Model Performance And Fit - 3

One should look for what is and not what he thinks should be. (Albert Einstein)

```
/opt/conda/envs/python-r-course-test/bin/python:1: DeprecationWarning: `import kerastuner`
is deprecated, please use `import keras_tuner`.
```

Warm up

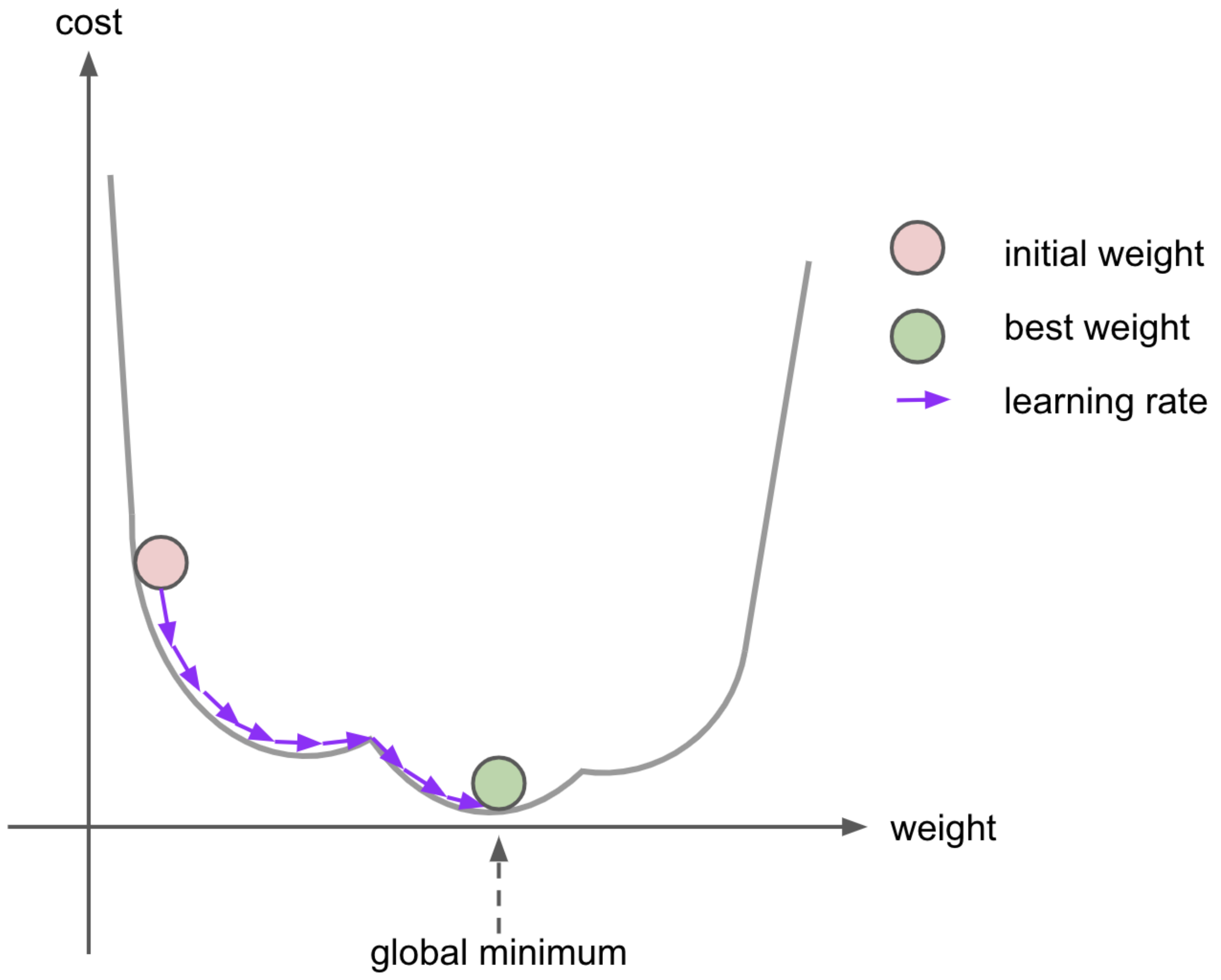
- Check out this blog with a list of Neural Network projects you can try: ***Link***
- What was the most interesting? Do you plan to try any of these projects? Can you think of other projects you want to work on?

Module completion checklist

Objective	Complete
Introduce loss functions in TensorFlow	
Backpropagation and gradient descent using TensorFlow	

