Open in Colab

01. Neural Network Regression with TensorFlow

There are many definitions for a regression problem but in our case, we're going to simplify it to be: predicting a number.

For example, you might want to:

- Predict the selling price of houses given information about them (such as number of rooms, size, number of bathrooms).
- Predict the coordinates of a bounding box of an item in an image.
- Predict the cost of medical insurance for an individual given their demographics (age, sex, gender, race).

In this notebook, we're going to set the foundations for how you can take a sample of inputs (this is your data), build a neural network to discover patterns in those inputs and then make a prediction (in the form of a number) based on those inputs.

What we're going to cover

Specifically, we're going to go through doing the following with TensorFlow:

- Architecture of a regression model
- Input shapes and output shapes
  - `X`: features/data (inputs)
  - `y`: labels (outputs)
- Creating custom data to view and fit
- Steps in modelling
  - Creating a model
  - Compiling a model
    - Defining a loss function
    - Setting up an optimizer
    - Creating evaluation metrics
  - Fitting a model (getting it to find patterns in our data)
- Evaluating a model
  - Visualizng the model ("visualize, visualize, visualize")
  - Looking at training curves
  - Compare predictions to ground truth (using our evaluation metrics)
- Saving a model (so we can use it later)
- Loading a model

Don't worry if none of these make sense now, we're going to go through each.

How you can use this notebook

You can read through the descriptions and the code (it should all run), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to **write more code**.

Typical architecture of a regresison neural network

The word *typical* is on purpose.

Why?

Because there are many different ways (actually, there's almost an infinite number of ways) to write neural networks.

But the following is a generic setup for ingesting a collection of numbers, finding patterns in them and then outputing some kind of target number.

Yes, the previous sentence is vague but we'll see this in action shortly.

Hyperparameter	Typical value
Input layer shape	Same shape as number of features (e.g. 3 for # bedrooms, # bathrooms, # car spaces in housing price prediction)
Hidden layer(s)	Problem specific, minimum = 1, maximum = unlimited
Neurons per hidden layer	Problem specific, generally 10 to 100
Output layer shape	Same shape as desired prediction shape (e.g. 1 for house price)
Hidden activation	Usually ReLU (rectified linear unit)
Output activation	None, ReLU, logistic/tanh
Loss function	MSE (mean square error) or MAE (mean absolute error)/Huber (combination of MAE/MSE) if outliers
Optimizer	SGD (stochastic gradient descent), Adam

***Table 1:** Typical architecture of a regression network. **Source:** Adapted from page 293 of Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow Book by Aurélien Géron*

Again, if you're new to neural networks and deep learning in general, much of the above table won't make sense. But don't worry, we'll be getting hands-on with all of it soon.

> **Note:** A **hyperparameter** in machine learning is something a data analyst or developer can set themselves, where as a **parameter** usually describes something a model learns on its own (a value not explicitly set by an analyst).

Okay, enough talk, let's get started writing code.

To use TensorFlow, we'll import it as the common alias `tf` (short for TensorFlow).

In [ ]:

```
import tensorflow as tf
print(tf.__version__) # check the version (should be 2.x+)
```

```
2.3.0
```

Creating data to view and fit

Since we're working on a **regression problem** (predicting a number) let's create some linear data (a straight line) to model.

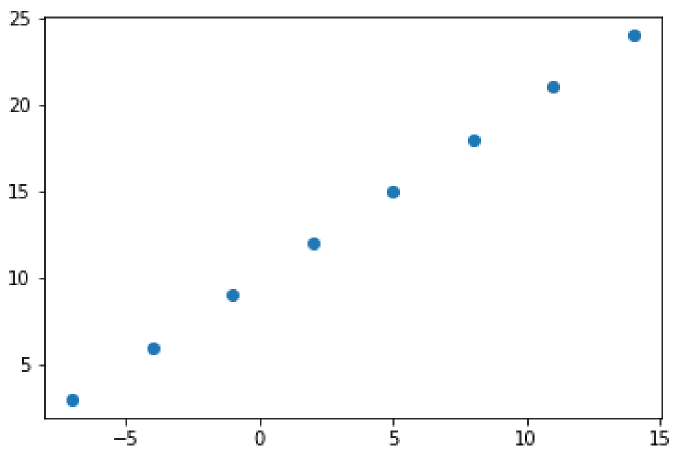
In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt

