In a machine learning job interview, you may be asked to implement a linear classifier from scratch in TensorFlow: a very simple task that serves as a filter between candidates who have some minimal machine learning background, and those who don’t. Let’s get you past that filter, and use your newfound knowledge of TensorFlow to implement such a linear classifier.

Because our linear classifier operates on 2D inputs, W is really just two scalar coefficients, w1 and w2: W = [[w1], [w2]]. Meanwhile, b is a single scalar coefficient. As such, for given input point [x, y], its prediction value is: prediction = [[w1], [w2]] • [x, y] + b = w1 \* x + w2 \* y + b.

##### Listing 3.13. Generating two classes of random points in a 2D plane

1

2

3

4

5

num\_samples\_per\_class = 1000

negative\_samples = np.random.multivariate\_normal(

mean=[0, 3], cov=[[1, 0.5],[0.5, 1]], size=num\_samples\_per\_class)

positive\_samples = np.random.multivariate\_normal(

mean=[3, 0], cov=[[1, 0.5],[0.5, 1]], size=num\_samples\_per\_class)

1

2

[copy](javascript:void(0))

negative\_samples and positive\_samples are both arrays with shape (1000, 2). Let’s stack them into a single array with shape (2000, 2):

##### Listing 3.14. Stacking the two classes into an array with shape (2000, 2)

1

inputs = np.vstack((negative\_samples, positive\_samples)).astype(np.float32)

[copy](javascript:void(0))

Let’s generate the corresponding target labels, an array of zeros and ones of shape (2000, 1), where targets[i,

0] is 0 if inputs[i] belongs to class 0 (and inversely):

##### Listing 3.15. Generating the corresponding targets (0 and 1)

1

2

targets = np.vstack((np.zeros((num\_samples\_per\_class, 1), dtype='float32'),

np.ones((num\_samples\_per\_class, 1), dtype='float32')))

[copy](javascript:void(0))

Let’s plot our data with Matplotlib, a well-known Python data visualization library (it comes preinstalled in Colab, so no need for you to install it yourself):

##### Listing 3.16. Plotting the two point classes

1

2

3

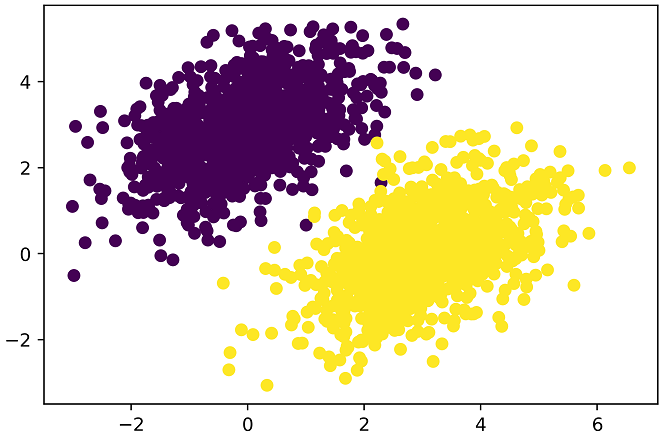
import matplotlib.pyplot as plt

plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])

plt.show()

[copy](javascript:void(0))

##### Figure 3.6. Our synthetic data: two classes of random points in the 2D plane



Now, let’s create a linear classifier that can learn to separate these two blobs. A linear classifier is an affine transformation (prediction = W • input + b) trained to minimize the square of the difference between predictions and the targets.

Ba dxq’ff kav, rj’c yclultaa s zmup mlesirp paexeml ndrz rux noq-kr-vnu lxpaeme vl z kur kwr-reyal enural kwonetr emlt rgv pnk kl tehcpar 2. Hovweer, bjrc mkjr, yvg dhsoul xg cfgk vr deutdnrans eyghvtiren obaut yxr vzvq, jfnv dd fjno.