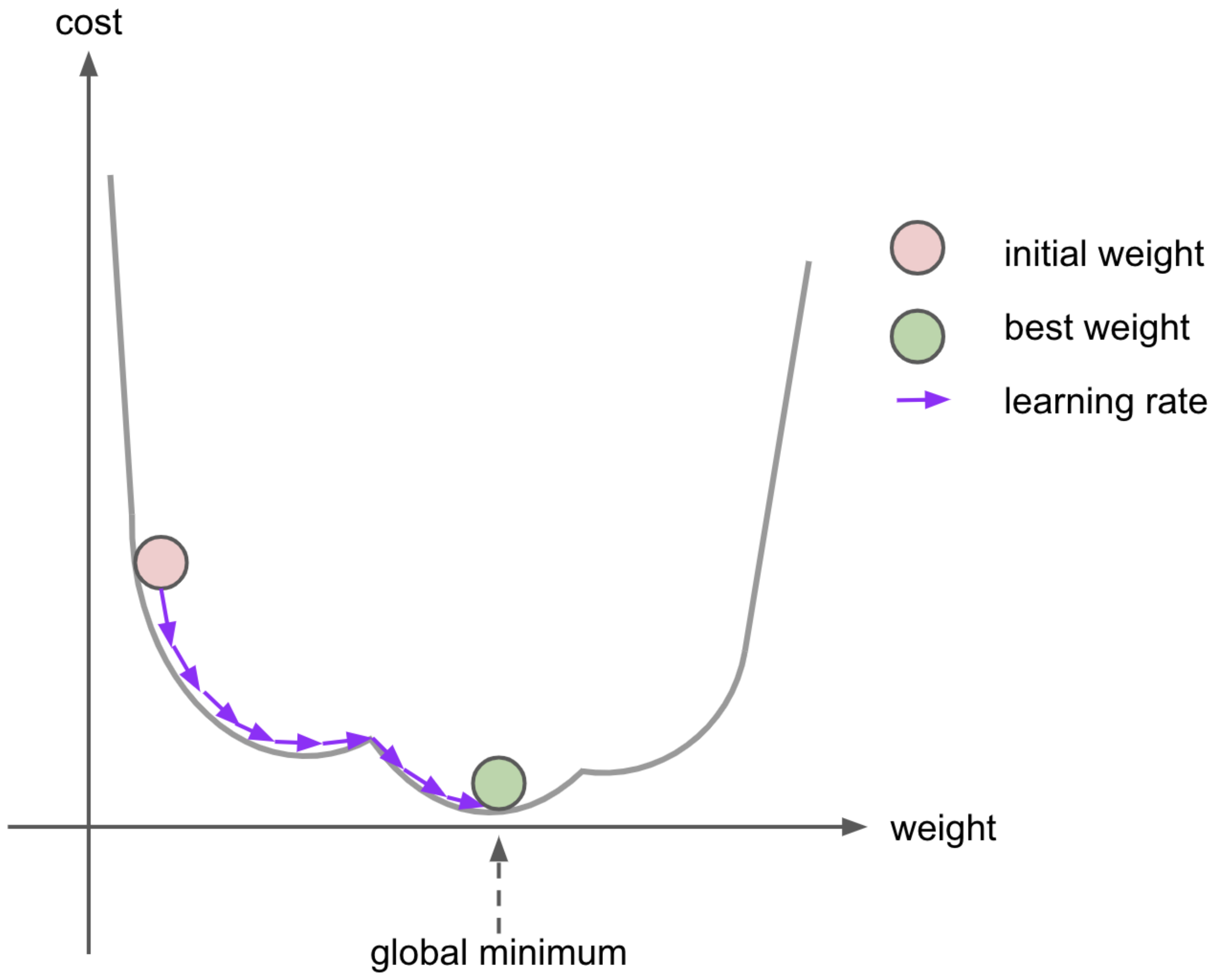
What is a loss function?

- The **loss function** (or cost function) is the function that describes the relationship between the neural network's weights and the error
- The value returned by this function is called **loss**



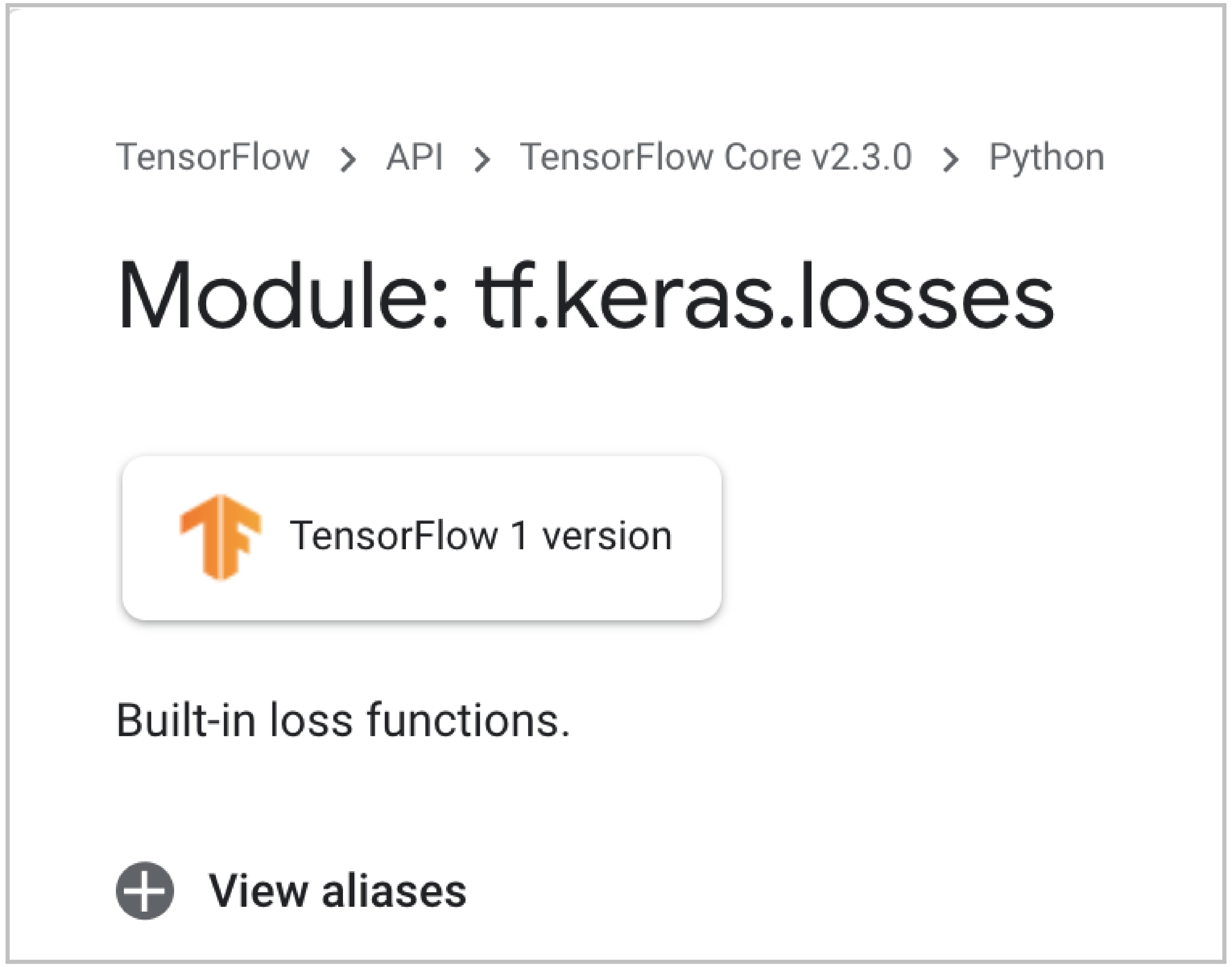
Loss function (cont'd)

