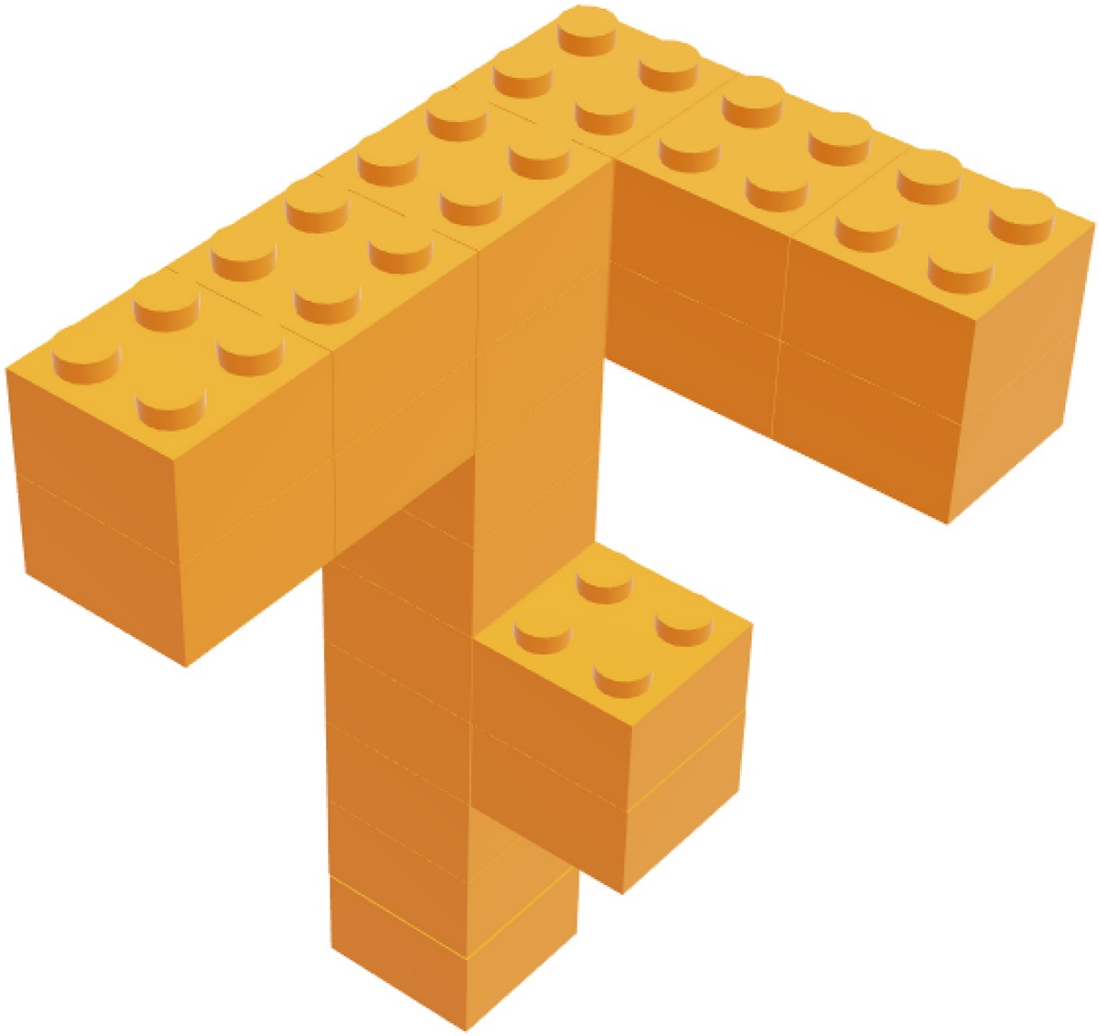
O'REALLY

Tensorflow 2 Tutorial



Ren Zhang

Table of Contents

Preface

Purpose of this writing

Tensorflow 2.0 was released in September 2019. Since the release, I have tried to follow a couple of resources trying to learn it. One problem I found during my learning is that these materials tend to focus on very high-level APIs without a detailed walkthrough on the lower-level building blocks. There is nothing wrong with quickly get hands dirty in few lines of code, but it is not enough when we try to tackle problems that are different in shapes and sizes compared to the tutorial examples. I wished there can be a slower but more of a from the ground up path to mastering the Tensorflow 2 eco-system.

The ideal target audiences for this writing are Tensorflow users with some understanding of neural networks and basic exposure to higher-level Tensorflow/Keras APIs but wanted to gain an intermediate mastering to enable more customization.

Due to my personal limit, the understanding and examples shown here can be error-prone and sometimes misleading(in a way that I did not realize), suggestions and corrections are greatly appreciated.

The book is a work in progress, the draft version is available to be viewed on legacy gitbook site or leanpub, and the source code is on github.

Chapter 1

Tensors, Operations, Variables and Automatic Differentiation

In this chapter, we will introduce the bare minimum lower level Tensorflow APIs to get started building and training models.

0. Setups for this section

```
import numpy as np
import tensorflow as tf
from pprint import pprint
print(tf.__version__)
tf.random.set_seed(42)
```

```
2.1.0
```

1. Tensors

Tensors are the objects that flow through operations(i.e., they are the inputs and the same time outputs of operations). They are N-dimensional arrays that hold elements of the same data type. The tensor objects in Tensorflow all have the shape and dtype properties, where shape is the number of elements that the tensor houses in each dimension, and dtype is the data type that all the elements within the tensor belong to.

Let's create some tensors from python and numpy objects using `tf.constant` and see their shapes and dtypes.

```
scalar = tf.constant(1, dtype=tf.int8)
vector = tf.constant([1, 2, 3], dtype=tf.float32)
matrix = tf.constant(np.array([[1, 2], [3, 4]]), dtype=tf.float32)
pprint([scalar, vector, matrix])
```

```
[<tf.Tensor: shape=(), dtype=int8, numpy=1>,
 <tf.Tensor: shape=(3,), dtype=float32, numpy=array([1., 2., 3.], d
type=float32)>,
 <tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[1., 2.],
       [3., 4.]], dtype=float32)>]
```

Under the hood, `__repr__` is using the `shape` accessor to get the shape information of the tensors. We can also use the `tf.shape` operation to get the shape of a tensor object.

```
print(matrix.shape)
pprint(tf.shape(matrix))
print(id(matrix))
```

```
(2, 2)
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([2, 2], dtype=int3
2)>
139880719227424
```

As for data type, we can't change a tensors' dtype with a mutator method as we can with numpy ndarrys(like `a = np.array([1, 2, 3]); a.dtype = np.float32)`). We can only use the `tf.cast` operation to create a new tensor with the desired new data type.

```
matrix = tf.cast(matrix, dtype=tf.int8)
pprint(matrix)
```

```
<tf.Tensor: shape=(2, 2), dtype=int8, numpy=
array([[1, 2],
       [3, 4]], dtype=int8)>
139880719226416
```

Notice the new tensor has the same values as the original tensor, but with int8 data type and it also has a different id number 4, indicating that it is a new tensor object.

There are handy ways to create special tensors in Tensorflow, let's just sample a few of them.

```
o = tf.zeros((2, 2))
x = tf.random.uniform((3, 2))
pprint(o)
pprint(x)
pprint(tf.ones_like(o))
```

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[0., 0.],
       [0., 0.]], dtype=float32)>
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[0.6645621 , 0.44100678],
       [0.3528825 , 0.46448255],
       [0.03366041, 0.68467236]], dtype=float32)>
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[1., 1.],
       [1., 1.]], dtype=float32)>
```

We saw earlier that we can convert numpy arrays to Tensors with `tf.constant`, we can do the reverse with `.numpy()` method.

```
x_numpy = x.numpy()
print(type(x_numpy))
```

```
<class 'numpy.ndarray'>
```

We can also explicitly copy tensors to devices.

```
with tf.device('/cpu:0'):
    x_cpu = tf.identity(x)
with tf.device('/gpu:0'):
    x_gpu = tf.identity(x)
print(x_cpu.device)
print(x_gpu.device)
```

```
/job:localhost/replica:0/task:0/device:CPU:0
/job:localhost/replica:0/task:0/device:GPU:0
```

Note we can actually do `x = x.cpu('0')` *or* `x = x.gpu('0')` *to achieve the same, but these two are deprecating.*

2. Operations

Operations takes in tensors and computes output tensors. Both `tf.shape` and `tf.cast` we saw earlier are actually tensor operations. Tensorflow provides a rich set of operations around tensors and these operations have routines for gradient computing built in.

Basic math operators in Python are overloaded by corresponding tensor operations. For example:

- Addition

```
e = tf.random.uniform((3, 2), dtype=x.dtype)
pprint(tf.math.reduce_all(x.__add__(e) == tf.add(x, e)))
```

```
<tf.Tensor: shape=(), dtype=bool, numpy=True>
```

- Element-wise multiplication

```
pprint(tf.math.reduce_all(x.__mul__(e) == tf.multiply(x, e)))
```

```
<tf.Tensor: shape=(), dtype=bool, numpy=True>
```

The `tf.linalg` module contains linear algebra operations. For example:

- Matrix Multiplication

```
pprint(tf.math.reduce_all(x.__matmul__(tf.transpose(e)) == tf.linal
g.matmul(x, e, transpose_b=True)))
```

```
<tf.Tensor: shape=(), dtype=bool, numpy=True>
```

TODO: work on a list of commonly used Tensorflow operations with example usecases.

Slicing and indexing are operations implemented in the `__getitem__` method of `tf.Tensor` and the behavior is similar to numpy.

```
pprint(matrix[0, 1])
pprint(matrix[1, :2])
pprint(matrix[tf.newaxis, 1, :2])
```

```
<tf.Tensor: shape=(), dtype=int8, numpy=2>
<tf.Tensor: shape=(2,), dtype=int8, numpy=array([3, 4], dtype=int8)
>
<tf.Tensor: shape=(1, 2), dtype=int8, numpy=array([[3, 4]], dtype=i
nt8)>
```

3. Variables

Tensors are immutable, the values in tensors can't be updated in-place. Variables are just like Tensors in terms of as inputs to operations, but with the in-place value update methods.

We can create variables with `tf.Variable` .

```
v = tf.Variable(x)
pprint(v)
```

```
<tf.Variable 'Variable:0' shape=(3, 2) dtype=float32, numpy=
array([[0.6645621 , 0.44100678],
       [0.3528825 , 0.46448255],
       [0.03366041, 0.68467236]], dtype=float32)>
```

Variables can be inputs to operations just as Tensors, note the output is a Tensor not Variable.

