugger is essentially a program that can monitor the state of our own program while it is running**. The most important command of any debugger is called a breakpoint. We can set a breakpoint anywhere in our code and the debugger will stop the execution at this exact point and give us access to the values of all the variables at that point, as well as the traceback of python calls.

There are two ways to interact with Python's debugger. Command line and IDEs. If you want to use the terminal you can go ahead, but I must warn you that it's quite tedious. You will have to insert the breakpoints inside the code and interact with the debugger through the terminal.

```
import pdb
pdb.set_trace()
```

Since we have used PyCharm throughout the book, we will stay consistent and use it here as well.

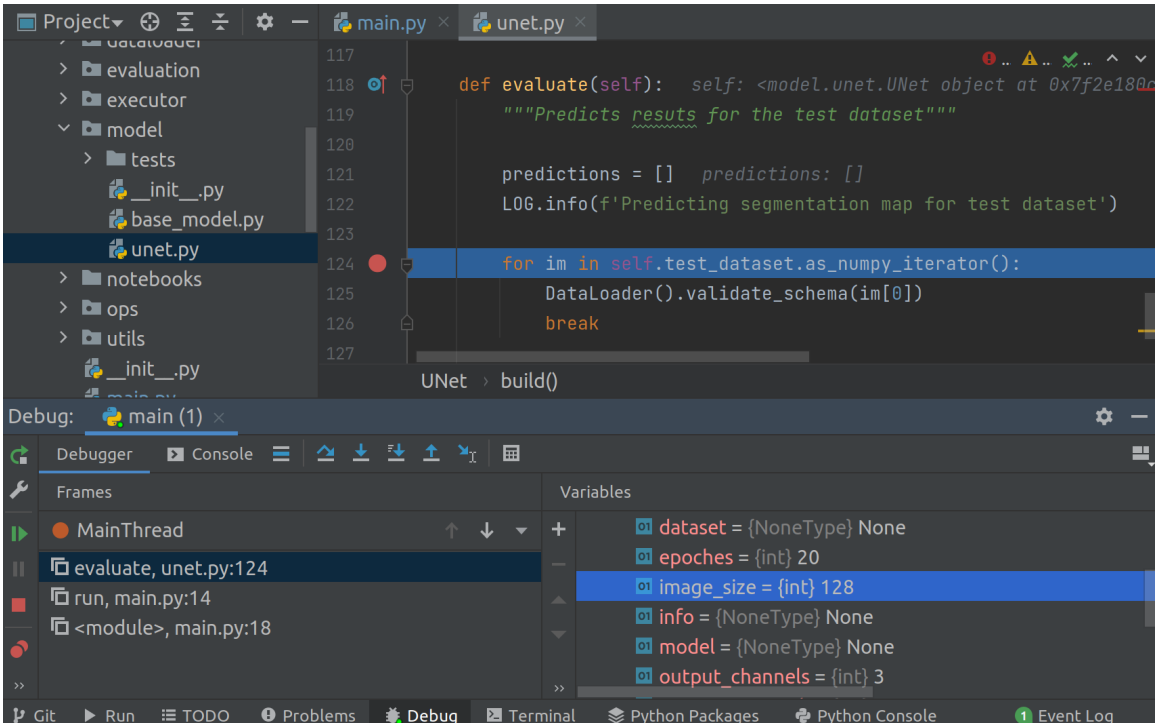

Figure 4.1: Python debugging on Pycharm

Let's have a closer look at the Figure 4.1. As you can see, we have set a breakpoint (the red dot on line 124) at the beginning of the for-loop in the `predict()` function, and we pressed the debug button.

The program then was executed normally until it hit the breakpoint where the debugger paused the state. In the debug window below the code, we can inspect all the variables at this stage of the execution. Let's assume for example that we wanted to debug the size of the input image: as you can see it's 128. Or the number of epoches. Again, it's very easy to spot that it's 20.

Tip: Using the debugger, we can access anywhere, any variable we want. Therefore, we can avoid having print statements all over the place.

From the breakpoint, we can continue to another breakpoint, or we can finish the program's execution. We also have a 3rd option. We can use the step function of the debugger to go into the next code line. And then go to the second next line. That way, we can choose to run our code as slowly as we want until we figure out what is wrong.