- We aim to find weights that minimize the error
- Optimizers within a neural network framework (e.g., Keras) operate on a given **loss function** to find the global minimum



Keras loss functions

- There is a long list of loss functions available in the `tf.keras.losses` module, including *BinaryCrossentropy*, *CategoricalCrossentropy*, *Cosinesimilarity*, *MSE*, *MAE*, and many more
- We'll cover a few of the common functions
- Check out the rest on the TensorFlow website by using ***this link***



Loss function: binary crossentropy

- **Binary crossentropy** (a.k.a. log loss) is a metric that takes into account the *uncertainty of your prediction based on how much it varies from the actual label*
- It *uses the prediction probabilities of a class*, not the actual assigned class label
- It is widely used to assess conventional binary classification problems and be extended to multi-class classification problems

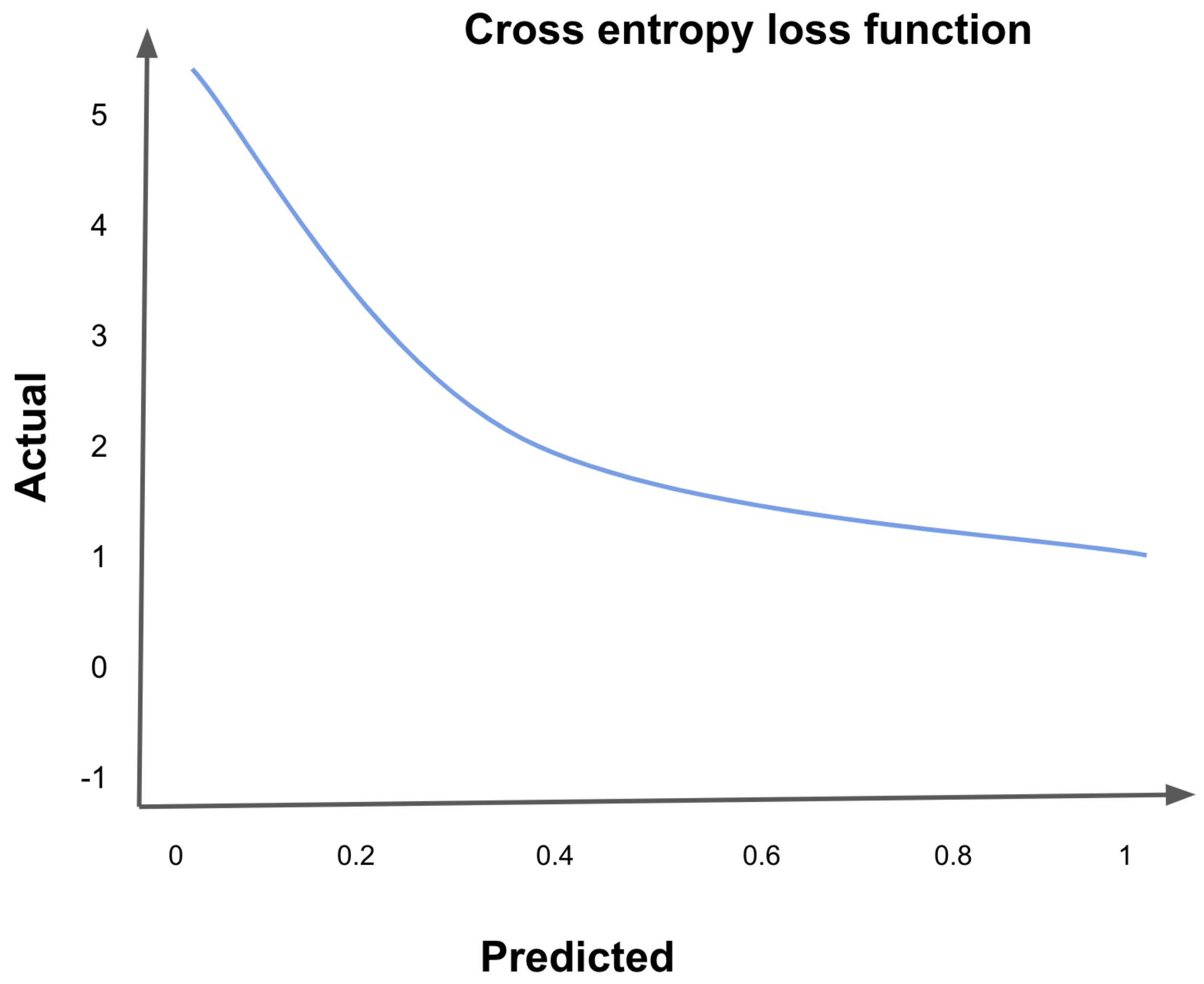
Binary crossentropy: formula and intuition