```
pprint(tf.square(v))
```

```
<tf.Tensor: id=65, shape=(3, 2), dtype=float32, numpy=
array([[0.4416428 , 0.19448698],
       [0.12452606, 0.21574403],
       [0.00113302, 0.46877623]], dtype=float32)>
```

Variables can be updated with `.assign` , `.assign_add` or `.assign_sub` methods.

```
v.assign(tf.square(v))
pprint(v)
```

```
<tf.Variable 'Variable:0' shape=(3, 2) dtype=float32, numpy=
array([[0.4416428 , 0.19448698],
       [0.12452606, 0.21574403],
       [0.00113302, 0.46877623]], dtype=float32)>
```

```
v.assign_sub(1 * tf.ones_like(v, dtype=v.dtype))
pprint(v)
```

```
<tf.Variable 'Variable:0' shape=(3, 2) dtype=float32, numpy=
array([[-0.55835724, -0.805513  ],
       [-0.8754739 , -0.784256  ],
       [-0.998867  , -0.5312238 ]], dtype=float32)>
```

Variables are typically used to represent the parameters and the states of the model, whereas the inputs, intermediate results, and outputs are Tensors.

4. Automatic Differentiation

Automatic differentiation is the implementation of the backpropagation algorithm which allows us to compute gradients of the loss function with respect to the model's parameters through chain rule. To do so, we need to keep track of the tensors and operations along the way. In Tensorflow we use the `tf.GradientTape` context to trace(recording/taping) what's happened inside, so that we can calculate the gradients afterward with the `.gradient()` from the tape object.

```
a = tf.Variable([4], dtype=tf.float32)
b = tf.Variable([5], dtype=tf.float32)

def f(a, b, power=2, d=3):
    return tf.pow(a, power) + d * b

with tf.GradientTape(watch_accessed_variables=True) as tape:
    c = f(a, b)

pprint(tape.gradient(target=c, sources=[a, b]))
```

```
[<tf.Tensor: shape=(1,), dtype=float32, numpy=array([8.], dtype=flo
at32)>,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([3.], dtype=flo
at32)>]
```

By default, the context will only keep track of the variables but not the tensors, meaning that we can only ask for gradients with respect to the variables by default (through the `watch_accessed_variables` argument which defaults to `True`). But we can explicitly ask the tape to watch things for us.

```
d = tf.constant(3, dtype=tf.float32)

with tf.GradientTape() as tape:
    tape.watch(d)
    c = f(a, b, d=d)

pprint(tape.gradient(target=c, sources=[d]))
```

```
[<tf.Tensor: shape=(), dtype=float32, numpy=5.0>]
```

Note that once we extracted the gradients from the tape, the resources it holds will be released.

```
with tf.GradientTape() as tape:
    c = f(a, b)

pprint(tape.gradient(c, [a]))
pprint(tape.gradient(c, [b]))
```

```
[<tf.Tensor: id=153, shape=(1,), dtype=float32, numpy=array([8.], d
type=float32)>]
---------------------------------------------------------------------
--------
RuntimeError                              Traceback (most recent ca
ll last)
<ipython-input-18-3b2edd8b0b33> in <module>
      3
      4 pprint(tape.gradient(c, [a]))
----> 5 pprint(tape.gradient(c, [b]))
...
RuntimeError: GradientTape.gradient can only be called once on non-
persistent tapes.
```

We can create a persistent gradient tape object to compute multiple gradients and manually release the resources.

```
with tf.GradientTape(persistent=True) as tape:
    c = f(a, b)

pprint(tape.gradient(c, [a]))
pprint(tape.gradient(c, [b]))

del tape
```

```
[<tf.Tensor: shape=(1,), dtype=float32, numpy=array([8.], dtype=flo
at32)>]
[<tf.Tensor: shape=(1,), dtype=float32, numpy=array([3.], dtype=flo
at32)>]
```

5. Linear Regression

With tensors, variables, operations and automatic differentiation, we can start building models. Let's train a linear regression model with the gradient descent algorithm.

```
# ground truth
true_weights = tf.constant(list(range(5)), dtype=tf.float32)[:, tf.new
axis]

# some random training data
x = tf.constant(tf.random.uniform((32, 5)), dtype=tf.float32)
y = tf.constant(x @ true_weights, dtype=tf.float32)

# model parameters
weights = tf.Variable(tf.random.uniform((5, 1)), dtype=tf.float32)

for iteration in range(1001):
    with tf.GradientTape() as tape:
        y_hat = tf.linalg.matmul(x, weights)
        loss = tf.reduce_mean(tf.square(y - y_hat))

    if not (iteration % 100):
        print('mean squared loss at iteration {:4d} is {:5.4f}'.format
(iteration, loss))

    gradients = tape.gradient(loss, weights)
    weights.assign_add(-0.05 * gradients)

pprint(weights)
```

```
mean squared loss at iteration    0 is 15.1603
mean squared loss at iteration  100 is 0.2243
mean squared loss at iteration  200 is 0.0586
mean squared loss at iteration  300 is 0.0161
mean squared loss at iteration  400 is 0.0046
mean squared loss at iteration  500 is 0.0013
mean squared loss at iteration  600 is 0.0004
mean squared loss at iteration  700 is 0.0001
mean squared loss at iteration  800 is 0.0000
mean squared loss at iteration  900 is 0.0000
mean squared loss at iteration 1000 is 0.0000
<tf.Variable 'Variable:0' shape=(5, 1) dtype=float32, numpy=
array([[3.1751457e-03],
       [1.0003914e+00],
       [2.0022070e+00],
       [2.9992850e+00],
       [3.9944589e+00]], dtype=float32)>
```

We will use this linear regression training example in the next few chapters and gradually replacing parts of it with higher-level Tensorflow API equivalents.

Appendix. Code for this Chapter

Chapter 2

AutoGraph

In this chapter, we will introduce how to convert python function into executable Tensorflow graphs.

0. Setups for this section

```
import inspect
import time
import numpy as np
import tensorflow as tf
from pprint import pprint
print(tf.__version__)
tf.random.set_seed(42)
np.random.seed(42)

true_weights = tf.constant(list(range(5)), dtype=tf.float32)[:, tf.new
axis]
x = tf.constant(tf.random.uniform((32, 5)), dtype=tf.float32)
y = tf.constant(x @ true_weights, dtype=tf.float32)
```

```
2.1.0
```

1. AutoGraph

A computational graph contains two things computation and data. Tensorflow graph `tf.Graph` is the computational graph made from operations as computation units and tensors as data units. Tensorflow has many optimizations around graphs, so executing in graph mode result in better utilization of distributed computing. Instead of writing

complicated graph mode code, Tensorflow 2 provides a tool *AutoGraph* to automatically analyze and convert python code into graph code which can then be traced to create a graph with *Function*.

The following is a python function we defined in chapter 1. It involves relatively simple math operations on tensors, so it is pretty much *graph ready*. Let's see if `tf.autograph` would do anything interesting to it.

```
def f(a, b, power=2, d=3):
    return tf.pow(a, power) + d * b

converted_f = tf.autograph.to_graph(f)
print(inspect.getsource(converted_f))
```

```
def tf__f(a, b, power=None, d=None):
  do_return = False
  retval_ = ag__.UndefinedReturnValue()
  with ag__.FunctionScope('f', 'fscope', ag__.ConversionOptions(rec
ursive=True, user_requested=True, optional_features=(), internal_co
nvert_user_code=True)) as fscope:
    do_return = True
    retval_ = fscope.mark_return_value(ag__.converted_call(tf.pow,
(a, power), None, fscope) + d * b)
  do_return,
  return ag__.retval(retval_)
```

The generated function is a regular python function. Even though it may look much more complicated than the original function, but most stuff is boilerplate code to handling details of function scopes and overloads function calls with `converted_call`. The line `retval_ = fscope.mark_return_value(ag__.converted_call(tf.pow, (a, power), None, fscope) + d * b)` tells us that the generated code is performing the same computation.

Let's move on to another python function with a bit graph unfriendly construct.

```
def cube(x):
    o = x
    for _ in range(2):
        o *= x
    return o

converted_cube = tf.autograph.to_graph(cube)
print(inspect.getsource(converted_cube))
```

```
def tf__cube(x):
  do_return = False
  retval_ = ag__.UndefinedReturnValue()
  with ag__.FunctionScope('cube', 'fscope', ag__.ConversionOptions(
recursive=True, user_requested=True, optional_features=(), internal
_convert_user_code=True)) as fscope:
    o = x

    def get_state():
      return ()

    def set_state(_):
      pass

    def loop_body(iterates, o):
      _ = iterates
      o *= x
      return o,
    o, = ag__.for_stmt(ag__.converted_call(range, (2,), None, fscop
e), None, loop_body, get_state, set_state, (o,), ('o',), ())
    do_return = True
    retval_ = fscope.mark_return_value(o)
  do_return,
  return ag__.retval(retval_)
```

We see that body of the `for` loop is converted into a `loop_body` function, which is invoked by `autograph.for_stmt`. This `for_stmt` operation is, in a sense, *overloads* the `for` statement. The purpose of this transformation is to make the code into a

functional style so that it can be executed in graph.

Let's look at yet another function, this time with a conditional statement.

```
def g(x):
    if tf.reduce_any(x < 0):
        return tf.square(x)
    return x
converted_g = tf.autograph.to_graph(g)
print(inspect.getsource(converted_g))
```

```
def tf__g(x):
  do_return = False
  retval_ = ag__.UndefinedReturnValue()
  with ag__.FunctionScope('g', 'fscope', ag__.ConversionOptions(rec
ursive=True, user_requested=True, optional_features=(), internal_co
nvert_user_code=True)) as fscope:

    def get_state():
      return ()

    def set_state(_):
      pass

    def if_true():
      do_return = True
      retval_ = fscope.mark_return_value(ag__.converted_call(tf.squ
are, (x,), None, fscope))
      return do_return, retval_

    def if_false():
      do_return = True
      retval_ = fscope.mark_return_value(x)
      return do_return, retval_
    cond = ag__.converted_call(tf.reduce_any, (x < 0,), None, fscop
e)
    do_return, retval_ = ag__.if_stmt(cond, if_true, if_false, get_
state, set_state, ('do_return', 'retval_'), ())
  do_return,
  return ag__.retval(retval_)
```

Here we see the similar thing happened with the conditional execution, it also has been converted into a functional form by *overload* the `if` statement.

The big idea about AutoGraph is that it translates the python code we wrote into a style that can be traced to create Tensorflow graphs. During the transformation, great efforts were made to rewrite the data-dependent conditionals and control flows, as these statements can't be straight forward overloaded by Tensorflow operations. There are a lot more about AutoGraph, the interested reader can refer to the paper for more details.

2. Functions

Once the code is graph friendly, we can trace the operations in the code to create a graph. The created graph then gets wrapped up in a `ConcreteFunction` object so that we can execute computations backed in graph mode with it. The details of how tracing works is omitted from here, due to my limited understanding of it.

Lets walkthrough how these works. First we provide Tensorflow our *graph friendly code* to create a `Function` object.