Vxr’c eraect etg asvraeilb W nhs b, dniaielztii rjuw oamrdn sauvle hzn jrwg sroez preeseyitcvl:

##### Listing 3.17. Creating the linear classifier variables

1

2

3

4

input\_dim = 2

output\_dim = 1

W = tf.Variable(initial\_value=tf.random.uniform(shape=(input\_dim, output\_dim)))

b = tf.Variable(initial\_value=tf.zeros(shape=(output\_dim,)))

1

2

[copy](javascript:void(0))

Here’s our forward pass function:

##### Listing 3.18. The forward pass function

1

2

def model(inputs):

return tf.matmul(inputs, W) + b

[copy](javascript:void(0))

## 3.6  Anatomy of a neural network: understanding core Keras APIs

Here’s our loss function:

##### Listing 3.19. The mean squared error loss function

1

2

3

def square\_loss(targets, predictions):

per\_sample\_losses = tf.square(targets - predictions)

return tf.reduce\_mean(per\_sample\_losses)

1

2

[copy](javascript:void(0))

Here’s our loss function:

##### Listing 3.19. The mean squared error loss function

1

2

3

def square\_loss(targets, predictions):

per\_sample\_losses = tf.square(targets - predictions)

return tf.reduce\_mean(per\_sample\_losses)

1

2

[copy](javascript:void(0))

Dxw, dro itaninrg ahrx, cihwh veesecri kmkz igartnin cyrc ngs spadtue vru itgwhse W npz b ea cc vr imzmneii vbr xaaf nx vpr srpz:

##### Listing 3.20. The training step function

1

2

3

4

5

6

7

8

9

10

learning\_rate = 0.1

def training\_step(inputs, targets):

with tf.GradientTape() as tape:

predictions = model(inputs)

loss = square\_loss(predictions, targets)

grad\_loss\_wrt\_W, grad\_loss\_wrt\_b = tape.gradient(loss, [W, b])

W.assign\_sub(grad\_loss\_wrt\_W \* learning\_rate)

b.assign\_sub(grad\_loss\_wrt\_b \* learning\_rate)

return loss

1

2

3

[copy](javascript:void(0))

You can think of layers as the LEGO bricks of deep learning, a metaphor that is made explicit by Keras. Building deep-learning models in Keras is done by clipping together compatible layers to form useful data-transformation pipelines.

##### Listing 3.21. The batch training loop

1

2

3

for step in range(20):

loss = training\_step(inputs, targets)

print('Loss at step %d: %.4f' % (step, loss))

[copy](javascript:void(0))

A simple API should have a single abstraction around which everything is centered. In Keras, that’s the Layer class. Everything in Keras is either a Layer or something that closely interacts with a Layer.

1

2

3

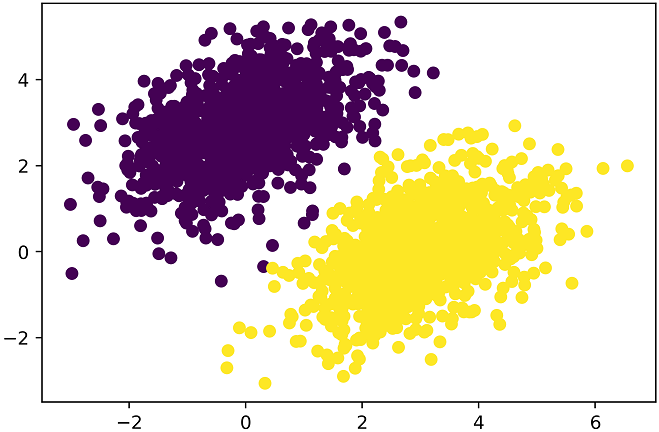
predictions = model(inputs)

plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)

plt.show()

[copy](javascript:void(0))

##### Figure 3.7. Our model’s predictions on the training inputs: pretty similar to the training targets



Recall that the prediction value for a given point [x, y] is simply prediction == [[w1], [w2]] • [x, y] + b == w1 \* x + w2 \* y + b. Thus, class "0" is defined as: w1 \* x + w2 \* y + b < 0.5 and class "1" is defined as: w1 \* x + w2 \* y + b > 0.5. You’ll notice that what you’re looking at is really the equation of a line in the 2D plane: w1 \* x + w2 \* y + b = 0.5. Above the line, class 1, below the line, class 0. You may be used to seeing line equations in the format y = a \* x + b; in the same format, our line becomes: y = - w1 / w2 \* x + (0.5 - b) / w2.