- Log loss should be minimized, so **the smaller the number, the better** (it can be 0 or greater)

- $L = \frac{-1}{N} \sum_1^N y_i \cdot \log(\hat{y_i} + (1 - y_i)) \cdot \log(1 - \hat{y_i})$, where
  - N is the number of observations in our model (i.e., the number of values in the `output` layer)
  - y_i is the target value
  - $\hat{y_i}$ is the predicted value

Binary crossentropy: formula and intuition (cont'd)

- The loss curve can be constructed to view how the value changes based on different probability thresholds



Loss function: categorical crossentropy

- **Categorical crossentropy** is a loss function that is suited for multi-class classification problems
  - *Multi-class classification problems involve more than two categories in the target variable*
- It's a generalized version of binary crossentropy and is computed in a similar fashion

- $L = \frac{-1}{N} \sum_1^N y_i \cdot \log(\hat{y}_i)$, where
  - N is the number of observations in our model (i.e., the number of values in the `output` layer)
  - y_i is the target value
  - $\hat{y}_i$ is the predicted value

Using categorical crossentropy in TensorFlow

- We need to provide the label inputs in the **one-hot-encoded format** while we use the categorical crossentropy as loss function
- One hot encoding is a process used to convert categorical variables to a numerical format
- For example, if we have a dataset that includes information three car manufacturers (BMW, Volkswagen, and Honda), the converted dataset would look as shown on the right

Original dataset

Company	Year	Price
1	1993	10000
2	2000	15000
3	2005	24000
1	2019	17000

After one-hot encoding

1	2	3	Price
1	0	0	10000
0	1	0	15000
0	0	1	24000
1	0	0	17000

1 - BMW, 2 - Volkswagen, 3 - Honda

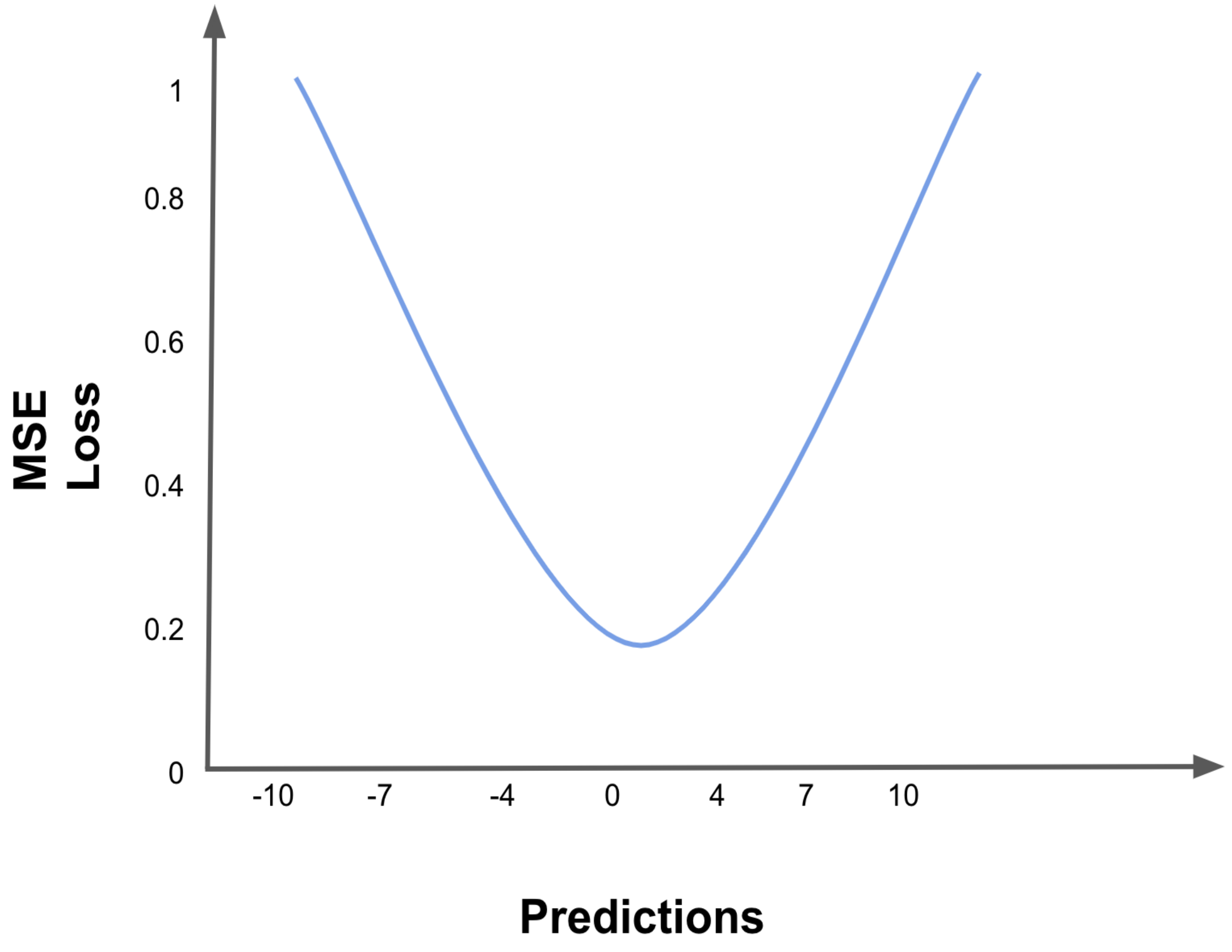
Loss function: sparse categorical crossentropy

- If we wish to provide the categorical labels as integers instead of one-hot encoding, we use the loss function - `SparseCategoricalCrossentropy`
- It’s equation for loss function is the same as categorical crossentropy