# Create features
X = np.array([-7.0, -4.0, -1.0, 2.0, 5.0, 8.0, 11.0, 14.0])

# Create labels
y = np.array([3.0, 6.0, 9.0, 12.0, 15.0, 18.0, 21.0, 24.0])
```

```
# Visualize it
plt.scatter(X, y);
```



Before we do any modelling, can you calculate the pattern between `X` and `y`?

For example, say I asked you, based on this data what the `y` value would be if `X` was 17.0?

Or how about if `X` was -10.0?

This kind of pattern discover is the essence of what we'll be building neural networks to do for us.

Regression input shapes and output shapes

One of the most important concepts when working with neural networks are the input and output shapes.

The input shape is the shape of your data that goes into the model.

The output shape is the shape of your data you want to come out of your model.

These will differ depending on the problem you're working on.

Neural networks accept numbers and output numbers. These numbers are typically represented as tensors (or arrays).

Before, we created data using NumPy arrays, but we could do the same with tensors.

In [ ]:

```
# Example input and output shapes of a regresson model
house_info = tf.constant(["bedroom", "bathroom", "garage"])
house_price = tf.constant([939700])
house_info, house_price
```

Out[ ]:

```
(<tf.Tensor: shape=(3,), dtype=string, numpy=array([b'bedroom', b'bathroom', b'garage'],
dtype=object)>,
 <tf.Tensor: shape=(1,), dtype=int32, numpy=array([939700], dtype=int32)>)
```

In [ ]:

```
house_info.shape
```

Out[ ]:

```
TensorShape([3])
```

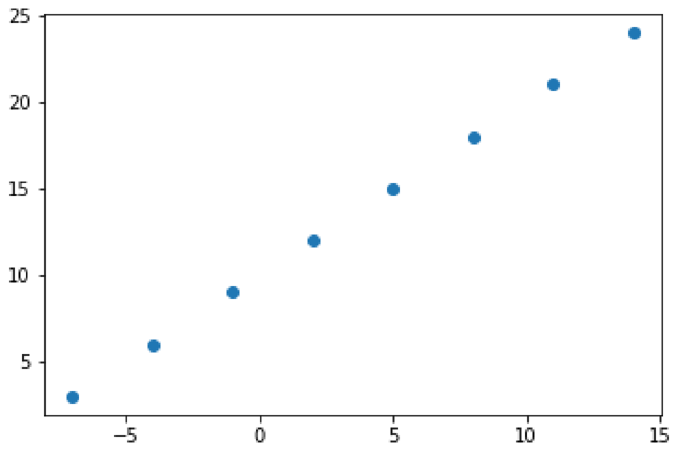
In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Create features (using tensors)
X = tf.constant([-7.0, -4.0, -1.0, 2.0, 5.0, 8.0, 11.0, 14.0])

# Create labels (using tensors)
y = tf.constant([3.0, 6.0, 9.0, 12.0, 15.0, 18.0, 21.0, 24.0])

# Visualize it
plt.scatter(X, y);
```



Our goal here will be to use `X` to predict `y`.

So our *input* will be `X` and our *output* will be `y`.

Knowing this, what do you think our input and output shapes will be?

Let's take a look.

In [ ]:

```
# Take a single example of X
input_shape = X[0].shape

# Take a single example of y
output_shape = y[0].shape

input_shape, output_shape # these are both scalars (no shape)
```

Out[ ]:

```
(TensorShape([]), TensorShape([]))
```

Huh?

From this it seems our inputs and outputs have no shape?

How could that be?

It's because no matter what kind of data we pass to our model, it's always going to take as input and return as ouput some kind of tensor.

But in our case because of our dataset (only 2 small lists of numbers), we're looking at a special kind of tensor, more specificially a rank 0 tensor or a scalar.

In [ ]:

```
# Let's take a look at the single examples invidually
X[0], y[0]
```

Out[ ]:

```
(<tf.Tensor: shape=(), dtype=float32, numpy=-7.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=3.0>)
```

In our case, we're trying to build a model to predict the pattern between `X[0]` equalling `-7.0` and `y[0]`

equalling `3.0`.