```
tf_func_f = tf.function(autograph=False)(f)
tf_func_g = tf.function(autograph=False)(converted_g)
tf_func_g2 = tf.function(autograph=True)(g)
print(tf_func_f.python_function is f)
print(tf_func_g.python_function is converted_g)
print(tf_func_g2.python_function is g)
```

```
True
True
True
```

As of now, there is no graph created yet. We only attached our function to this Function object. Notice that we specified `autograph=False` when constructing the Function in the first two cases, because we know both `f` and `converted_g` are graph friendly. But good old `g` is not, so we need to turn on `autograph` for it. In fact, `tf.function(autograph=False)(tf.autograph.to_graph(g))` is roughly equivlent to `tf.function(autograph=True)(g)` .

Note that we can create Function with `tf.function(autograph=False)(g)` this will succeed without error. But we won't be able to create any graphs with this Function in the next step.

The next step is to provide Tensorflow a signature, i.e. a *description of inputs*, allowing it to create a graph by tracing how the input tensors would flow through the operations in the graph and record the graph in a callable object.

```
concrete_g = tf_func_g.get_concrete_function(x=tf.TensorSpec(shape=[3]
, dtype=tf.float32))
print(concrete_g)
```

```
<tensorflow.python.eager.function.ConcreteFunction object at 0x7f7c
f8736240>
```

We can use this concrete function directly as if it is a operation shipped with Tensorflow, or we can call the Function object, which will look up the concrete function and use it. Either way, the computation will be executed with the created graph.

```
pprint(concrete_g(tf.constant([-1, 1, -2], dtype=tf.float32)))
pprint(tf_func_g(tf.constant([-1, 1, -2], dtype=tf.float32)))
```

```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1., 1., 4.], dt
ype=float32)>
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1., 1., 4.], dt
ype=float32)>
```

The Function object is like a graph factory. When detailed input specifications were provided, it uses the graph code as receipt to create new graphs. When asked with an known specifications, it will dig up the graph in the storage and serve it. When called with an unknown signature, it will trigger the creation of the concrete function first. Let's try to make a bunch graphs.

```
concrete_f = tf_func_f.get_concrete_function(a=tf.TensorSpec(shape=[1]
, dtype=tf.float32), b=tf.TensorSpec(shape=[1], dtype=tf.float32))
print(concrete_f)
pprint(concrete_f(tf.constant(1.), tf.constant(2.)))
pprint(tf_func_f(1., 2.))
pprint(tf_func_f(a=tf.constant(1., dtype=tf.float32), b=2, power=2.))
pprint(tf_func_f(a=tf.constant(1., dtype=tf.float32), b=2., d=3))
pprint(tf_func_f(a=tf.constant(1., dtype=tf.float32), b=2., d=3., powe
r=3.))
```

```
<tensorflow.python.eager.function.ConcreteFunction object at 0x7fb8
a40bf080>
<tf.Tensor: shape=(), dtype=float32, numpy=7.0>
<tf.Tensor: shape=(), dtype=float32, numpy=7.0>
<tf.Tensor: shape=(), dtype=float32, numpy=7.0>
<tf.Tensor: shape=(), dtype=float32, numpy=7.0>
<tf.Tensor: shape=(), dtype=float32, numpy=7.0>
```

How many graphs we created during the block of code above? The answer is 4. Can you figure out which call created which graph?

```
print(tf_func_f._get_tracing_count())
```

```
4
```

```
for i, f in enumerate(tf_func_f._list_all_concrete_functions_for_seria
lization()):
    print(i, f.structured_input_signature)
```

```
> 0 ((TensorSpec(shape=(), dtype=tf.float32, name='a'), 2.0, 3.0, 3.0)
, {})
> 1 ((TensorSpec(shape=(1,), dtype=tf.float32, name='a'), TensorSpec(s
hape=(1,), dtype=tf.float32, name='b'), 2, 3), {})
> 2 ((1.0, 2.0, 2, 3), {})
> 3 ((TensorSpec(shape=(), dtype=tf.float32, name='a'), 2, 2.0, 3), {}
)
```

We can see that with slight difference in the input signature, Tensorflow would create multiple graphs, which may ended up locking a lot of resources if not managed right.

Lastly, `tf.function` is also available as a decorator, which makes life easier. The following two are equivalent.

```
@tf.function(autograph=False)
def square(x):
    return x * x

def square(x):
    return x * x
square = tf.function(autograph=False)(square)
```

3. Linear Regression Revisited

Now let's go back to our linear regression example from last time and try to sugar it up with `tf.function`. Let's run the baseline.

```
t0 = time.time()

weights = tf.Variable(tf.random.uniform((5, 1)), dtype=tf.float32)

for iteration in range(1001):
    with tf.GradientTape() as tape:
        y_hat = tf.linalg.matmul(x, weights)
        loss = tf.reduce_mean(tf.square(y - y_hat))

    if not (iteration % 200):
        print('mean squared loss at iteration {:4d} is {:5.4f}'.format
(iteration, loss))

    gradients = tape.gradient(loss, weights)
    weights.assign_add(-0.05 * gradients)

pprint(weights)

print('time took: {} seconds'.format(time.time() - t0))
```

```
mean squared loss at iteration    0 is 16.9626
mean squared loss at iteration  200 is 0.0363
mean squared loss at iteration  400 is 0.0062
mean squared loss at iteration  600 is 0.0012
mean squared loss at iteration  800 is 0.0003
mean squared loss at iteration 1000 is 0.0001
<tf.Variable 'Variable:0' shape=(5, 1) dtype=float32, numpy=
array([[-1.9201761e-03],
       [ 1.0043812e+00],
       [ 2.0000753e+00],
       [ 3.0021193e+00],
       [ 3.9955370e+00]], dtype=float32)>
time took: 1.3443846702575684  seconds
```

Let's see if `@tf.function` can speed things up.

```
t0 = time.time()

weights = tf.Variable(tf.random.uniform((5, 1)), dtype=tf.float32)

@tf.function
def train_step():
    with tf.GradientTape() as tape:
        y_hat = tf.linalg.matmul(x, weights)
        loss = tf.reduce_mean(tf.square(y - y_hat))
    gradients = tape.gradient(loss, weights)
    weights.assign_add(-0.05 * gradients)
    return loss

for iteration in range(1001):
    loss = train_step()
    if not (iteration % 200):
        print('mean squared loss at iteration {:4d} is {:5.4f}'.format
(iteration, loss))

pprint(weights)
print('time took: {} seconds'.format(time.time() - t0))
```

```
mean squared loss at iteration    0 is 18.7201
mean squared loss at iteration  200 is 0.0325
mean squared loss at iteration  400 is 0.0017
mean squared loss at iteration  600 is 0.0002
mean squared loss at iteration  800 is 0.0000
mean squared loss at iteration 1000 is 0.0000
<tf.Variable 'Variable:0' shape=(5, 1) dtype=float32, numpy=
array([[-3.1365452e-03],
       [ 1.0065703e+00],
       [ 2.0000944e+00],
       [ 3.0032609e+00],
       [ 3.9934685e+00]], dtype=float32)>
time took: 0.4259765148162842 seconds
```

Even as simple as our example is, we can get some nice speed up with computing in graph mode.

4. Caveats

To Be Con'd

Appendix. Code for this Chapter

Chapter3

Models, Layers and Activations

In this chapter, we will introduce how to organize the components of neural network models.

0. Setups for this section

```
import time
import numpy as np
import pandas as pd
import tensorflow as tf
from functools import reduce
from matplotlib import pyplot as plt
from pprint import pprint
print(tf.__version__)
tf.random.set_seed(42)

true_weights = tf.constant(list(range(5)), dtype=tf.float32)[:, tf.new
axis]
x = tf.constant(tf.random.uniform((32, 5)), dtype=tf.float32)
y = tf.constant(x @ true_weights, dtype=tf.float32)
```

```
2.1.0
```

1. Models

A model is a set of parameters and the computation methods using these parameters. Since these two elements are tied together, we could organize them better by encapsulating the two into a class.

Recall our linear regression model. Its parameters and computation are as follows:

```
weights = tf.Variable(tf.random.uniform((5, 1)), dtype=tf.float32)
y_hat = tf.linalg.matmul(x, weights)
```

We can write a simple class to do the job.

```
class LinearRegression(object):
    def __init__(self, num_parameters):
        self._weights = tf.Variable(tf.random.uniform((num_parameters,
1)), dtype=tf.float32)

    @tf.function
    def __call__(self, x):
        return tf.linalg.matmul(x, self._weights)

    @property
    def variables(self):
        return self._weights
```

Note that we decorated the `__call__` method with `tf.function`, hence a graph will be generated to back up the computation.

With this class, we can rewrite the previous training code as:

```
model = LinearRegression(5)

@tf.function
def train_step():
    with tf.GradientTape() as tape:
        y_hat = model(x)
        loss = tf.reduce_mean(tf.square(y - y_hat))
    gradients = tape.gradient(loss, model.variables)
    model.variables.assign_add(tf.constant([-0.05], dtype=tf.float32)
* gradients)
    return loss

t0 = time.time()
for iteration in range(1001):
    loss = train_step()
    if not (iteration % 200):
        print('mean squared loss at iteration {:4d} is {:5.4f}'.format
(iteration, loss))

pprint(model.variables)
print('time took: {} seconds'.format(time.time() - t0))
```

```
mean squared loss at iteration    0 is 18.7201
mean squared loss at iteration  200 is 0.0325
mean squared loss at iteration  400 is 0.0017
mean squared loss at iteration  600 is 0.0002
mean squared loss at iteration  800 is 0.0000
mean squared loss at iteration 1000 is 0.0000
<tf.Variable 'Variable:0' shape=(5, 1) dtype=float32, numpy=
array([[-3.1365452e-03],
       [ 1.0065703e+00],
       [ 2.0000944e+00],
       [ 3.0032609e+00],
       [ 3.9934685e+00]], dtype=float32)>
time took: 0.40303897857666016 seconds
```

Here, we still decorated the `train_step` function with `tf.function`. The reason is that the loss and gradient calculation can also benefit from graphs.

This simple model class works fine. But it can be better if we subclass from `tf.keras.Model`.