4.3.3 Debugging data with schema validation

Now that we have a decent way to find bugs in the code, we should have a look at the second most common source of errors in machine learning: data. As you can imagine, data aren't always in perfect form. And in fact they never are in real-life scenarios. Let me give you an example: they may contain corrupted data points, some values may be missing, they may have a different format. They may even have a different range/distribution than expected.

To catch all these before training or prediction, one of the most common ways is **Schema Validation**. We can define the **schema as a contract of the format of our data**. Practically, the schema is a file containing all the required features for a model, their form and their type. Note that it can be in whichever format we want (many Tensorflow models use proto files). To monitor the incoming data and catch abnormalities, we can validate them against the schema. This will help us automate the data checking process.

Schema validation is especially useful when the model is deployed on a production environment and accepts user generated data. In the case of our project, once we have the UNet running in production, the user will be able to send any image he wants. As a result, we need to have a way to validate them before feeding them into the model.

Since our input data are images, which are literally 4-dimensional tensors of shape [batch, channels, height, width], an example schema will look as depicted below. Note that this is not what your typical schema will look like. Since we deal with images as inputs, we should be concise and give the correct schema as illustrated below:

```
SCHEMA = {
```

```
  "type": "object",
  "properties": {
    "image":{
        "type":"array",
        "items":{
            "type": "array",
            "items": {
                "type": "array",
                "items": {
                    "type": "array",
                    "items": {
                        "type": "number"
                    }
                }
            }
        }
    }
  },
  "required":["image"]
}
```

Essentially, our data type is a Python object as you can see in the first line. This object contains a property called `image` which is of type `array` and has a set of items. Typically, your schema will end at this point, but in our case, we need to go deep to declare all 4 dimensions of our image.

You can think of it as a type of recursion where we define the same item inside the other. Deep into the recursion, we define the type of our values to be numeric. Finally, the last line of the schema indicates all the required properties of our object. In this particular case, it's just the image.

A more common example of what a schema will look is the below:

```
SCHEMA = {
  "type": "object",
  "properties":{
    "feature-1":{
        "type":"string"
      },
    "feature-2":{
        "type":"integer"
     },
```

```
    "feature-3":{
        "type":"string"
    }
  },
  "required":["feature-1", "feature-3"]
}
```

In that case, we should expect data such as the one below:

```
{
  "feature-1": "deep-learning",
  "feature-2" : 45,
  "feature-3": "production"
}
```

I hope that clears things up. Once we have our schema, we can use it to validate our data. In Python, there is the built-in `jsonschema` package [6], which can help us do exactly that.

```
import jsonschema
from configs.data_schema import SCHEMA


class DataLoader:
  """Data Loader class"""

  @staticmethod
  def validate_schema(data_point):
    jsonschema.validate({'image':data_point.tolist()}, SCHEMA)
```

We can call the `validate_schema()` function whenever we like to check our data against our schema. How does it compare to running `print(tensor.shape)` everywhere around your production codebase? More elegant and easy, I would dare to say.

Caveat: Schema validation is an expensive and very slow operation in general, so we should think carefully where and when to enforce it because it will affect our program performance.

Advanced Tip: For those who use Tensorflow Extended (TFX) to serve their models, the data validation library can infer a schema automatically from the data. More on that on Chapter 10.

[6] Jsonschema: https://python-jsonschema.readthedocs.io/en/stable/

4.3.4 Logging

Logging goes hand in hand with debugging. Logs are records relevant to our software that are printed or stored. Logging is the act of keeping a log. But why do we need to keep logs? Logs are an essential part of troubleshooting applications and infrastructure performance. When our code is executed on a production environment in a remote machine, for instance Google Cloud, we can't really go there and start printing stuff around. Instead, in such remote environments, we use logs to have a clear image of what's going on. Logs do not exist only to capture the state of our program but also to discover possible exceptions and errors.