So now we get our answer, we're trying to use 1 `X` value to predict 1 `y` value.

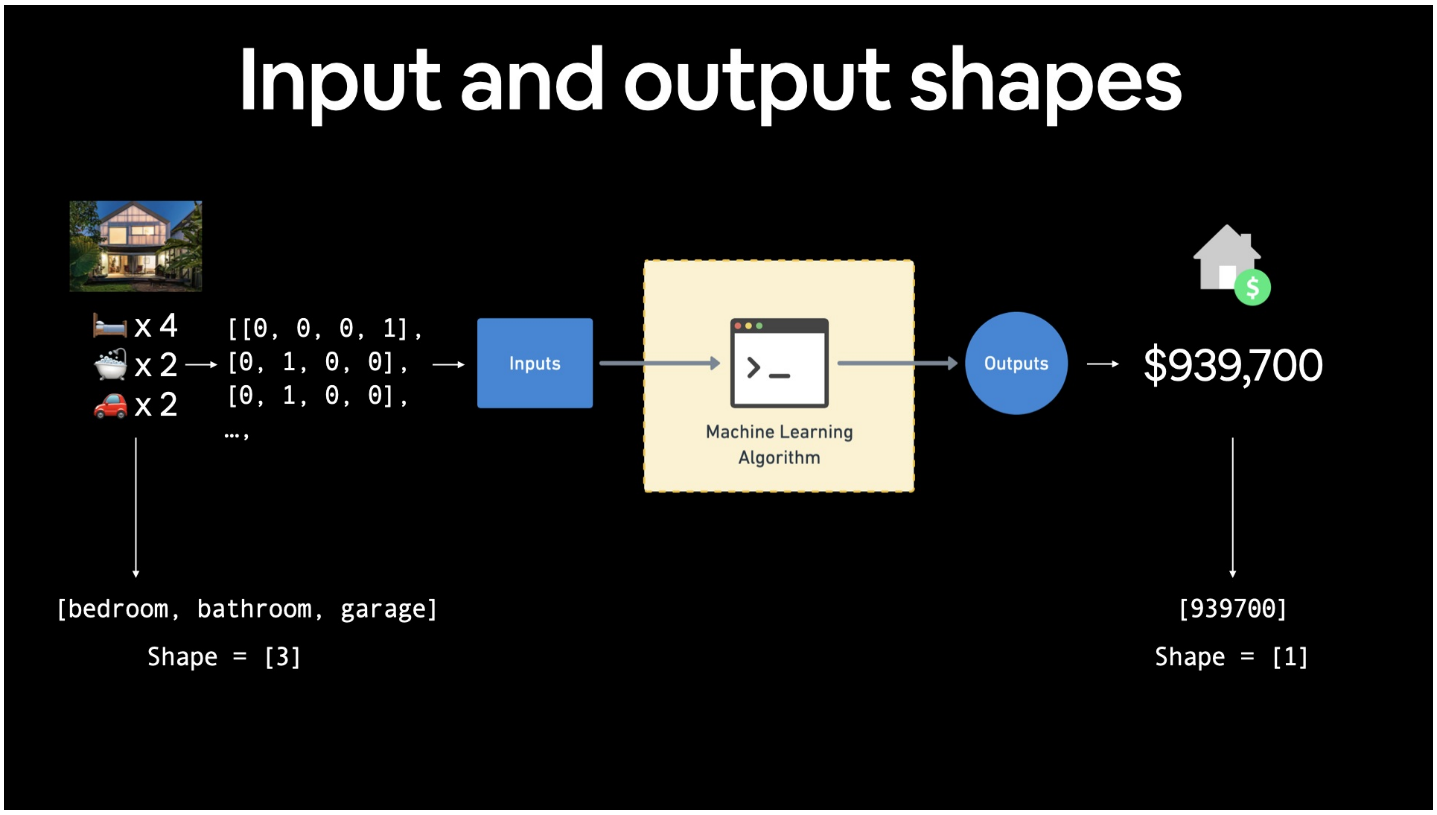
You might be thinking, "this seems pretty complicated for just predicting a straight line...".