```
class LinearRegression(tf.keras.Model):
    def __init__(self, num_parameters, **kwargs):
        super().__init__(**kwargs)
        self._weights = tf.Variable(tf.random.uniform((num_parameters,
1)), dtype=tf.float32)

    @tf.function
    def call(self, x):
        return tf.linalg.matmul(x, self._weights)
```

This model class has a few differences compare with the organic version. First is that we implemented the `call` method rather than the *dunder* version of it. Under the hood, `tf.keras.Model`'s `__call__` method is a wrapper over this `call` method, and it is performing, among many other things, things like converting inputs to tensors and graph building. The second thing is that we dropped the accessor for the variables because we inherited one.

With this subclass model, we need to modify the training code a bit to make it work. The `.variables` accessor from `tf.keras.Model` gives us a collection of references to the model's variables, to accommodate complex models with many sets of variables. So, as a result, the corresponding `gradients` will be a collection too.

```
model = LinearRegression(5)

@tf.function
def train_step():
    with tf.GradientTape() as tape:
        y_hat = model(x)
        loss = tf.reduce_mean(tf.square(y - y_hat))
    gradients = tape.gradient(loss, model.variables)

    for g, v in zip(gradients, model.variables):
        v.assign_add(tf.constant([-0.05], dtype=tf.float32) * g)
    return loss

t0 = time.time()
for iteration in range(1001):
    loss = train_step()
    if not (iteration % 200):
        print('mean squared loss at iteration {:4d} is {:5.4f}'.format
(iteration, loss))

pprint(model.variables)
print('time took: {} seconds'.format(time.time() - t0))
```

```
mean squared loss at iteration    0 is 18.7201
mean squared loss at iteration  200 is 0.0325
mean squared loss at iteration  400 is 0.0017
mean squared loss at iteration  600 is 0.0002
mean squared loss at iteration  800 is 0.0000
mean squared loss at iteration 1000 is 0.0000
[<tf.Variable 'Variable:0' shape=(5, 1) dtype=float32, numpy=
array([[-3.0575476e-03],
       [ 1.0064486e+00],
       [ 2.0001044e+00],
       [ 3.0031927e+00],
       [ 3.9935682e+00]], dtype=float32)>]
time took: 0.39172911643981934 seconds
```

Through subclassing `tf.keras.Model`, we get to use many methods from it, like print out a summary and the Keras model training/testing methods.

```
print(model.summary())
model.compile(loss='mse', metrics=['mae'])
print(model.evaluate(x, y, verbose=-1))
```

```
Model: "linear_regression"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Total params: 5
Trainable params: 5
Non-trainable params: 0
_________________________________________________________________
None
[4.27835811933619e-06, 0.0016658232]
```

Let's also try using the `.fit` API to train our linear regression model.

```
model = LinearRegression(5)
model.compile(optimizer='SGD', loss='mse')
model.optimizer.lr.assign(.05)

t0 = time.time()
history = model.fit(x, y, epochs=1001, verbose=0)
pprint(history.history['loss'][::200])
pprint(model.variables)
print('time took: {} seconds'.format(time.time() - t0))
```

```
[19.127595901489258,
 0.02714746817946434,
 0.0010814086999744177,
 6.190096610225737e-05,
 6.568436219822615e-06,
 1.0572820201559807e-06]
[<tf.Variable 'Variable:0' shape=(5, 1) dtype=float32, numpy=
array([[-1.2103192e-03],
       [ 1.0031627e+00],
       [ 2.0000432e+00],
       [ 3.0014870e+00],
       [ 3.9966750e+00]], dtype=float32)>]
time took: 1.2371714115142822 seconds
```

Not surprisingly we recovered the ground truth, but it took significantly more time compared to our custom training loop, it is probably due to the convenient `.fit` API is doing a bunch more stuff than updating parameters.

Now, let's spice up the model a bit by adding a additional useless bias term and initialize it with a large value. Ideally, after training, this bias term would become an insignificantly small number.

```
class LinearRegressionV2(tf.keras.Model):
    def __init__(self, num_parameters, **kwargs):
        super().__init__(**kwargs)
        self._weights = tf.Variable(tf.random.uniform((num_parameters,
1)), dtype=tf.float32)
        self._bias = tf.Variable([100], dtype=tf.float32)

    @tf.function
    def call(self, x):
        return tf.linalg.matmul(x, self._weights) + self._bias

model = LinearRegressionV2(5)

t0 = time.time()
for iteration in range(1001):
    loss = train_step()

pprint(model.variables)
```

```
[<tf.Variable 'Variable:0' shape=(5, 1) dtype=float32, numpy=
array([[0.95831835],
       [0.01680839],
       [0.3156035 ],
       [0.16013157],
       [0.7148702 ]], dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([1
00.], dtype=float32)>]
```

Hmm, something is very wrong, the bias term is not updated at all. Guess where might be the problem? It's in the `train_step` function. Since the input signature has not changed(there is none), the function is been lazy and did not realize it should do a retracing. Thus it is training with the old graph, in which the bias term simply does not exist!

To address this issue, we can make `model` as an input to `train_step`, so that when the function is invoked with a different model it will create or grab a graph accordingly.

```
@tf.function
def train_step(model):
    with tf.GradientTape() as tape:
        y_hat = model(x)
        loss = tf.reduce_mean(tf.square(y - y_hat))
    gradients = tape.gradient(loss, model.variables)

    for g, v in zip(gradients, model.variables):
        v.assign_add(tf.constant([-0.05], dtype=tf.float32) * g)
    return loss

model = LinearRegression(5)
for iteration in range(1001):
    loss = train_step(model)
pprint(model.variables)

model = LinearRegressionV2(5)
for iteration in range(5001):
    loss = train_step(model)
pprint(model.variables)

print(train_step._get_tracing_count())
```

```
[<tf.Variable 'Variable:0' shape=(5, 1) dtype=float32, numpy=
array([[-3.4687696e-03],
       [ 1.0070834e+00],
       [ 2.0001261e+00],
       [ 3.0035236e+00],
       [ 3.9930055e+00]], dtype=float32)>]
[<tf.Variable 'Variable:0' shape=(5, 1) dtype=float32, numpy=
array([[-2.6756483e-03],
       [ 9.9403673e-01],
       [ 1.9959891e+00],
       [ 2.9956400e+00],
       [ 3.9983556e+00]], dtype=float32)>,
 <tf.Variable 'Variable:0' shape=(1,) dtype=float32, numpy=array([0
.00964569], dtype=float32)>]
2
```

We see that Tensorflow traced two graph for our training function, one for each model.

2. Layers

We typically use the Model class for the overall model architecture, that is how may combine many smaller computational units to do the job. It is ok to code up small units with `tf.keras.Model` and then combine them, but since we won't really need many of the model specific functionalities with these smaller building blocks(for example, its unlikely we would ever want to call the `.fit` method on a unit). It is better to use the `tf.keras.layers.Layer` class. Actually, the Model class is a wrapper over the Layer class.

Lets say that we want to 'upgrade' the linear regression model to be a composition of a few linear transformations. We can code up the linear transformation as a Layer class and then just combine a bunch of its instances of it them in one model.

```
class Linear(tf.keras.layers.Layer):
    def __init__(self, num_inputs, num_outputs, **kwargs):
        super().__init__(**kwargs)
        self._weights = tf.Variable(tf.random.uniform((num_inputs, num
_outputs)), dtype=tf.float32)

    @tf.function
    def call(self, x):
        return tf.linalg.matmul(x, self._weights)

class Regression(tf.keras.Model):
    def __init__(self, num_inputs_per_layer, num_outputs_per_layer, **
kwargs):
        super().__init__(**kwargs)
        self._layers = [Linear(num_inputs, num_outputs)
                        for (num_inputs, num_outputs) in zip(num_input
s_per_layer, num_outputs_per_layer)]

    @tf.function
    def call(self, x):
        for layer in self._layers:
            x = layer(x)
        return x
```

In `Linear` class definition above, we swapped the super class and generalized it with the option to specify output size. In the `Regression` model class, we have the option to use one or a chain of this Linear layers. One obvious benefit with this set up, is that we now separated the concern of how individual computing units should work and the overall architecture design of the model. We use *Layer* to handle the first, and use *Model* to tackle the latter.

Let's see if the model is trainable.

```
model = Regression([5, 3], [3, 1])

for iteration in range(1001):
    loss = train_step(model)

print('Mean absolute error is: ', tf.reduce_mean(tf.abs(y - model(x)))
.numpy())
```

```
Mean absolute error is:  1.2218952e-06
```

One problem with this linear layer is that it needs the complete sizing information and allocates resources for all the variables upfront. Ideally, we want it to be a bit *lazy*, it should calculate variable sizes and occupy resources only when needed. To archive this, we implement the `build` method which will handle the variable initialization. The `build` method can be explicitly called, or it will be invoked automatically the first time there is data flow to it. With this, the constructor now only stores the hyperparameters for the layer.

```
class Linear(tf.keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super(Linear, self).__init__(**kwargs)
        self.units = units

    def build(self, input_shape):
        self._weights = self.add_weight(shape=(input_shape[-1], self.u
nits))
        super().build(input_shape)

    @tf.function
    def call(self, x):
        output = tf.linalg.matmul(x, self._weights)
        return output

class Regression(tf.keras.Model):
    def __init__(self, units, **kwargs):
        super().__init__(**kwargs)
        self._layers = [Linear(unit) for unit in units]

    @tf.function
    def call(self, x):
        for layer in self._layers:
            x = layer(x)
        return x

model = Regression([3, 1])

pprint(model.variables)  # should be empty

for iteration in range(1001):
    loss = train_step(model)

print('Mean absolute error is: ', tf.reduce_mean(tf.abs(y - model(x)))
.numpy())

pprint(model.variables)
```

```
[]
Mean absolute error is:  8.5681677e-07
[<tf.Variable 'linear_2/Variable:0' shape=(5, 3) dtype=float32, num
py=
array([[ 0.31275296, -0.34405178,  0.3254243 ],
       [-0.00516788,  0.7107771 ,  0.00381865],
       [ 1.0885493 ,  0.08547033,  0.54993755],
       [ 1.5066153 ,  0.12321236,  0.01402206],
       [ 1.1230973 ,  1.2026962 , -0.6002657 ]], dtype=float32)>,
 <tf.Variable 'linear_3/Variable:0' shape=(3, 1) dtype=float32, num
py=
array([[ 1.8777134 ],
       [ 1.4221834 ],
       [-0.30101135]], dtype=float32)>]
```

Tensorflow has a lot of layer options, we will cover a sample of them in later application specific chapters. For now, we will just quickly see if the linear layer(called `Dense`) from Tensorflow works the same.