Loss function: mean squared error (MSE)

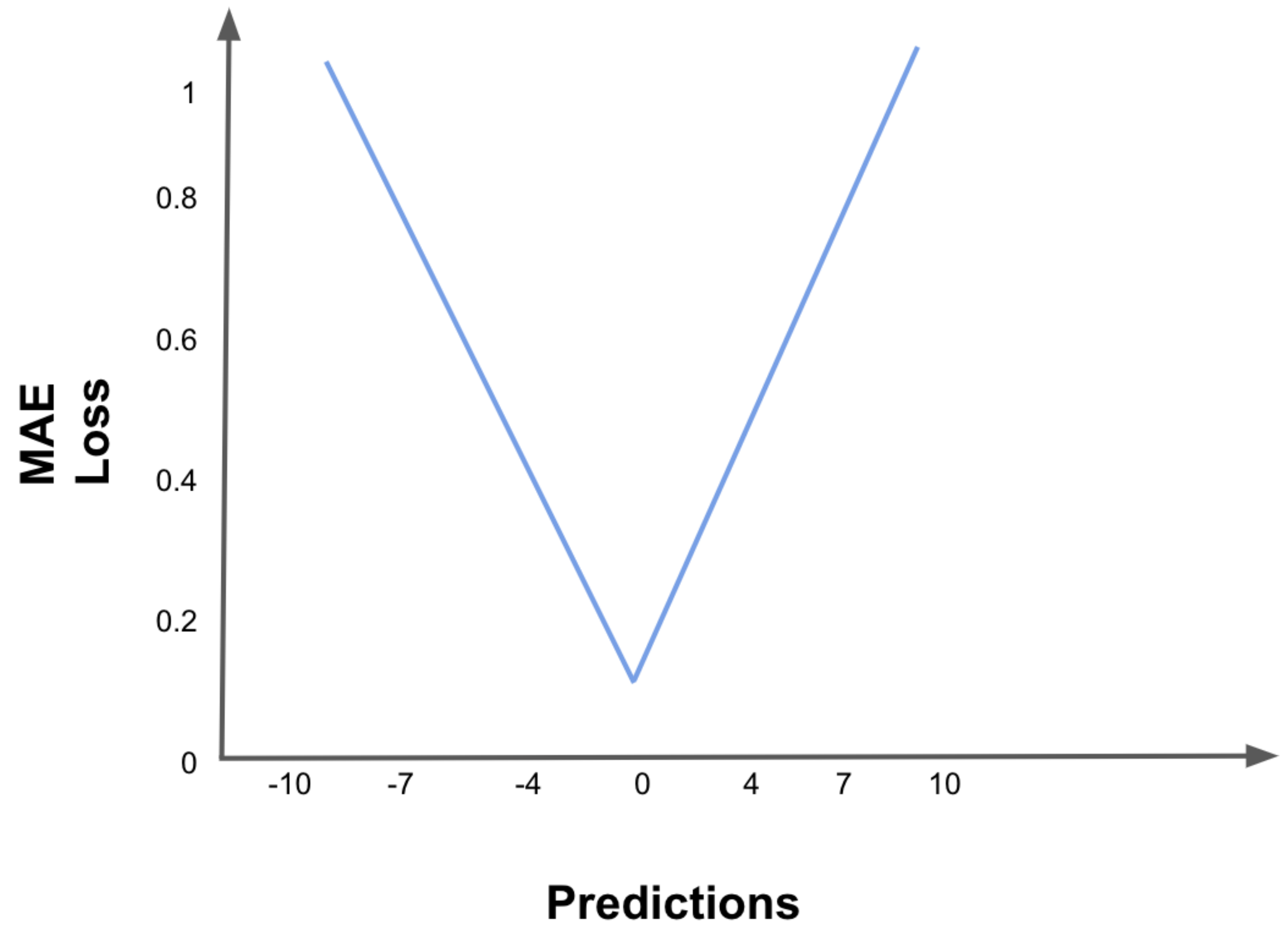
- **MSE** is used as loss function in regression problems
- It's measured as the average of squared difference between the predictions and the actual observations

- $MSE = \frac{\sum_{i=1}^{N}(y_i - \hat{y_i})^2}{N}$, where
  - N is the number of observations in our model (i.e. the number of values in the `output` layer)
  - y_i is the target value
  - $\hat{y_i}$ is the predicted value
- *Due to squaring, predictions that are far away from the actual values are penalized heavily compared to predictions closer to actual values*



Loss function: mean absolute error (MAE)

- **MAE** is also a loss function used in the regression problems
- It is calculated by taking the mean of the absolute differences between predicted and the actual values



- $MAE = (\frac{1}{N}) \sum_{i=1}^{N} |y_i - x_i|$, where
  - N is the number of observations in our model (i.e. the number of values in the `output` layer)
  - y_i is the target value
  - $\hat{y}_i$ is the predicted value
- *MAE is preferred to be chosen as the loss function when there are more outliers in the training data*

Loss functions summary

Loss function	Use case
Binary crossentropy	Binary classification problems
Categorical crossentropy	Multi-class classification problems with target labels in one-hot-encoding format
Sparse categorical crossentropy	Multi-class classification with target labels as integers
Mean squared error (MSE)	Regression problems
Mean absolute error	Regression problems

Module completion checklist

Objective	Complete
Introduce loss functions in TensorFlow	✔
Backpropagation and gradient descent using TensorFlow	

Overview

- Further down this section, we will reinforce our understanding of some neural network components and illustrate the mechanics of **backpropagation** on a simple **feedforward neural network**

- *Note: the gradients that we will deal with today will be vector representations of the derivative of the activation function*