But why not use simple print statements? Aren't they enough? Actually, no they are not! Why? Here is an outline of some advantages logs provide over print statements:

- **We can log different severity levels** (DEBUG, INFO, WARNING, ERROR, CRITICAL) and choose to show only the level we care about. For example, we can stuff our code with debug logs, but we may not want to show all of them in production to avoid having millions of log rows. Instead, we show only warnings and errors.
- **We can choose the output channel**. This is not possible with prints as they always use the console. Some of our options are writing them to a file, sending them over http, printing them on the console, streaming them to a secondary location, or even sending them over email.
- **Timestamps are included by default**.
- The format of the message is easily **configurable**.

Tip: A general rule of thumb is to avoid print statements as much as possible and replace them with either debugging processes or logs.

And it's incredibly easy to use. Let's dive in and use it in our codebase.

4.3.5 Python's Logging module

Python's default module for logging is called `logging`. In order to use it, all we have to do is:

```
import logging

logging.warning('Warning. Our pants are on fire...")
```

But since we are developing a production-ready pipeline with highly extensible and modularized code, we should include it in a more elegant way. We can go into the `utils` folder and create a file called `logger.py` so we can import it anywhere we like.

```python
import logging.config
import yaml

with open('configs/logging_config.yaml', 'r') as f:
    config = yaml.safe_load(f.read())
    logging.config.dictConfig(config)
    logging.captureWarnings(True)

def get_logger(name: str):
    """Logs a message
    Args:
    name(str): name of logger
    """
    logger = logging.getLogger(name)
    return logger
```

The `get_logger` function will be imported when we want to log stuff and it will create a logger with a specific name. The `name` is essential so we can identify the origin of our log rows. To make the logger easily configurable, we will put all the specifications inside a config file. And since we already saw json formats, let's use a different format called `yaml`. In practice it's better to stick with a single format but here I will use a different one for educational purposes.

Our file will load the yaml file and will pass its parameters into the `logging module` to set its default behaviour.

A simple configuration file looks something like this:

```yaml
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    formatter: simple
    stream: ext://sys.stdout
Root:
  Level: DEBUG
  handlers: [console]
```

As you can see:

- We set the default format in the `formatters` node.
- We define the console as the output channel (`handler`) and streaming as the transmission method.
- We set the default level to `DEBUG`. This means that all logs above that level will be printed.

Important information: For reference, the order of levels is: DEBUG < INFO < WARNING < ERROR < CRITICAL.

So whenever we need to log something, all we have to do is import the file and use the built-in functions such as `.info()`, `.debug()` and `.error()`.

```
from utils.logger import get_logger

LOG = get_logger('unet')

def evaluate(self):
  """Predicts results for the test dataset"""

  predictions = []
  LOG.info('Predicting segmentation map for test dataset')

  for image, mask in self.test_dataset:
    LOG.debug(f'Predicting segmentation map {image}')
    predictions.append(self.model.predict(image))
  return predictions
```

A good practice is to log `info` on critical turning points such as “Data loading”, “Data pre-processed”, “Training started” and use `debug` to print data points, variables, tensor shapes. and lower-level details.

Tip: In general, most engineers log info and above levels, when the code is executed in a production environment and keep the debug level for when things break in order to debug a functionality.

Last but not least, I want to close this chapter by mentioning a few extremely useful Tensorflow functions and packages we can use to log Tensorflow-related stuff.

4.3.6 Useful Tensorflow debugging and logging functions

I feel like I should warn you that in this section, we will take a rather deep dive into Tensorflow so if you are not familiar with it or you prefer a different framework feel free to skip. But since our codebase for this book is using Tensorflow, I couldn't really avoid mentioning these.

Let's start with the definition of the computational graph because it is directly important on how logging works on Tensorflow.

A computational graph is defined as a directed graph where the nodes correspond to mathematical operations. Computational graphs are a way of expressing and evaluating a mathematical expression. Most deep learning frameworks define a computational graph each time a model is compiled. That way backpropagation can easily be executed regardless of the complexity of the model architecture. In more details, the framework applies recursively the chain rule to compute the gradients all the way to the inputs of the graph.