```
class Regression(tf.keras.Model):
    def __init__(self, units, **kwargs):
        super().__init__(**kwargs)
        self._layers = [tf.keras.layers.Dense(unit, use_bias=False) for
 unit in units]  # the only change

    @tf.function
    def call(self, x):
        for layer in self._layers:
            x = layer(x)
        return x

model = Regression([3, 1])

for iteration in range(1001):
    loss = train_step(model)

print('Mean absolute error is: ', tf.reduce_mean(tf.abs(y - model(x)))
.numpy())

pprint(model.variables)
```

```
Mean absolute error is:  1.206994e-06
[<tf.Variable 'dense/kernel:0' shape=(5, 3) dtype=float32, numpy=
array([[ 0.3859551 , -0.5379335 , -0.3105871 ],
       [-0.3746996 , -0.28152603,  0.45071942],
       [ 1.0012726 , -0.08671893,  0.7469904 ],
       [ 0.2887405 , -0.875283  ,  1.0489686 ],
       [ 0.43472984, -0.5105482 ,  1.6707851 ]], dtype=float32)>,
 <tf.Variable 'dense_1/kernel:0' shape=(3, 1) dtype=float32, numpy=
array([[ 0.41473213],
       [-0.86908275],
       [ 2.020607  ]], dtype=float32)>]
```

Indeed, it is working as expected.

3. Activations

So our newest model is a composition of two linear transformation, but the composition of two linear transformations is just another linear transformation.

```
reduced_model = reduce(tf.linalg.matmul, model.variables)
print(reduced_model)
print(tf.reduce_all(tf.abs(model(x) - x @ reduced_model) < 1e-6))
```

```
tf.Tensor(
[[2.2053719e-06]
 [9.9999624e-01]
 [1.9999999e+00]
 [2.9999964e+00]
 [4.0000048e+00]], shape=(5, 1), dtype=float32)
tf.Tensor(True, shape=(), dtype=bool)
```

Without anything interesting in between, this is just adding unnecessary complexity. The simplest thing to do is to add some non-linear in-place transformation to the intermediate results, and this is call activations.

Let's add activations as layers.

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

    @tf.function
    def call(self, x):
        return tf.maximum(tf.constant(0, x.dtype), x)


class NeuralNetwork(tf.keras.Model):
    def __init__(self, units, last_linear=True, **kwargs):
        super().__init__(**kwargs)
        layers = []
        n = len(units)
```

```
        for i, unit in enumerate(units):
            layers.append(Linear(unit))
            if i < n - 1 or not last_linear:
                layers.append(ReLU())
        self._layers = layers

    @tf.function
    def call(self, x):
        for layer in self._layers:
            x = layer(x)
        return x

model = NeuralNetwork([3, 1])

for iteration in range(1001):
    loss = train_step(model)

print('Mean absolute error is: ', tf.reduce_mean(tf.abs(y - model(x)))
.numpy())

pprint(model.variables)

reduced_model = reduce(tf.linalg.matmul, model.variables)
print(reduced_model)
print(tf.reduce_any(tf.abs(model(x) - x @ reduced_model) < 1e-6))
```

```
Mean absolute error is:  0.005225517
[<tf.Variable 'linear_4/Variable:0' shape=(5, 3) dtype=float32, num
py=
array([[ 0.10789682, -0.31562465, -0.00532613],
       [-0.05060532, -0.46457097,  0.42605096],
       [-0.61450577,  0.45144582,  0.8922052 ],
       [ 0.439215  , -0.59760475,  1.2671946 ],
       [ 0.36582235,  0.14377609,  1.7026857 ]], dtype=float32)>,
 <tf.Variable 'linear_5/Variable:0' shape=(3, 1) dtype=float32, num
py=
array([[ 0.13811348],
       [-0.3345939 ],
       [ 2.3246977 ]], dtype=float32)>]
tf.Tensor(
[[0.10812643]
 [1.138893  ]
 [1.8381848 ]
 [3.206461  ]
 [3.960648  ]], shape=(5, 1), dtype=float32)
tf.Tensor(False, shape=(), dtype=bool)
```

With our underline ground truth to be a simple linear model, the added non-linear activations is not really helpful, and in fact it made the optimization harder.

Since activation functions usually follows immediately after linear transformations, we can fuse them together, so that the model code can be simpler.

```
class Linear(tf.keras.layers.Layer):
    def __init__(self, units, use_bias=True, activation='linear', **kw
args):
        super(Linear, self).__init__(**kwargs)
        self.units = units
        self.use_bias = use_bias
        self.activation = activation

    def build(self, input_shape):
        self._weights = self.add_weight(shape=(input_shape[-1], self.u
nits))
```

```
        if self.use_bias:
            self._bias = self.add_weight(shape=(self.units), initializ
er='ones')
        super().build(input_shape)

    @tf.function
    def call(self, x):
        output = tf.linalg.matmul(x, self._weights)
        if self.use_bias:
            output += self._bias
        if self.activation == 'relu':
            output = tf.maximum(tf.constant(0, x.dtype), output)
        return output


class NeuralNetwork(tf.keras.Model):
    def __init__(self, units, use_bias=True, last_linear=True, **kwarg
s):
        super().__init__(**kwargs)
        layers = [Linear(unit, use_bias, 'relu') for unit in units[:-1
]]
        layers.append(Linear(units[-1], use_bias, 'linear' if last_lin
ear else 'relu'))
        self._layers = layers

    @tf.function
    def call(self, x):
        for layer in self._layers:
            x = layer(x)
        return x

model = NeuralNetwork([3, 1])

for iteration in range(1001):
    loss = train_step(model)

print('Mean absolute error is: ', tf.reduce_mean(tf.abs(y - model(x)))
.numpy())

pprint(model.variables)
```

```
Mean absolute error is:  0.0075564235
[<tf.Variable 'linear_6/Variable:0' shape=(5, 3) dtype=float32, num
py=
array([[ 0.13394988,  0.51800126,  0.09352156],
       [-0.15037392,  0.25468782,  0.49990052],
       [-0.69800115, -0.42260593,  0.8586105 ],
       [-0.08989831, -0.33655766,  1.2849052 ],
       [ 1.0911363 , -0.32361698,  1.6646731 ]], dtype=float32)>,
 <tf.Variable 'linear_6/Variable:0' shape=(3,) dtype=float32, numpy
=array([ 0.5917124,  0.5500173, -0.576148 ], > dtype=float32)>,
 <tf.Variable 'linear_7/Variable:0' shape=(3, 1) dtype=float32, num
py=
array([[ 0.14004707],
       [-0.45114964],
       [ 2.2325916 ]], dtype=float32)>,
 <tf.Variable 'linear_7/Variable:0' shape=(1,) dtype=float32, numpy
=array([1.4508461], dtype=float32)>]
```

There are also many `activation` functions we can choose from that ship with Tensorflow. We will do a survey on them later.

4. Fully Connected Networks

With the code above, we just made a fully connected network, or historically called multi layer perceptron(with out any actual perceptron) as well as feed forward neural network. Its essentially a sequence of linear transformation with in-place non-linear activations sandwiched in between. We usually think of the initial layers as feature extractors that is performing some kind on implicit feature engineering and selection, and think of the last layer as a regressor or classifier per task.

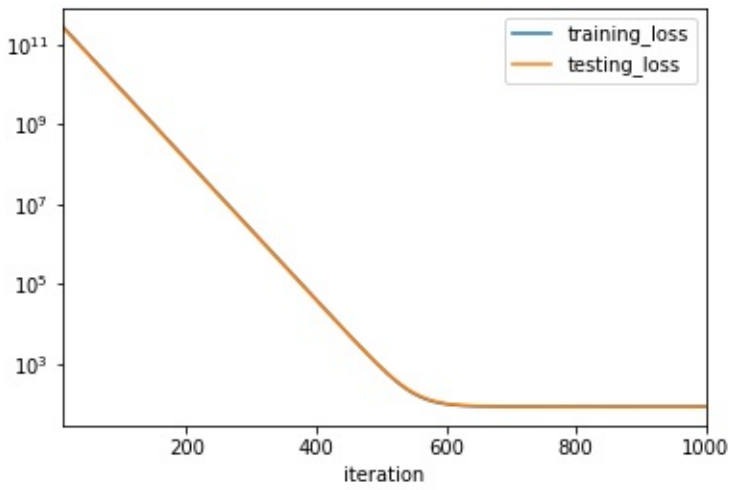
Note how we are using a list to host the layers and applying them sequentially in the call method. Lets quickly implement a quality of life improvement model class called `Sequential` to do this. It is pretty much a water down version of `tf.keras.Sequential`.

```
class Sequential(tf.keras.Model):
    def __init__(self, layers, **kwargs):
        super().__init__(**kwargs)
        self._layers = layers

    @tf.function
    def call(self, x):
        for layer in self._layers:
            x = layer(x)
        return x


class MLP(tf.keras.Model):
    def __init__(self, num_hidden_units, num_targets, hidden_activatio
n='relu', **kwargs):
        super().__init__(**kwargs)
        if type(num_hidden_units) is int: num_hidden_units = [num_hidd
en_units]
        self.feature_extractor = Sequential([tf.keras.layers.Dense(uni
t, activation=hidden_activation)
                                             for unit in num_hidden_un
its])
        self.last_linear = tf.keras.layers.Dense(num_targets, activati
on='linear')

    @tf.function
    def call(self, x):
        features = self.feature_extractor(x)
        outputs = self.last_linear(features)
        return outputs
```