And you'd be right.

But the concepts we're covering here, the concepts of input and output shapes to a model are fundamental.

In fact, they're probably two of the things you'll spend the most time on when you work with neural networks: **making sure your input and outputs are in the correct shape**.

If it doesn't make sense now, we'll see plenty more examples later on (soon you'll notice the input and output shapes can be almost anything you can imagine).



If you were working on building a machine learning algorithm for predicting housing prices, your inputs may be number of bedrooms, number of bathrooms and number of garages, giving you an input shape of 3 (3 different features). And since you're trying to predict the price of the house, your output shape would be 1.

Steps in modelling with TensorFlow

Now we know what data we have as well as the input and output shapes, let's see how we'd build a neural network to model it.

In TensorFlow, there are typically 3 fundamental steps to creating and training a model.

1. **Creating a model** - piece together the layers of a neural network yourself (using the Functional or Sequential API) or import a previously built model (known as transfer learning).
2. **Compiling a model** - defining how a models performance should be measured (loss/metrics) as well as defining how it should improve (optimizer).
3. **Fitting a model** - letting the model try to find patterns in the data (how does `X` get to `y`).

Let's see these in action using the Keras Sequential API to build a model for our regression data. And then we'll step through each.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)
```

```
# Create a model using the Sequential API
model = tf.keras.Sequential([
  tf.keras.layers.Dense(1)
])

# Compile the model
model.compile(loss=tf.keras.losses.mae, # mae is short for mean absolute error
              optimizer=tf.keras.optimizers.SGD(), # SGD is short for stochastic gradien
t descent
              metrics=["mae"])

# Fit the model
model.fit(X, y, epochs=5)
```

```
Epoch 1/5
1/1 [==============================] - 0s 1ms/step - loss: 11.5048 - mae: 11.5048
Epoch 2/5
1/1 [==============================] - 0s 1ms/step - loss: 11.3723 - mae: 11.3723
Epoch 3/5
1/1 [==============================] - 0s 887us/step - loss: 11.2398 - mae: 11.2398
Epoch 4/5
1/1 [==============================] - 0s 3ms/step - loss: 11.1073 - mae: 11.1073
Epoch 5/5
1/1 [==============================] - 0s 2ms/step - loss: 10.9748 - mae: 10.9748
```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa239e64be0>
```

Boom!

We've just trained a model to figure out the patterns between `X` and `y`.

How do you think it went?

In [ ]:

```
# Check out X and y
X, y
```

Out[ ]:

```
(<tf.Tensor: shape=(8,), dtype=float32, numpy=array([-7., -4., -1.,  2.,  5.,  8., 11., 1
4.], dtype=float32)>,
 <tf.Tensor: shape=(8,), dtype=float32, numpy=array([ 3.,  6.,  9., 12., 15., 18., 21., 2
4.], dtype=float32)>)
```

What do you think the outcome should be if we passed our model an `X` value of 17.0?

In [ ]:

```
# Make a prediction with the model
model.predict([17.0])
```

Out[ ]:

```
array([[12.716021]], dtype=float32)
```

It doesn't go very well... it should've output something close to 27.0.

> **Question:** What's Keras? I thought we were working with TensorFlow but every time we write TensorFlow code, `keras` comes after `tf` (e.g. `tf.keras.layers.Dense()`)?

Before TensorFlow 2.0+, Keras was an API designed to be able to build deep learning models with ease. Since TensorFlow 2.0+, its functionality has been tightly integrated within the TensorFlow library.