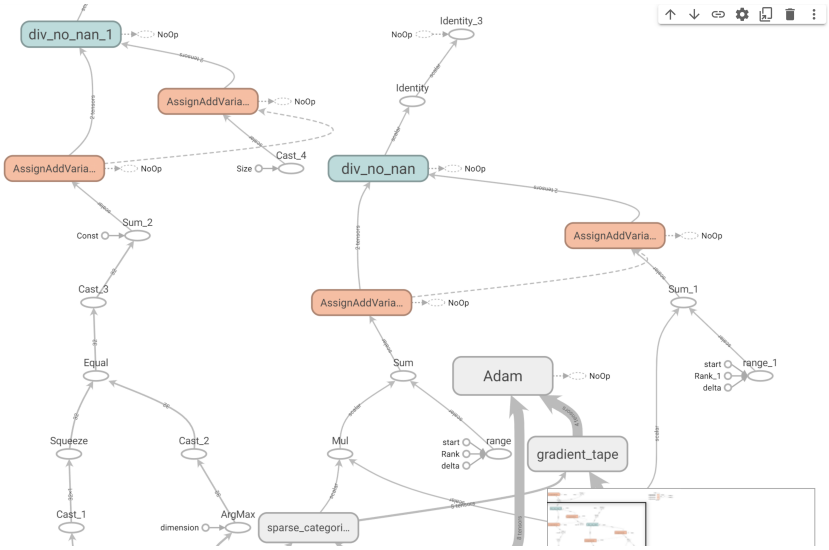

Figure 4.2: A computational graph

Tensorflow code is not your normal code and as we said before, it's not trivial to debug and test it. One of the main reasons is that Tensorflow used to have a static computational graph, meaning that you had to define the model, compile it and then run it. This made debugging much, much harder, because we couldn't access variables and states as we normally do in other applications.

However, in Tensorflow 2.0 the default execution mode is the eager (dynamic) mode, meaning that the graph is dynamic following the Pytorch pattern. Of course, there are still

cases when the code can't be executed eagerly. And even in eager mode, the computational graph still exists in the background. That's why we need these functions as they have been built with that in mind. They provide additional flexibility that normal logging simply won't.

Note: The Python debugger works only when Tensorflow is running in eager mode because the graph is compiled.

1. `tf.print` is Tensorflow's built-in print function that can be used to print tensors but also to let us define the output stream and the current level. It is based on the fact that it is actually a separate component inside the computational graph. Thus, it communicates by default with all the other components. Especially in the case that a function is not run eagerly, normal print statements won't work and we have to use `tf.print()`.

2. `tf.Variable.assign` can be used to assign values to a variable during runtime, in case you want to test things or explore different alternatives. It will directly change the computational graph so that the new value can be picked from the rest of the nodes.

3. `tf.summary` provides an API to write summary data into files. Let's say we want to save metrics on a file or a specific tensor to track its values. We can do just that with `tf.summary()`. In essence, it's a logging system to save anything we like into a file. Plus, it is integrated with Tensorboard so we can visualize our summaries with little effort.

4. `tf.debugging` is a set of assert functions (tailored to tensors) that can be put inside our code to validate our data, our weights or our model.

5. `tf.debugging.enable_metrics()` is part of the same module but I had to mention it separately because it's simply amazing. This little function will cause the code to error out as soon as an operation's output tensor contains infinity or NaN. This is really helpful in a production environment as well as during training.

6. `get_concrete_function(input).graph`. This simple but amazing function can be used to convert any python function into a `tf.Graph` so we can access all sorts of things from here (shapes, value types etc).

7. `tf.keras.callbacks` are functions that are used during training to pass information to external sources. The most common use case is passing training data into Tensorboard but that is not all. They can also be used to save csv data, early stop the training based on a metric, or even change the learning rate. It's an extremely useful tool especially for those who don't want to write Tensorflow code and prefer the simplicity of Keras.

Here is an example on how to use Tensorboard with a simple callback:

```
model.fit(
    x_train, # input
    y_train, # output
    batch_size=train_size,
    verbose=0, # Suppress chatty output; use Tensorboard instead
    epochs=100,
    validation_data=(x_test, y_test),
    callbacks=[tf.keras.callbacks.TensorBoard(log_dir=logdir)],
)
```

You will find more details about Tensorboard on the section 6.1.4.