Let's try to apply our MLP model to a real regression problems: the boston housing dataset shipped with Tensorflow. The dataset is splitted into two sets, a training set and a testing set. We will train the model on training set only, but record the loss on both sets to see if the reduction in training set loss is inline with reduction in the unseen testing set.

```
(x_tr, y_tr), (x_te, y_te) = tf.keras.datasets.boston_housing.load_dat
a()
y_tr, y_te = map(lambda x: np.expand_dims(x, -1), (y_tr, y_te))
x_tr, y_tr, x_te, y_te = map(lambda x: tf.cast(x, tf.float32), (x_tr,
y_tr, x_te, y_te))

@tf.function
def train_step(model, x, y):
    with tf.GradientTape() as tape:
        loss = tf.reduce_mean(tf.square(y - model(x)))
    gradients = tape.gradient(loss, model.variables)
    for g, v in zip(gradients, model.variables):
        v.assign_add(tf.constant([-0.01], dtype=tf.float32) * g)
    return loss

@tf.function
def test_step(model, x, y):
    return tf.reduce_mean(tf.square(y - model(x)))

def train(model, n_epochs=1000, his_freq=10):
    history = []
    for iteration in range(1, n_epochs + 1):
        tr_loss = train_step(model, x_tr, y_tr)
        te_loss = test_step(model, x_te, y_te)
        if not iteration % his_freq:
            history.append({
                'iteration': iteration,
                'training_loss': tr_loss.numpy(),
                'testing_loss': te_loss.numpy()
            })
    return model, pd.DataFrame(history)

mlp, mlp_history = train(MLP(4, 1))
pprint(mlp_history.tail())
ax = mlp_history.plot(x='iteration', kind='line', logy=True)
fig = ax.get_figure()
fig.savefig('ch3_plot_1.png')
```

```
95         960      84.622253     83.714134
96         970      84.622231     83.713562
97         980      84.622299     83.713020
98         990      84.622231     83.712601
99        1000      84.622269     83.712318
```



It may seem that our model has nicely converged. Since there is not much a discrepancy between training set and testing set performance. However if we look at the numbers more closely, they are pretty bad. A simple constant prediction have MSE around 83. What could go wrong? We will look at other optimizers in the next chapter.

```
print(tf.reduce_mean(tf.square(y_te - tf.reduce_mean(y_te))))
```

```
tf.Tensor(83.24384, shape=(), dtype=float32)
```

Appendix. Code for this Chapter

Chapter4

Optimizers

In this chapter, we will introduce how to implement different optimizers.

0. Setups for this section

```
import numpy as np
import pandas as pd
import tensorflow as tf
from pprint import pprint
print(tf.__version__)
tf.random.set_seed(42)
(x_tr, y_tr), (x_te, y_te) = tf.keras.datasets.boston_housing.load_dat
a()
y_tr, y_te = map(lambda x: np.expand_dims(x, -1), (y_tr, y_te))
x_tr, y_tr, x_te, y_te = map(lambda x: tf.cast(x, tf.float32), (x_tr,
y_tr, x_te, y_te))


class MLP(tf.keras.Model):
    def __init__(self, num_hidden_units, num_targets, hidden_activatio
n='relu', **kwargs):
        super().__init__(**kwargs)
        if type(num_hidden_units) is int: num_hidden_units = [num_hidd
en_units]
        self.feature_extractor = tf.keras.Sequential([tf.keras.layers.
Dense(unit, activation=hidden_activation)
                                                      for unit in num_
hidden_units])
        self.last_linear = tf.keras.layers.Dense(num_targets, activati
on='linear')

    @tf.function
    def call(self, x):
        features = self.feature_extractor(x)
        outputs = self.last_linear(features)
        return outputs
```

```
2.1.0
```

1. Gradient Descent

Lets formally look at the gradient descent algorithm we have been using so far.

1. $g \leftarrow \nabla_\theta J(\theta)$
2. $u \leftarrow -\eta g$
3. $\theta \leftarrow \theta + u$

Where:

- $g \leftarrow \nabla_\theta J(\theta)$ is the gradient of loss function J with respect to our model's parameters θ.
- u is the updates we will apply to each parameters, in this case it is the negative gradient multiplied by the learning rate η.
- Lastly, the update rule is to increment θ by u.

Take a look at our previous optimization code:

```
gradients = tape.gradient(loss, model.variables)

for g, v in zip(gradients, model.variables):
    v.assign_add(tf.constant([-0.05], dtype=tf.float32) * g)
```

With only a slight tweak, the code will become literally a line by line translation of the formula.

```
eta = tf.constant(-.05, dtype=tf.float32)
gradients = tape.gradient(loss, model.variables)
for grad, var in zip(gradients, model.variables):
    update = - eta * grad
    var.assign_add(update)
```

Its obvious that our optimization contains some data(the learning rate η) and some operations(the parameter updates). We can put them into a class.

```
class GradientDescent(object):
    def __init__(self, lr=.01):
        self._lr = tf.Variable(lr, dtype=tf.float32)

    def apply_gradients(self, grads_and_vars):
        for grad, var in grads_and_vars:
            update = - self._lr * grad
            var.assign_add(update)
```

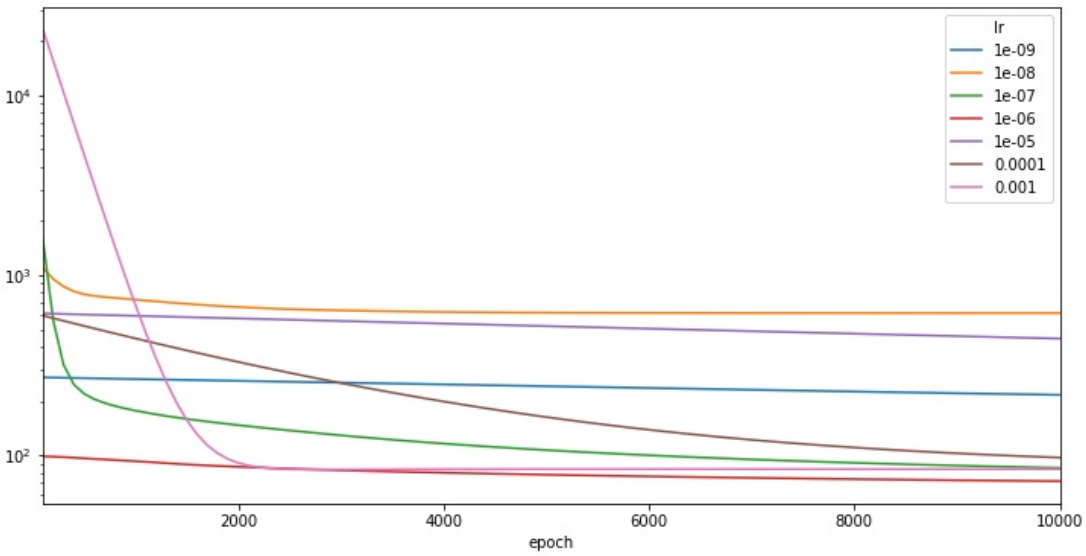
With this optimizer class, our training code is becoming more modularized.

```
@tf.function
def train_step(model, optimizer, x, y):
    with tf.GradientTape() as tape:
        loss = tf.reduce_mean(tf.square(y - model(x)))
    gradients = tape.gradient(loss, model.variables)
    optimizer.apply_gradients(zip(gradients, model.variables))
    return loss


@tf.function
def test_step(model, x, y):
    return tf.reduce_mean(tf.square(y - model(x)))


def train(model, optimizer, n_epochs=10000, his_freq=100):
    history = []
    for epoch in range(1, n_epochs + 1):
        tr_loss = train_step(model, optimizer, x_tr, y_tr)
        te_loss = test_step(model, x_te, y_te)
        if not epoch % his_freq:
            history.append({'epoch': epoch,
                            'training_loss': tr_loss.numpy(),
                            'testing_loss': te_loss.numpy()})
    return model, pd.DataFrame(history)
```

Recall that from the end of last chapter, our model is no better than a constant function. Wondering if learning rate of the optimizer has anything to do with it. Let's to code up some experiments on how learning rate affects the training progress.

```
def lr_experiments(learning_rates):
    experiments = []
    for lr in learning_rates:
        model, history = train(MLP(4, 1), GradientDescent(lr))
        history['lr'] = lr
        experiments.append(history)

    experiments = pd.concat(experiments, axis=0)
    return experiments

experiments = lr_experiments(learning_rates=[10 ** -i for i in range(3
, 10)])
ax = experiments.\
    pivot(index='epoch', columns='lr', values='testing_loss').\
    plot(kind='line', logy=True, figsize=(12, 6))

ax.get_figure().savefig('ch4_plot_1.png')
print(experiments.groupby('lr')['testing_loss'].min())
```

```
lr
1.000000e-09    216.570267
1.000000e-08    616.280029
1.000000e-07     85.039314
1.000000e-06     71.763756
1.000000e-05    444.852142
1.000000e-04     97.061386
1.000000e-03     83.243866
Name: testing_loss, dtype: float64
```



We see that with learning rate 1e-6, we can finally beat the naive baseline model, which is a constant. Wait a second, in what situation would our network be equivalent to a constant? Only if the activations are mostly zeros, and thus the model would almost always output the bias term from the last layer. Let's see if that is the case.

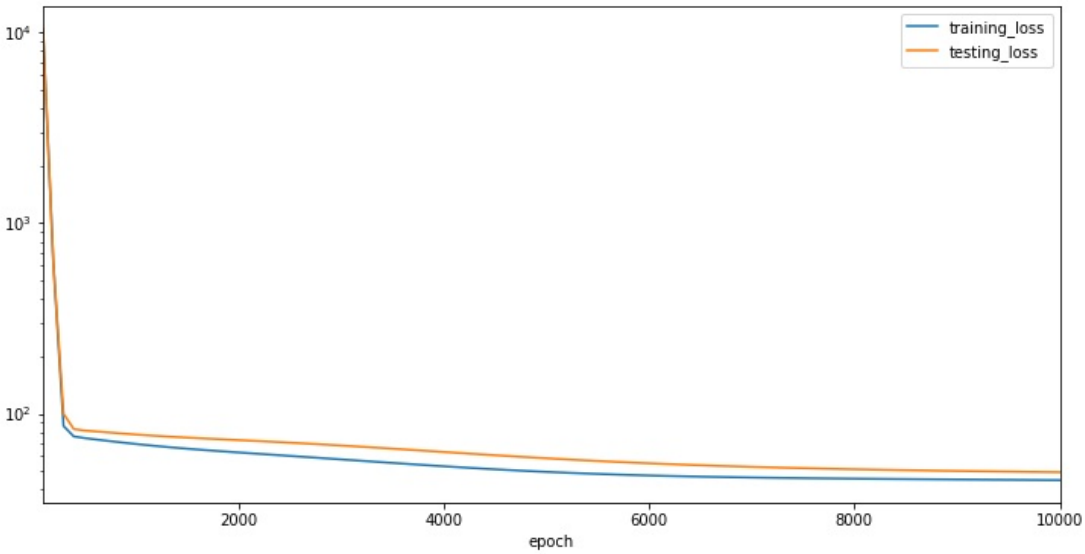
```
model, history = train(MLP(4, 1), GradientDescent(1e-3))
print(model.layers[0](x_te))
print(model.layers[0](x_te) @ model.layers[1].variables[0])
print(model.layers[1].variables[1])
```

```
 tf.Tensor(
 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  ...
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]], shape=(102, 4), dtype=float32)
 tf.Tensor(
 [[0.]
  [0.]
  ...
  [0.]
  [0.]], shape=(102, 1), dtype=float32)
 <tf.Variable 'dense_15/bias:0' shape=(1,) dtype=float32, numpy=arra
 y([22.394573], dtype=float32)>
```

