Introduction to Keras and TensorFlow

All-ones or all-zeros tensors

In [ ]:
```
import tensorflow as tf
x = tf.ones(shape=(2, 1))
print(x)
```

```
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)
```

In [ ]:
```
x = tf.zeros(shape=(2, 1))
print(x)
```

```
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)
```

Random tensors

In [ ]:
```
x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)
print(x)
```

```
tf.Tensor(
[[0.8309784 ]
 [0.02322833]
 [0.20062953]], shape=(3, 1), dtype=float32)
```

In [ ]:
```
x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
print(x)
```

```
tf.Tensor(
[[0.68336225]
 [0.27214646]
 [0.9351466 ]], shape=(3, 1), dtype=float32)
```

NumPy arrays are assignable

In [ ]:
```
import numpy as np
x = np.ones(shape=(2, 2))
x[0, 0] = 0.
```

Creating a TensorFlow variable

In [ ]:
```
v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1))
```

```
print(v)
```

```
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[-0.35847458],
       [ 0.3193509 ],
       [ 0.41017598]], dtype=float32)>
```

Assigning a value to a TensorFlow variable

In [ ]:
```
v.assign(tf.ones((3, 1)))
```

Out[ ]:
```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, nu
mpy=
array([[1.],
       [1.],
       [1.]], dtype=float32)>
```

Assigning a value to a subset of a TensorFlow variable

In [ ]:
```
v[0, 0].assign(3.)
```

Out[ ]:
```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, nu
mpy=
array([[3.],
       [1.],
       [1.]], dtype=float32)>
```

Using `assign_add`

In [ ]:
```
v.assign_add(tf.ones((3, 1)))
```

Out[ ]:
```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, nu
mpy=
array([[4.],
       [2.],
       [2.]], dtype=float32)>
```

Tensor operations: Doing math in TensorFlow

A few basic math operations

In [ ]:
```
a = tf.ones((2, 2))
b = tf.square(a)
c = tf.sqrt(a)
d = b + c
e = tf.matmul(a, b)
e *= d
```

A second look at the GradientTape API

Using the `GradientTape`

In [ ]:
```
input_var = tf.Variable(initial_value=3.)
with tf.GradientTape() as tape:
   result = tf.square(input_var)
gradient = tape.gradient(result, input_var)
```