Generate some fake data

- Let's generate some fake data and train our neural network
- Imagine that our data is drawn from a linear function:
  - $y = 3.5 * studyhours + 50$, where 3.5 is our true `weight` and 50 is our true `bias`

```
TRUE_W = 3.5          #<- true weight
TRUE_b = 50.0         #<- true bias
NUM_EXAMPLES = 1000   #<- number of observations
```

```
# Simulate inputs and noise from normal distribution.
inputs = tf.random.normal(shape=[NUM_EXAMPLES])
noise = tf.random.normal(shape=[NUM_EXAMPLES])
```

```
# Compute the outputs based on our equation.
outputs = inputs * TRUE_W + TRUE_b + noise
```

Neural network architecture

- Let's create a neural network class called `Model`
- **Note:** This is essentially a linear regression whose coefficients are trained by gradient descent
  - *In practice, gradient descent works with much more complex functions within multi-layer networks*

```
# Define model.
class Model(object):

  def __init__(self):
    self.W = tf.Variable(8.0)    #<- initial weight
    self.b = tf.Variable(40.0)   #<- initial bias

  def __call__(self, x):
    return self.W * x + self.b #<- compute the equation
```

```
# Initialize the model.
model = Model()
```

```
# Check if it outputs correct results.
assert model(3.0).numpy() == 64.0
```

Loss function

- Here we will use **Mean Squared Error (MSE)**, because this is a regression problem
- We are trying to predict a continuous target y

```
# Define loss function.
def loss(target_y, predicted_y):
  "MSE"
  return tf.reduce_mean(tf.square(target_y - predicted_y))
```

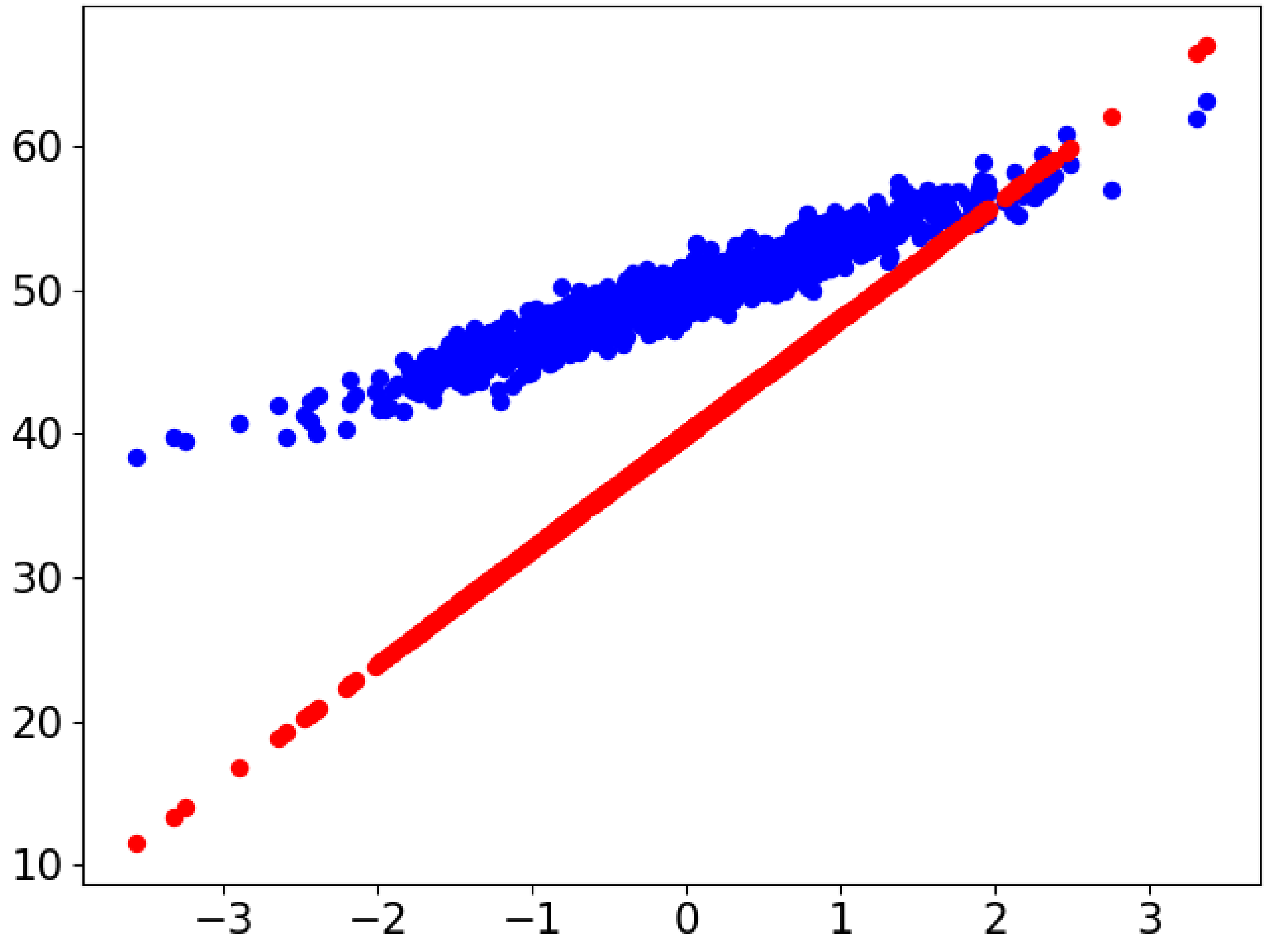
Initial weights

- The initial weights were chosen by us in this instance
- In practice, weights are initialized randomly

```
print('Current loss: %1.6f' % loss(model(inputs),
      outputs).numpy())
```

```
Current loss: 121.319221
```

```
plt.scatter(inputs, outputs, c = 'b')
plt.scatter(inputs, model(inputs), c = 'r')
plt.show()
```