Let’s plot this line:

1

2

3

4

x = np.linspace(-1, 4, 100)

y = - W[0] / W[1] \* x + (0.5 - b) / W[1]

plt.plot(x, y, '-r')

plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)

1

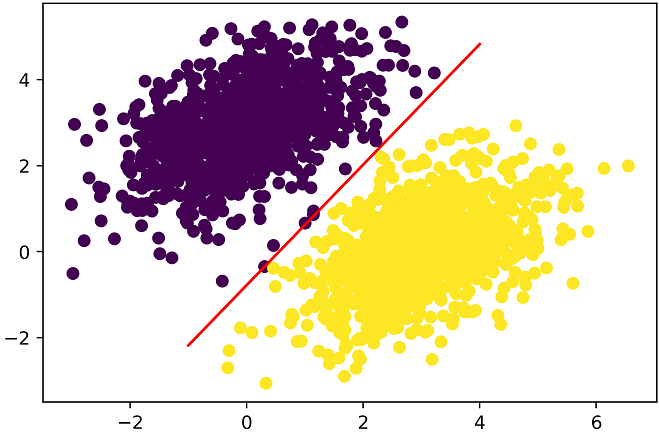
2

3

4

[copy](javascript:void(0))

##### Figure 3.8. Our model, visualized as a line



Now, you’re probably wondering, why did we had to implement call() and build(), since we ended up using our layer by plainly calling it, that is to say, by using its call method? It’s because we want to be able to create the state just in time. Let’s see how that works.

## 3.6  Anatomy of a neural network: understanding core Keras APIs

Just like with LEGO bricks, you can only “clip” together layers that are compatible. The notion of layer compatibility here refers specifically to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape. Consider the following example:

### 3.6.1  Layers: the building blocks of deep learning

This layer will return a tensor where the first dimension has been transformed to be 32. It can only be connected to a downstream layer that expects 32-dimensional vectors as its input.

When using Keras, you don’t have to worry about size compatibility most of the time, because the layers you add to your models are dynamically built to match the shape of the incoming layer. For instance, suppose you write the following:

You can think of layers as the LEGO bricks of deep learning, a metaphor that is made explicit by Keras. Building deep-learning models in Keras is done by clipping together compatible layers to form useful data-transformation pipelines.

#### The base Layer class in Keras

In the toy version of a Dense layer that we’ve implemented in chapter 2 (which we named NaiveDense), we had to pass the layer’s input size explicitly to the constructor in order to be able to create its weights. That’s not ideal, because it would lead to models that looks like this, where each new layer needs to be made aware of the shape of the layer before it:

A Layer is an object that encapsulates some state (weights) and some computation (a forward pass). The weights are typically defined in a build() (although they could also be created in the constructor init()), and the computation is defined in the call() method.

It would be even worse when the rules used by a layer to produce its output shape are complex. For instance, what if our layer returned outputs of shape (batch, input\_size \* 2 if input\_size % 2 == 0 else input\_size \* 3)?

from tensorflow import keras

class SimpleDense(keras.layers.Layer):

def \_\_init\_\_(self, units, activation=None):

super(SimpleDense, self).\_\_init\_\_()

self.units = units

self.activation = activation

def build(self, input\_shape):

input\_dim = input\_shape[-1]

self.W = self.add\_weight(shape=(input\_dim, self.units),

initializer='random\_normal')

self.b = self.add\_weight(shape=(self.units,),

initializer='zeros')

def call(self, inputs):

y = tf.matmul(inputs, self.W) + self.b

if self.activation is not None:

y = self.activation(y)

return y

1

2

3

4

[copy](javascript:void(0))

In SimpleDense, we no longer create weights in the constructor like in the NaiveDense example, instead, we create them in a dedicated state-creation method build(), which receives as argument the first input shape seen by the layer. The build() method is called automatically the first time the layer is called (via its call method). In fact, that’s why we defined the computation in a call method rather than in call! The call method of the base layer schematically looks like this:

Once instantiated, a layer like this can be used just like a function, taking as input a TensorFlow tensor:

>>> my\_dense = SimpleDense(units=32, activation=tf.nn.relu)

>>> input\_tensor = tf.ones(shape=(2, 784))

>>> output\_tensor = my\_dense(input\_tensor)

>>> print(output\_tensor.shape)

(2, 32)

Now, you’re probably wondering, why did we had to implement call() and build(), since we ended up using our layer by plainly calling it, that is to say, by using its call method? It’s because we want to be able to create the state just in time. Let’s see how that works.