Using `GradientTape` with constant tensor inputs

In [ ]:
```
input_const = tf.constant(3.)
with tf.GradientTape() as tape:
   tape.watch(input_const)
   result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

Using nested gradient tapes to compute second-order gradients

In [ ]:
```
time = tf.Variable(0.)
with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as inner_tape:
        position =  4.9 * time ** 2
    speed = inner_tape.gradient(position, time)
acceleration = outer_tape.gradient(speed, time)
```

An end-to-end example: A linear classifier in pure TensorFlow

Generating two classes of random points in a 2D plane

In [ ]:
```
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)
```

Stacking the two classes into an array with shape (2000, 2)

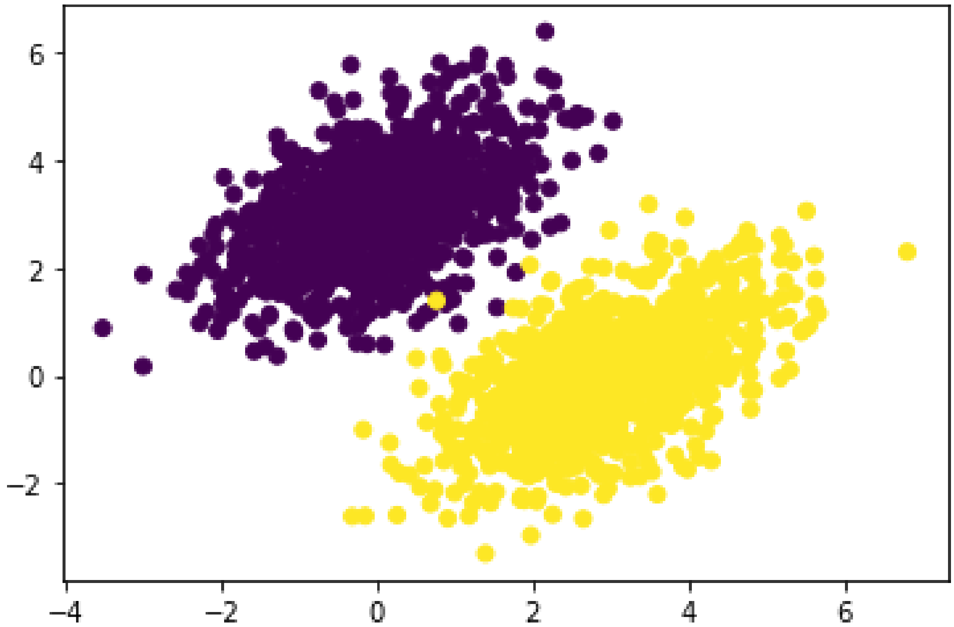
In [ ]:
```
inputs = np.vstack((negative_samples, positive_samples)).ast
```

Generating the corresponding targets (0 and 1)

In [ ]:
```
targets = np.vstack((np.zeros((num_samples_per_class, 1), dt
                     np.ones((num_samples_per_class, 1), dty
```

Plotting the two point classes

In [ ]:
```
import matplotlib.pyplot as plt
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])
plt.show()
```



Creating the linear classifier variables

In [ ]:
```
input_dim = 2
output_dim = 1
W = tf.Variable(initial_value=tf.random.uniform(shape=(input
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
```

The forward pass function

In [ ]:
```
def model(inputs):
    return tf.matmul(inputs, W) + b
```

The mean squared error loss function

In [ ]:
```
def square_loss(targets, predictions):
    per_sample_losses = tf.square(targets - predictions)
    return tf.reduce_mean(per_sample_losses)
```

The training step function

In [ ]:
```
learning_rate = 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
```

```
        loss = square_loss(targets, predictions)
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [
    W.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b * learning_rate)
    return loss
```

The batch training loop

In [ ]:

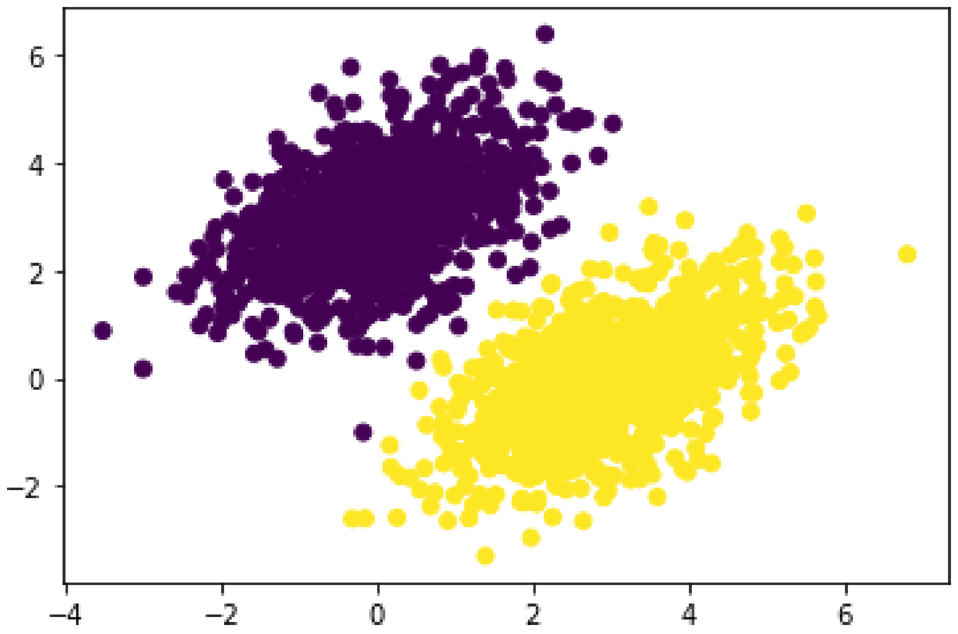
```
for step in range(40):
    loss = training_step(inputs, targets)
    print(f"Loss at step {step}: {loss:.4f}")
```