Update weights based on gradient

- Due to the high complexity of models and their non-linearity, it is common for gradient descent to get stuck in a local minimum, but there are ways to combat this:
  - Stochastic Gradient Descent
  - More advanced GD-based optimizers
- Now let's define the training function

```
# Define the train function for our NN.
def train(model, inputs, outputs, learning_rate):

  with tf.GradientTape() as t:
     current_loss = loss(outputs, model(inputs)) #<- compute loss

  # Compute partial derivatives:
  # how much does a particular obvs + W + b contribute to that loss.
  dW, db = t.gradient(current_loss, [model.W, model.b])

  # Update with new weights and bias using our learning rate.
  model.W.assign_sub(learning_rate * dW)
  model.b.assign_sub(learning_rate * db)
```

Train the neural network

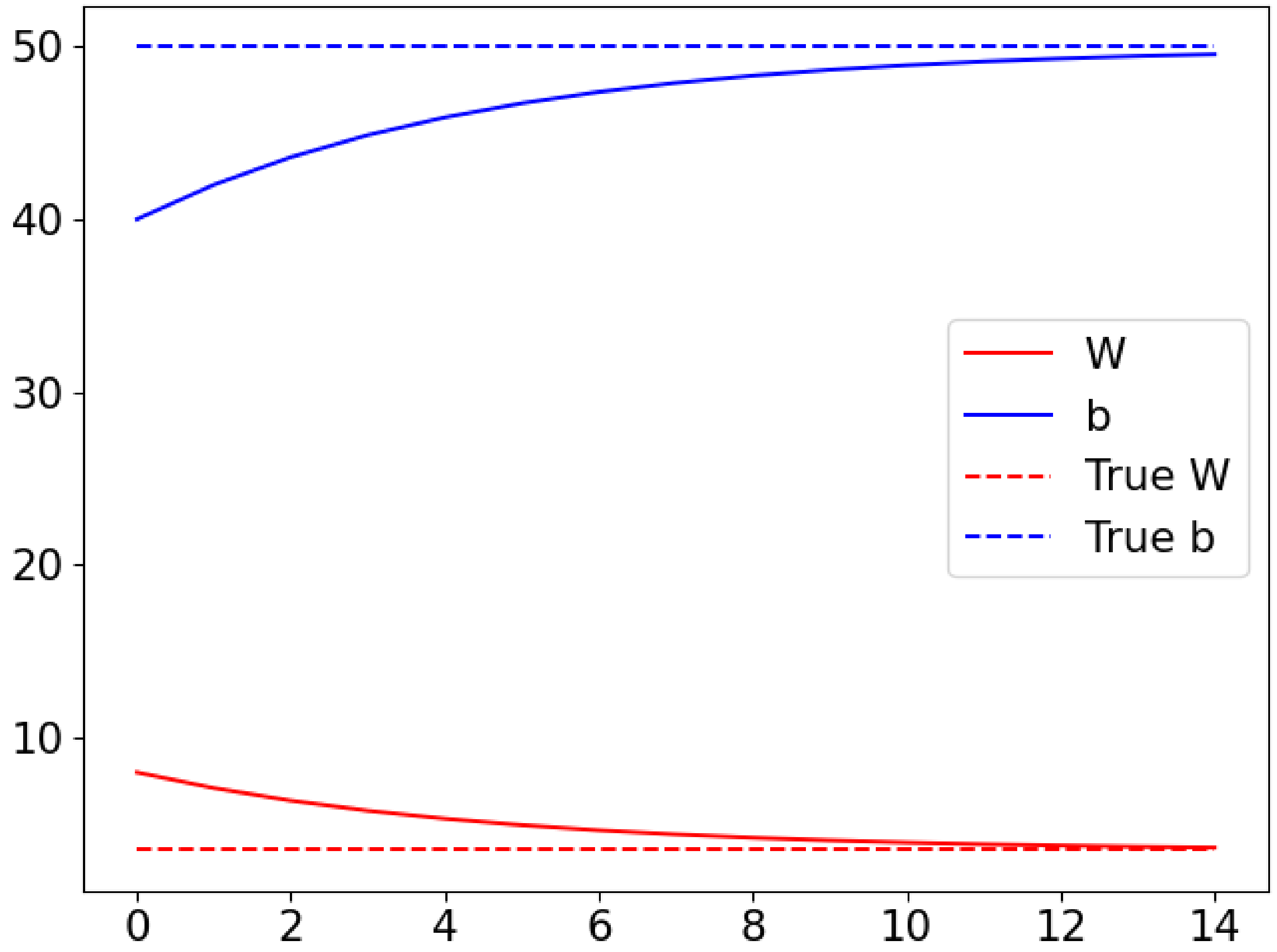
```
model = Model()

# Store some history of weights.
Ws, bs = [], []
epochs = range(15)

for epoch in epochs:
  Ws.append(model.W.numpy())
  bs.append(model.b.numpy())
  current_loss = loss(outputs, model(inputs))

  train(model, inputs, outputs, learning_rate=0.1)
  print('Epoch %2d: W=%1.2f b=%1.2f loss=%2.5f' % (epoch, Ws[-1], bs[-1], current_loss))
```

```
Epoch  0: W=8.00 b=40.00 loss=121.31922
Epoch  1: W=7.09 b=42.00 loss=78.03817
Epoch  2: W=6.37 b=43.59 loss=50.32687
Epoch  3: W=5.79 b=44.87 loss=32.58436
Epoch  4: W=5.32 b=45.89 loss=21.22446
Epoch  5: W=4.95 b=46.71 loss=13.95112
Epoch  6: W=4.65 b=47.37 loss=9.29426
Epoch  7: W=4.41 b=47.89 loss=6.31264
Epoch  8: W=4.22 b=48.31 loss=4.40362
Epoch  9: W=4.07 b=48.65 loss=3.18134
Epoch 10: W=3.95 b=48.91 loss=2.39876
Epoch 11: W=3.85 b=49.13 loss=1.89770
Epoch 12: W=3.77 b=49.30 loss=1.57689
Epoch 13: W=3.71 b=49.44 loss=1.37148
Epoch 14: W=3.66 b=49.55 loss=1.23997
```