#### Automatic shape inference: building layers on the fly

### 3.6.2  From layers to models

1

2

from tensorflow.keras import layers

layer = layers.Dense(32, activation='relu')

– missing 2 lines –

from tensorflow.keras import models

from tensorflow.keras import layers

model = models.Sequential([

layers.Dense(32, activation='relu'),

layers.Dense(32)

])

[copy](javascript:void(0))

The topology of a model defines a hypothesis space. You may remember that in chapter 1, we described machine learning as “searching for useful representations of some input data, within a predefined space of possibilities, using guidance from a feedback signal.” By choosing a network topology, you constrain your space of possibilities (hypothesis space) to a specific series of tensor operations, mapping input data to output data. What you’ll then be searching for is a good set of values for the weight tensors involved in these tensor operations.

To learn from data, you have to make assumptions about it. These assumptions define what can be learned. As such, the structure of your hypothesis space — the architecture of your model — is extremely important. It encodes the assumptions you make about your problem, the prior knowledge that the model starts with. For instance, if you’re working on a two-class classification problem with a model made of a single Dense layer with no activation (a pure affine transformation), you are assuming that your two classes are linearly separable.

model = NaiveSequential([

NaiveDense(input\_size=784, output\_size=32, activation='relu'),

NaiveDense(input\_size=32, output\_size=64, activation='relu'),

NaiveDense(input\_size=64, output\_size=32, activation='relu'),

NaiveDense(input\_size=32, output\_size=10, activation='softmax')

])

[copy](javascript:void(0))

### 3.6.3  The "compile" step: configuring the learning process

Once the model architecture is defined, you still have to choose three more things:

Once you’ve picked your loss, optimizer, and metrics, you can use the built-in compile() and fit() methods to start training your model. Alternatively, you could also write your own custom training loops — we cover how to do this in chapter 6. It’s a lot more work! For now, let’s take a look at compile() and fit().

1

2

3

4

5

def \_\_call\_\_(self, inputs):

if not self.built:

self.build(inputs.shape)

self.built = True

return self.call(inputs)

[copy](javascript:void(0))

With automatic shape inference, our previous example becomes simple and neat:

1

2

3

4

5

6

model = keras.Sequential([

SimpleDense(32, activation='relu'),

SimpleDense(64, activation='relu'),

SimpleDense(32, activation='relu'),

SimpleDense(10, activation='softmax')

])

[copy](javascript:void(0))

Note that automatic shape inference is not the only thing that the Layer class' call method handles. It takes care of many more things, in particular routing between eager and graph execution (a concept you’ll learn about in chapter 6), and input masking (which we cover in chapter TODO). For now, just remember: when implementing your own layers, put the forward pass in the call method.

### 3.6.2  From layers to models

A deep-learning model is a graph of layers. In Keras, that’s the Model class. For now, you’ve only seen Sequential models (a subclass of Model), which are simple stack of layers, mapping a single input to a single output. But as you move forward, you’ll be exposed to a much broader variety of network topologies. Some common ones are:

* In chapter 6, we cover how to create custom losses and metrics. In general, you won’t have to create your own losses, metrics, or optimizers from scratch, because Keras offers a wide range of built-in options that is likely to include what you need:

### 3.6.3  The "compile" step: configuring the learning process

Once the model architecture is defined, you still have to choose three more things:

* The call to fit returns a History object. This object contains a history field which is a dict mapping keys such as "loss" or specific metric names to the list of their per-epoch values.
* Optimizer  — Determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD).

--- missing small para + 2 lines --



model = keras.Sequential([keras.layers.Dense(1)])

model.compile(optimizer='rmsprop',

loss='mean\_squared\_error',

metrics=['accuracy'])

1

2

3

4

[copy](javascript:void(0))

The value of the loss on the validation data is called the “validation loss”, to distinguish it from the “training loss”. Note that it’s essential to keep the training data and validation data strictly separate: the purpose of validation is to monitor whether what the model is learning is actually useful on new data. If any of the validation data has been seen by the model during training, your validation loss and metrics will be flawed.