```
Loss at step 0: 1.8413
Loss at step 1: 0.3466
Loss at step 2: 0.1495
Loss at step 3: 0.1168
Loss at step 4: 0.1059
Loss at step 5: 0.0983
Loss at step 6: 0.0917
Loss at step 7: 0.0858
Loss at step 8: 0.0804
Loss at step 9: 0.0754
Loss at step 10: 0.0709
Loss at step 11: 0.0668
Loss at step 12: 0.0631
Loss at step 13: 0.0597
Loss at step 14: 0.0566
Loss at step 15: 0.0538
Loss at step 16: 0.0512
Loss at step 17: 0.0489
Loss at step 18: 0.0467
Loss at step 19: 0.0448
Loss at step 20: 0.0430
Loss at step 21: 0.0414
Loss at step 22: 0.0399
Loss at step 23: 0.0386
Loss at step 24: 0.0373
Loss at step 25: 0.0362
Loss at step 26: 0.0352
Loss at step 27: 0.0343
Loss at step 28: 0.0334
Loss at step 29: 0.0327
Loss at step 30: 0.0320
Loss at step 31: 0.0313
Loss at step 32: 0.0307
Loss at step 33: 0.0302
Loss at step 34: 0.0297
Loss at step 35: 0.0293
Loss at step 36: 0.0289
Loss at step 37: 0.0285
```

```
Loss at step 38: 0.0282
Loss at step 39: 0.0279
```

In [ ]:

```
predictions = model(inputs)
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0]
plt.show()
```

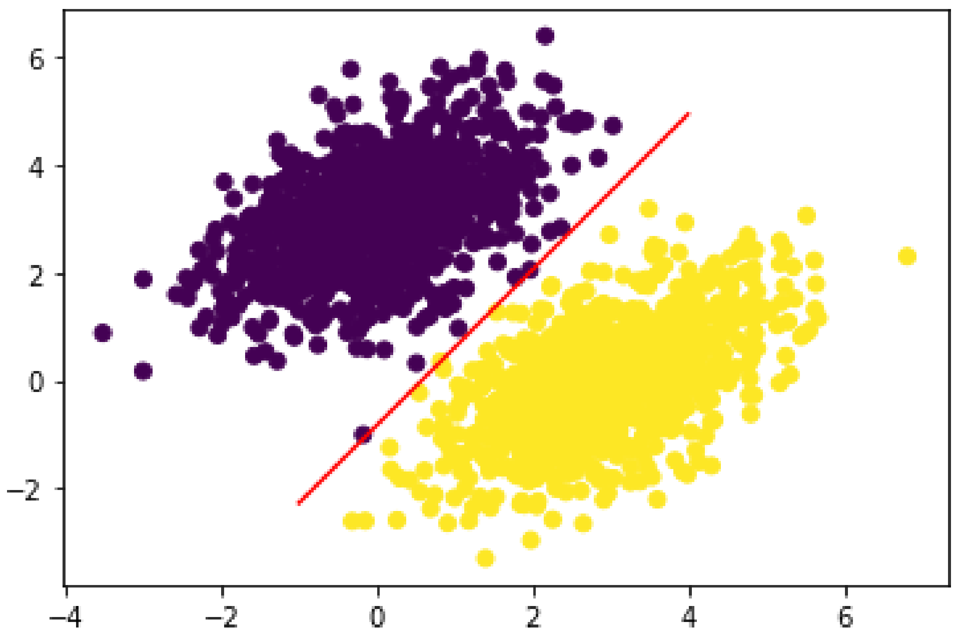


In [ ]:

```
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]
```

Out[ ]:

```
<matplotlib.collections.PathCollection at 0x7fa2db6dad10>
```



Anatomy of a neural network:

Understanding core Keras APIs

Layers: The building blocks of deep learning

The base Layer class in Keras

A `Dense` layer implemented as a `Layer` subclass

In [ ]:
```
from tensorflow import keras

class SimpleDense(keras.layers.Layer):

    def __init__(self, units, activation=None):
        super().__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.unit
                                 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
```

In [ ]:
```
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)
```

Automatic shape inference: Building layers on the fly

In [ ]:
```
from tensorflow.keras import layers
layer = layers.Dense(32, activation="relu")
```

In [ ]:
```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential([
    layers.Dense(32, activation="relu"),
```

```
    layers.Dense(32)
])
```

In [ ]:
```
model = keras.Sequential([
    SimpleDense(32, activation="relu"),
    SimpleDense(64, activation="relu"),
    SimpleDense(32, activation="relu"),
    SimpleDense(10, activation="softmax")
])
```

From layers to models

The "compile" step: Configuring the learning process

In [ ]:
```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer="rmsprop",
              loss="mean_squared_error",
              metrics=["accuracy"])
```

In [ ]:
```
model.compile(optimizer=keras.optimizers.RMSprop(),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])
```

Picking a loss function

Understanding the fit() method

Calling `fit()` with NumPy data

In [ ]:
```
history = model.fit(
    inputs,
    targets,
    epochs=5,
    batch_size=128
)
```

```
Epoch 1/5
16/16 [==============================] - 1s 2ms/step - loss: 
15.0589 - binary_accuracy: 0.0020
Epoch 2/5
16/16 [==============================] - 0s 2ms/step - loss: 
14.6048 - binary_accuracy: 0.0020
Epoch 3/5
16/16 [==============================] - 0s 2ms/step - loss: 
14.2170 - binary_accuracy: 0.0020
```

```
Epoch 4/5
16/16 [==============================] - 0s 2ms/step - loss: 13.8422 - binary_accuracy: 0.0020
Epoch 5/5
16/16 [==============================] - 0s 2ms/step - loss: 13.4738 - binary_accuracy: 0.0020
```

In [ ]:

```
history.history
```

Out[ ]:

```
{'loss': [15.058869361877441,
  14.604792594909668,
  14.217011451721191,
  13.84217357635498,
  13.473837852478027],
 'binary_accuracy': [0.0020000000949949026,
  0.0020000000949949026,
  0.0020000000949949026,
  0.0020000000949949026,
  0.0020000000949949026]}
```

Monitoring loss and metrics on validation data

Using the `validation_data` argument

In [ ]:

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer=keras.optimizers.RMSprop(learning_ra
              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)
)
```

```
Epoch 1/5
88/88 [==============================] - 1s 5ms/step - loss:
```

```
0.3618 - binary_accuracy: 0.8979 - val_loss: 0.0778 - val_bi
nary_accuracy: 0.9450
Epoch 2/5
88/88 [==============================] - 0s 3ms/step - loss:
0.0765 - binary_accuracy: 0.9536 - val_loss: 0.0733 - val_bi
nary_accuracy: 0.9617
Epoch 3/5
88/88 [==============================] - 0s 3ms/step - loss:
0.0721 - binary_accuracy: 0.9600 - val_loss: 0.0659 - val_bi
nary_accuracy: 0.9650
Epoch 4/5
88/88 [==============================] - 0s 3ms/step - loss:
0.0687 - binary_accuracy: 0.9564 - val_loss: 0.0287 - val_bi
nary_accuracy: 0.9967
Epoch 5/5
88/88 [==============================] - 0s 3ms/step - loss:
0.0708 - binary_accuracy: 0.9657 - val_loss: 0.0350 - val_bi
nary_accuracy: 1.0000
```

Out[ ]:

```
<keras.callbacks.History at 0x7fa2d9b5ce90>
```

Inference: Using a model after training

In [ ]:

```
predictions = model.predict(val_inputs, batch_size=128)
print(predictions[:10])
```

```
5/5 [==============================] - 0s 3ms/step
[[0.24031283]
 [0.84266007]
 [0.91847444]
 [0.14388952]
 [0.11975732]
 [0.08451074]
 [0.17176777]
 [0.09675378]
 [0.14447355]
 [0.07130358]]
```