Inspect the results

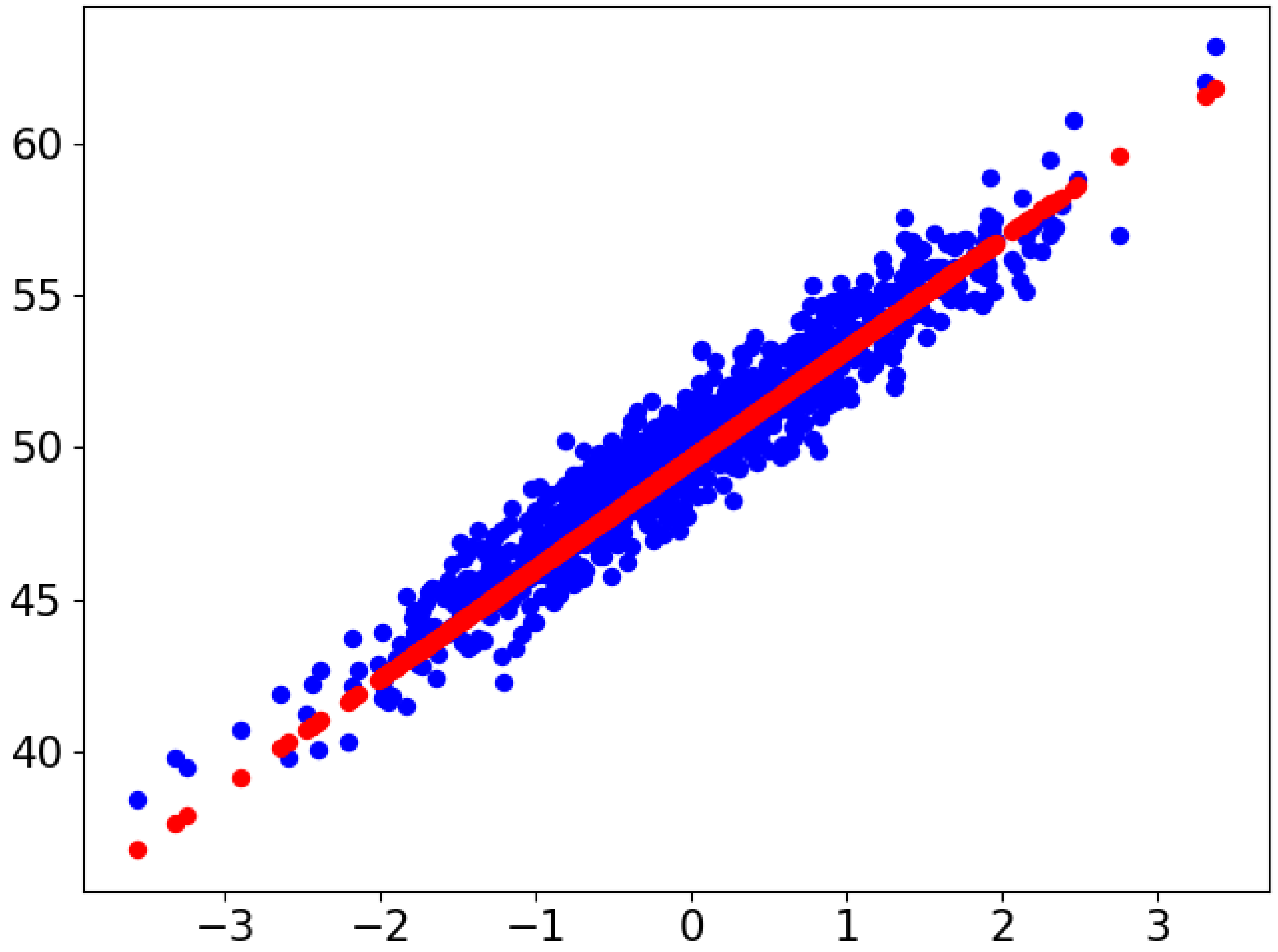
```
plt.plot(epochs, Ws, 'r', epochs, bs, 'b')
plt.plot([TRUE_W] * len(epochs), 'r--',
         [TRUE_b] * len(epochs), 'b--')
plt.legend(['W', 'b', 'True W', 'True b'])
plt.show()
```

- The weights and the bias converged very close to their true values



Inspect the results (cont'd)

```
plt.scatter(inputs, outputs, c='b')
plt.scatter(inputs, model(inputs), c='r')
plt.show()
```



Inspect the results (cont'd)

```
print('Current loss: %1.6f' % loss(model(inputs), outputs).numpy())
```

```
Current loss: 1.155764
```

- The fit of the model to our data is now very good as opposed to the initial starting point
- We can also see that in just `15` epochs our very simple neural network was able to go from a loss of `100+` down to just over `1`!

Knowledge check

Link: ***https://forms.gle/Fp9QjEDpENXyn8QM9***

Module completion checklist

Objective	Complete
Introduce loss functions in TensorFlow	✔
Backpropagation and gradient descent using TensorFlow	✔

Congratulations on completing this module!