1

2

3

model.compile(optimizer=keras.optimizers.RMSprop(),

loss=keras.losses.MeanSquaredError(),

metrics=[keras.metrics.BinaryAccuracy()])

[copy](javascript:void(0))

oss\_and\_metrics = model.evaluate(val\_inputs, val\_targets, batch\_size=128)

1

2

3

model.compile(optimizer=keras.optimizers.RMSprop(learning\_rate=1e-4),

loss=my\_custom\_loss,

metrics=[my\_custom\_metric\_1, my\_custom\_metric\_2])

[copy](javascript:void(0))

### 3.6.7  Inference: using a model after training

Optimizers:

* SGD() (with or without momentum)
* However, this will process all inputs in new\_inputs at once, which may not be feasible if you’re looking at a lot of data (in particular, it may require more memory than your GPU has).
* A better way to do inference is to use the predict() method. It will iterate over the data in small batches, and return a NumPy array of predictions. And unlike call, it can also process TensorFlow Dataset objects.
* Adagrad()
* For instance, if we use predict() on some of our validation data with the linear model we trained earlier, we get scalar scores between 0 and 1 — below 0.5 indicates that the model considers the corresponding point to belong to class 0, and above 0.5 indicates that the model considers the corresponding point to belong to class 1.

Losses:

* CategoricalCrossentropy()
* SparseCategoricalCrossentropy()
* BinaryCrossentropy()
* Keras is the standard API to do deep learning on top of TensorFlow. It’s what we’ll use throughout this book.
* Key TensorFlow objects include tensors, variables, tensor operations, and the gradient tape.
* The central class of Keras is the Layer. A layer encapsulate some weights and some computation. Layers are assembled into models.
* Before you start training a model, you need to pick an optimizer, a loss, and some metrics, which you specify via the model.compile() method.

Metrics:

* Once your model is trained, use the model.predict() method to generate predictions on new inputs.
* SparseCategoricalAccuracy()
* BinaryAccuracy()
* AUC()
* Precision()
* Recall()

### 3.6.4  Picking a loss function

Choosing the right loss function for the right problem is extremely important: your network will take any shortcut it can to minimize the loss; so if the objective doesn’t fully correlate with success for the task at hand, your network will end up doing things you may not have wanted. Imagine a stupid, omnipotent AI trained via SGD, with this poorly chosen objective function: “maximizing the average well-being of all humans alive.” To make its job easier, this AI might choose to kill all humans except a few and focus on the well-being of the remaining ones — because average well-being isn’t affected by how many humans are left. That might not be what you intended! Just remember that all neural networks you build will be just as ruthless in lowering their loss function — so choose the objective wisely, or you’ll have to face unintended side effects.

Fortunately, when it comes to common problems such as classification, regression, and sequence prediction, there are simple guidelines you can follow to choose the correct loss. For instance, you’ll use binary crossentropy for a two-class classification problem, categorical crossentropy for a many-class classification problem, and so on. Only when you’re working on truly new research problems will you have to develop your own objective functions. In the next few chapters, we’ll detail explicitly which loss functions to choose for a wide range of common tasks.

### 3.6.5  Understanding the "fit" method

After compile() comes fit(). The fit method implements the training loop itself. Its key arguments are:

* The data (inputs and targets) to train on. It will typically be passed either in the form of NumPy arrays, of a TensorFlow Dataset object. You’ll learn more about the Dataset API in the next chapters.
* The number of epochs to train for: how many times the training loop should iterate over the data passed.
* The batch size to use within each epoch of mini-batch gradient descent: the number of training examples considered to compute the gradients for one weight update step.