Indeed, with learning rate 1e-3 our network basically collapsed to a bias term. In this case, the initial gradients exploded, if the learning rate is also very big, it can kill almost all the units immediately. So, we would need to either reduce the learning rate or apply some kind of control of the gradient values. Normalize the gradient is a good addition we should add to the gradient descent optimizer.

```
class GradientDescent(object):
    def __init__(self, lr=.01, clipnorm=None):
        self._lr = tf.Variable(lr, dtype=tf.float32)
        self.clipnorm = clipnorm

    def apply_gradients(self, grads_and_vars):
        for grad, var in grads_and_vars:
            if self.clipnorm: grad = tf.clip_by_norm(grad, self.clipno
rm)
            update = - self._lr * grad
            var.assign_add(update)

model, history = train(MLP(4, 1), GradientDescent(1e-3, clipnorm=2))
ax = history.plot(x='epoch', logy=True, figsize=(12, 6))
ax.get_figure().savefig('ch4_plot_2.png')
print(history['testing_loss'].min())
```

```
49.352027893066406
```



Now, we tried training our model again, this time, we normalize the gradients to have l2 norm equal to a small number 2, and indeed we got much better result even with the previously problematic learning rate value 1e-3.

2. Stochastic Gradient Descent

Yet another problem with the above gradient descent optimization is that it uses the full dataset every time it needs to do a parameter update. It can be very both very slow and memory hungry. A quick fix is to use small samples of the data to get estimates of the gradients and perform much more frequent updates. This is called stochastic gradient descent(SGD).

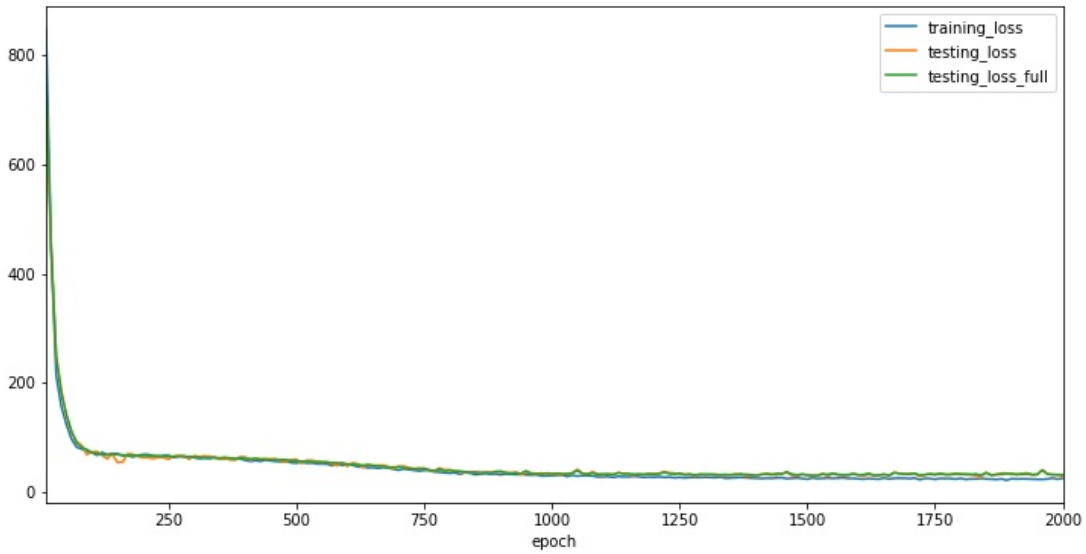
To train our model with SGD, we don't need to modify our optimizer class. Instead, we need to modify the training code to include sampling. To make life a bit easier, we can wrap the data tensors into iterators.

```
class Dataset(object):
    def __init__(self, tensors, batch_size=32, shuffle=True):
        self.tensors = tensors if isinstance(tensors, (list, tuple)) e
lse (tensors, )
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.total = tensors[0].shape[0]
        assert all(self.total == tensor.shape[0] for tensor in self.te
nsors), 'Tensors should have matched length'
        self.n_steps = self.total // self.batch_size
        self._indices = tf.range(self.total)

    def __iter__(self):
        self._i = 0
        if self.shuffle:
            self._indices = tf.random.shuffle(self._indices)
        return self

    def __next__(self):
        if self._i >= self.n_steps:
            raise StopIteration
        else:
            start = self._i * self.batch_size
            end = start + self.batch_size
            indices = self._indices[start: end]
            samples = (tf.gather(tensor, indices) for tensor in self.t
ensors)
            self._i += 1
            return samples
```

With this `Dataset` class, we can make very small change to the training code.

```
train_dataset = Dataset((x_tr, y_tr))
test_dataset = Dataset((x_te, y_te))


def train(model, optimizer, n_epochs, batch_size=32, his_freq=10):
    history = []
    for epoch in range(1, n_epochs + 1):
        tr_loss = []
        for x, y in train_dataset:
            tr_loss.append(train_step(model, optimizer, x, y).numpy())

        te_loss = []
        for x, y in test_dataset:
            te_loss.append(test_step(model, x, y).numpy())

        te_loss_full = test_step(model, x_te, y_te)

        if not epoch % his_freq:
            history.append({'epoch': epoch,
                            'training_loss': np.mean(tr_loss),
                            'testing_loss': np.mean(te_loss),
                            'testing_loss_full': te_loss_full.numpy()}
)
    return model, pd.DataFrame(history)

model, history = train(MLP(4, 1), GradientDescent(1e-3, 2), n_epochs=2
000)
ax = history.plot(x='epoch', kind='line', figsize=(12, 6))
ax.get_figure().savefig('ch4_plot_3.png')

print(history.testing_loss_full.min())
```

```
30.624069213867188
```



We can see that training with small batches of random samples, the trajectory of loss values becomes a bit zig-zag in shape, but still follows the similar path as previously when we train with full dataset.

3 Momentum

If we look at the learning curve, we see that after the initial huge drop, the progress becomes very slow but steady. Wondering if there is something can help us accelerate the progress? One improvement made to Gradient Descent is to accelerate it by builds up the velocity, which equals to use an exponential moving average(EMA) version of the gradients. In formula it looks like:

1. $g \leftarrow \nabla_\theta J(\theta)$
2. $u \leftarrow \beta u - \eta g$
3. $\theta \leftarrow \theta + u$

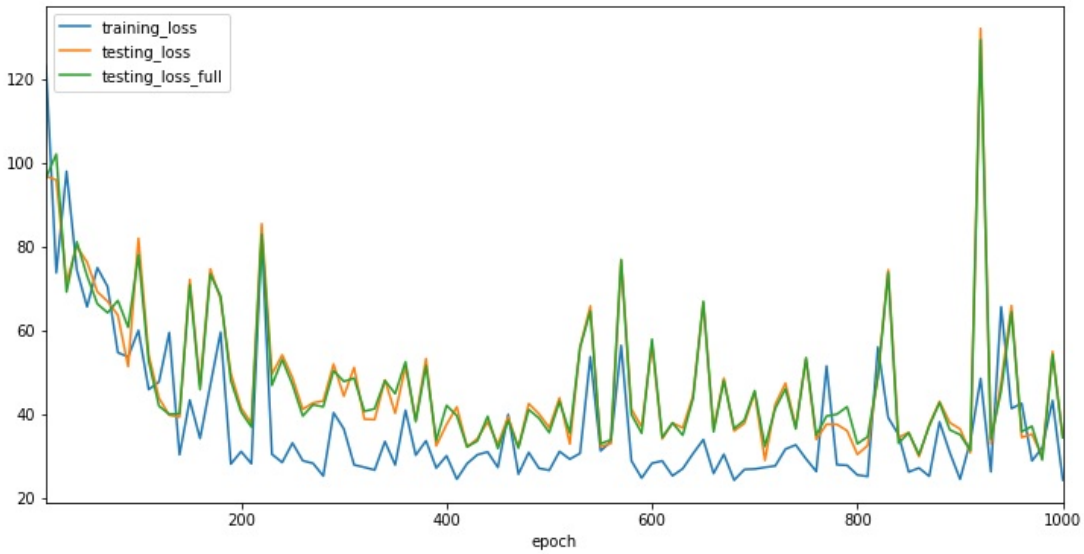
It is not too difficult to upgrade our gradient descent optimizer to add this feature.

```
class Momentum(object):
    def __init__(self, lr=.01, beta=.9, clipnorm=None):
        self._lr = tf.Variable(lr, dtype=tf.float32)
        self._beta = tf.Variable(beta, dtype=tf.float32)
        self.clipnorm = clipnorm

    def init_moments(self, var_list):
        self._moments = {var._unique_id: tf.Variable(tf.zeros_like(var
))
                         for var in var_list}

    def apply_gradients(self, grads_and_vars):
        for grad, var in grads_and_vars:
            if self.clipnorm: grad = tf.clip_by_norm(grad, self.clipno
rm)
            m = self._moments[var._unique_id]
            m.assign(self._beta * m - self._lr * grad)
            update = m
            var.assign_add(m)
```

What added a set of moment variables to accumulates gradients, and we used a mapping to keep track of the association between variables and the accumulators.

```
model = MLP(4, 1)
model.build(input_shape=(32,13))
optimizer = Momentum(lr=1e-3, beta=.98, clipnorm=2)
optimizer.init_moments(model.variables)

model, history = train(model, optimizer, n_epochs=1000)
ax = history.plot(x='epoch', kind='line', figsize=(12, 6))
ax.get_figure().savefig('ch4_plot_4.png')

print(history.testing_loss_full.min())
```

```
29.115631103515625
```



4. Second Moment

As with the first moment of the gradients, second moment were also made use to guide optimization. The Adam optimizer looks like this in formula

1. $g \leftarrow \nabla_\theta J(\theta)$
2. $m \leftarrow \beta_1 m + (1-\beta_1) g$
3. $v \leftarrow \beta_2 v + (1-\beta_2) g^2$
4. $\hat{\eta} \leftarrow \eta \frac{\sqrt{1-\beta_2^T}}{1-\beta_1^T}$
5. $u \leftarrow -\hat{\eta} \frac{m}{\sqrt{v}+\epsilon}$
6. $\theta \leftarrow \theta + u$

In a nutshell, the optimizer uses two sets of accumulators to keep track of the first two moments of the gradients. The algorithm uses the second moment to scale the first moment, intuitively this works like a signal to noise ratio adjustment, with the first moment be the signal and the second moment be noise. Since all the operations are elementwise, this signal to noise treatment is customized for each and every parameter in the model, so effectively every parameter has its own learning rate.

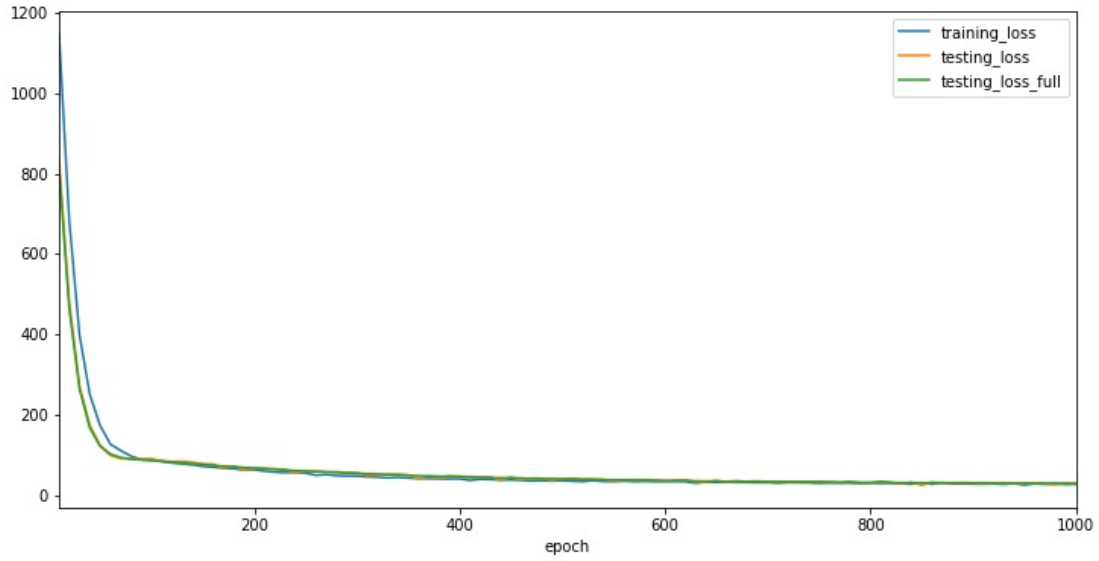
With a few copy, paste and edit, we can upgrade the gradient descent with momentum optimizer to Adam easily.

```
class Adam(object):
    def __init__(self, lr=.01, beta_1=.9, beta_2=.999, epsilon=1e-8, c
lipnorm=None):
        self._lr = tf.Variable(lr, dtype=tf.float32)
        self._beta_1 = tf.Variable(beta_1, dtype=tf.float32)
        self._beta_2 = tf.Variable(beta_2, dtype=tf.float32)
        self._epsilon = tf.constant(epsilon, dtype=tf.float32)
        self.clipnorm = clipnorm
        self._t = tf.Variable(0, dtype=tf.float32)

    def init_moments(self, var_list):
        self._m = {var._unique_id: tf.Variable(tf.zeros_like(var))
                   for var in var_list}
        self._v = {var._unique_id: tf.Variable(tf.zeros_like(var))
                   for var in var_list}

    def apply_gradients(self, grads_and_vars):
        self._t.assign_add(tf.constant(1., self._t.dtype))
        for grad, var in grads_and_vars:
            if self.clipnorm: grad = tf.clip_by_norm(grad, self.clipno
rm)

            m = self._m[var._unique_id]
            v = self._v[var._unique_id]

            m.assign(self._beta_1 * m + (1. - self._beta_1) * grad)
            v.assign(self._beta_2 * v + (1. - self._beta_2) * tf.squar
e(grad))

            lr = self._lr * tf.sqrt(1 - tf.pow(self._beta_2, self._t))
 / (1 - tf.pow(self._beta_1, self._t))
            update = -lr * m / (tf.sqrt(v) + self._epsilon)
            var.assign_add(update)
```

Let's see if we can get better result with Adam.

```
model = MLP(4, 1)
model.build(input_shape=(32,13))
optimizer = Adam(lr=1e-3, beta_1=.9, beta_2=.999, epsilon=1e-8, clipno
rm=2)
optimizer.init_moments(model.variables)

model, history = train(model, optimizer, n_epochs=1000)
ax = history.plot(x='epoch', kind='line', figsize=(12, 6))
ax.get_figure().savefig('ch4_plot_5.png')

print(history.testing_loss_full.min())
```

```
28.03963851928711
```



Adam is pretty much the default first choice of optimizer for most people on most problems. Since we have already organically coded it up, we can now take a look at how we can go about to do it by sub classing the `tf.keras.optimizers.Optimizer`.

To implement a custom `tf.keras` optimizer, there are a few methods we need to implement:

1. `_resource_apply_dense()` - this is the method used to perform parameter updates with dense gradient tensors.

2. `_resource_apply_sparse()` - above but works with sparse gradient tensors.
3. `_create_slots()` - optionally if the optimizer require more variables. If the optimizer only uses gradients and variables(like SGD), this is not needed.
4. `_get_config()` - optionally for save(serialize) / load(de-serialize) the optimizer with hyperparameters.

our `init_moments()` method roughly corresponds to the `_create_slots()` and our `apply_gradients()` method needs to be moved to `_resource_apply_dense()`. And since we have quite a few hyperparameters, we need to code up the `_get_config()` method too.

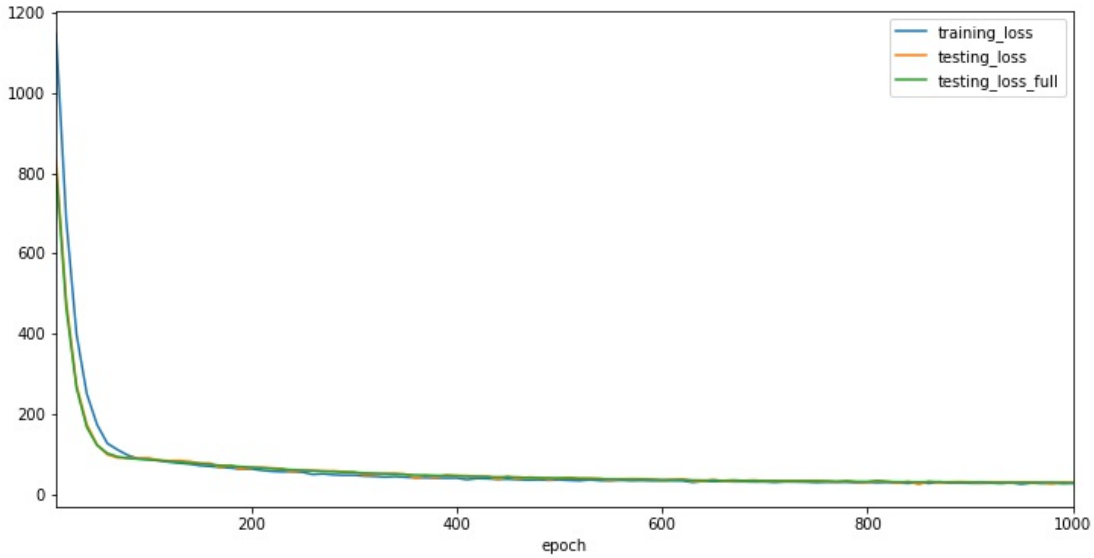
```
class Adam(tf.keras.optimizers.Optimizer):
    def __init__(self, learning_rate=.001, beta_1=.9, beta_2=.999, eps
ilon=1e-8, name='Adam', **kwargs):
        super().__init__(name, **kwargs)
        self._set_hyper('learning_rate', kwargs.get('lr', learning_rat
e))
        self._set_hyper('beta_1', beta_1)
        self._set_hyper('beta_2', beta_2)
        self.epsilon = epsilon or tf.keras.backend.epsilon()

    def _create_slots(self, var_list):
        for var in var_list:
            self.add_slot(var, 'm')
        for var in var_list:
            self.add_slot(var, 'v')

    def _resource_apply_dense(self, grad, var):
        dtype = var.dtype.base_dtype
        t = tf.cast(self.iterations + 1, dtype)
        lr = self._decayed_lr(dtype)
        beta_1 = self._get_hyper('beta_1', dtype)
        beta_2 = self._get_hyper('beta_2', dtype)
        epsilon = tf.convert_to_tensor(self.epsilon, dtype)

        m = self.get_slot(var, 'm')
        v = self.get_slot(var, 'v')

        m = m.assign(beta_1 * m + (1. - beta_1) * grad)
```

```
        v = v.assign(beta_2 * v + (1. - beta_2) * tf.square(grad))

        lr = lr * tf.sqrt(1 - tf.pow(beta_2, t)) / (1 - tf.pow(beta_1,
 t))
        update = -lr * m / (tf.sqrt(v) + epsilon)
        var_update = var.assign_add(update)
        updates = [var_update, m, v]
        return tf.group(*updates)

    def get_config(self):
        config = super().get_config()
        config.update({
            'learning_rate': self._serialize_hyperparameter('learning_
rate'),
            'beta_1': self._serialize_hyperparameter('beta_1'),
            'beta_2': self._serialize_hyperparameter('beta_2'),
            'epsilon': self.epsilon,
            'total_steps': self._serialize_hyperparameter('total_steps'
),
        })
        return config
```

Time to see how does it compares.

```
model = MLP(4, 1)
optimizer = Adam(lr=1e-3, beta_1=.9, beta_2=.999, epsilon=1e-8)

model, history = train(model, optimizer, n_epochs=1000)
ax = history.plot(x='epoch', kind='line', figsize=(12, 6))
ax.get_figure().savefig('ch4_plot_5.png')

print(history.testing_loss_full.min())
```

```
29.335590362548828
```



From now on, we will just use `tf.keras.optimizers.Adam` whenever we are going to train a model.

So we spent a lot time in optimizers in this chapter, along the way we looked at the effect of learning rate, normalization of the gradient values to prevent network collapse, and looked into the black box of how a few classic optimizer works. And we see all these has a positive impact on the training result, as our network is fitted much better compare to at the end of last chapter. Next we will look into Loss functions.