##### Listing 3.22. Calling fit with NumPy data

1

2

3

4

5

6

history = model.fit(

inputs,

targets,

epochs=5,

batch\_size=128

)

1

2

3

4

– –

>>> history.history

{'binary\_accuracy': [0.855, 0.9565, 0.9555, 0.95, 0.951],

'loss': [0.6573270302042366,

0.07434618508815766,

0.07687718723714351,

0.07412414988875389,

0.07617757616937161]}

[copy](javascript:void(0))

### 3.6.6  Monitoring loss & metrics on validation data

The goal of machine learning is not to obtain models that perform well on the training data — which is easy, all you have to do is follow the gradient. The goal is to obtain models that perform well in general, in particular on data points that the model has never encountered before. Just because a model performs well on its training data doesn’t mean it will perform well on data it has never seen! For instance, it’s possible that your model could end up merely memorizing a mapping between your training samples and their targets, which would be useless for the task of predicting targets for data the model has never seen before. We’ll go over this point in much more detail in the chapter 5.

To keep an eye on how the model does on new data, it’s standard practice to reserve a subset of the training data as “validation data”: you won’t be training the model on this data, but you will use it to compute a loss value and metrics value. You do this use the validation\_data argument in fit(). Like the training data, the validation data could be passed as NumPy arrays or as a TensorFlow Dataset object.

##### Listing 3.23. Using the validation data argument

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

model = keras.Sequential([keras.layers.Dense(1)])

model.compile(optimizer=keras.optimizers.RMSprop(learning\_rate=0.1),

loss=keras.losses.MeanSquaredError(),

metrics=[keras.metrics.BinaryAccuracy()])

indices\_permutation = np.random.permutation(len(inputs))

shuffled\_inputs = inputs[indices\_permutation]

shuffled\_targets = targets[indices\_permutation]

num\_validation\_samples = int(0.3 \* len(inputs))

val\_inputs = shuffled\_inputs[-num\_validation\_samples:]

val\_targets = shuffled\_targets[-num\_validation\_samples:]

training\_inputs = shuffled\_inputs[:num\_validation\_samples]

training\_targets = shuffled\_targets[:num\_validation\_samples]

model.fit(

training\_inputs,

training\_targets,

epochs=5,

batch\_size=16,

validation\_data=(val\_inputs, val\_targets)

)

1

2

3

4

[copy](javascript:void(0))

The value of the loss on the validation data is called the “validation loss”, to distinguish it from the “training loss”. Note that it’s essential to keep the training data and validation data strictly separate: the purpose of validation is to monitor whether what the model is learning is actually useful on new data. If any of the validation data has been seen by the model during training, your validation loss and metrics will be flawed.

Note that if you want to compute the validation loss and metrics after training is complete, you can call the evaluate method:

loss\_and\_metrics = model.evaluate(val\_inputs, val\_targets, batch\_size=128)

evaluate() will iterate in batches (of size batch\_size) over the data passed, and return a list of scalars, where the first entry is the validation loss and the following entries are the validation metrics. If the model has no metrics, only the validation loss is returned (rather than a list).

Once you’ve trained your model, you’re going to want to use it to make predictions on new data. This is called "inference". To do this, a naive approach would simply be to call the model:

1

predictions = model(new\_inputs)

1

[copy](javascript:void(0))

However, this will process all inputs in new\_inputs at once, which may not be feasible if you’re looking at a lot of data (in particular, it may require more memory than your GPU has).

A better way to do inference is to use the predict() method. It will iterate over the data in small batches, and return a NumPy array of predictions. And unlike call, it can also process TensorFlow Dataset objects.

1

predictions = model.predict(new\_inputs, batch\_size=128)

1

[copy](javascript:void(0))

For instance, if we use predict() on some of our validation data with the linear model we trained earlier, we get scalar scores between 0 and 1 — below 0.5 indicates that the model considers the corresponding point to belong to class 0, and above 0.5 indicates that the model considers the corresponding point to belong to class 1.

>>> predictions = model.predict(val\_inputs, batch\_size=128)

>>> print(predictions[:10])

[[0.3590725 ]

[0.82706255]

[0.74428225]

[0.682058 ]

[0.7312616 ]

[0.6059811 ]

[0.78046083]

[0.025846 ]

[0.16594526]

[0.72068727]]

[copy](javascript:void(0))

For now, this is all your need to know about Keras models. At this point, you are ready to move on to solving real-world machine problems with Keras, in the next chapter.