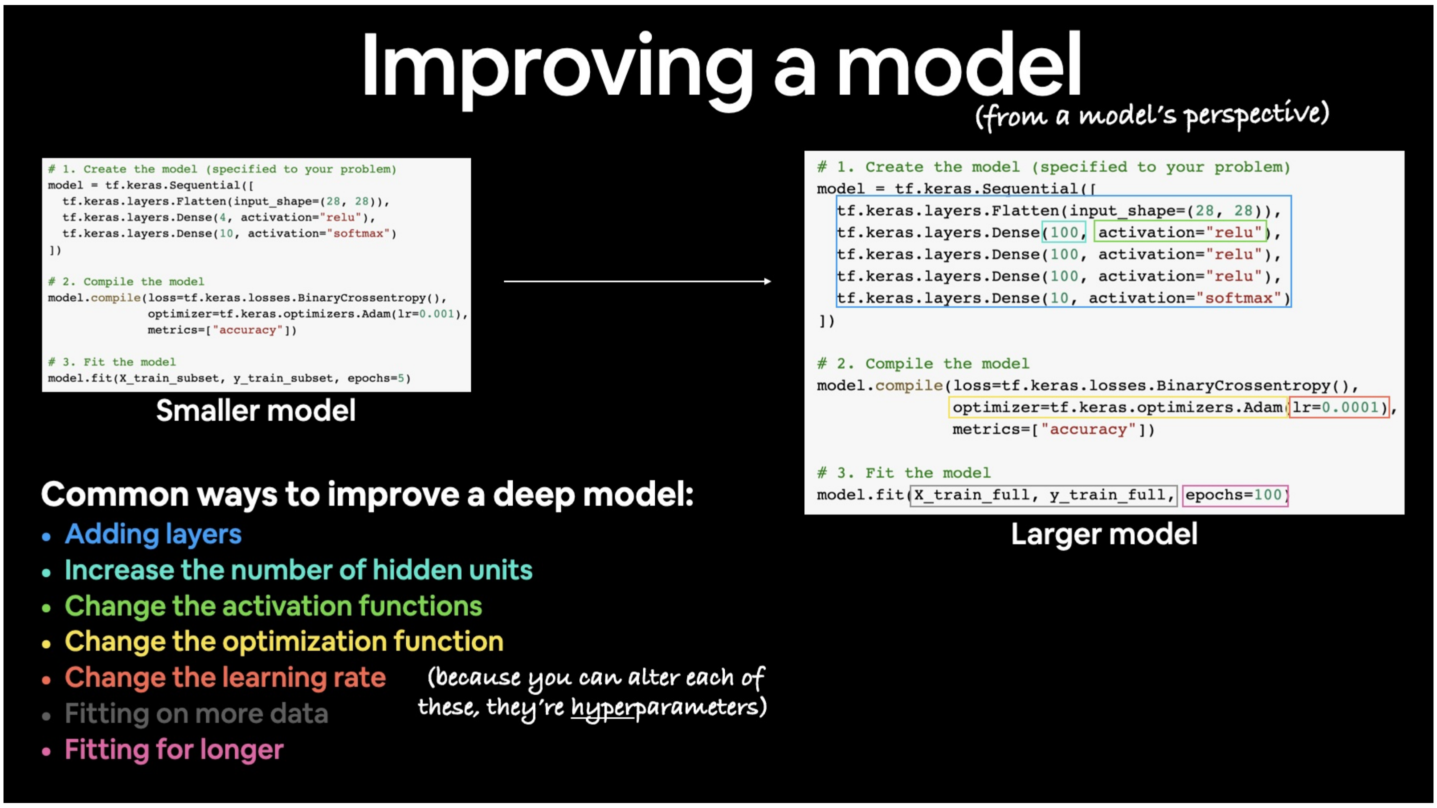
Improving a model

How do you think you'd improve upon our current model?

If you guessed by tweaking some of the things we did above, you'd be correct.

To improve our model, we alter almost every part of the 3 steps we went through before.

1. **Creating a model** - here you might want to add more layers, increase the number of hidden units (also called neurons) within each layer, change the activation functions of each layer.
2. **Compiling a model** - you might want to choose optimization function or perhaps change the **learning rate** of the optimization function.
3. **Fitting a model** - perhaps you could fit a model for more **epochs** (leave it training for longer) or on more data (give the model more examples to learn from).



Improving a model

(from a model's perspective)

```
# 1. Create the model (specified to your problem)
model = tf.keras.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(4, activation="relu"),
  tf.keras.layers.Dense(10, activation="softmax")
])

# 2. Compile the model
model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
              optimizer=tf.keras.optimizers.Adam(lr=0.001),
              metrics=["accuracy"])

# 3. Fit the model
model.fit(X_train_subset, y_train_subset, epochs=5)
```

Smaller model

```
# 1. Create the model (specified to your problem)
model = tf.keras.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(100, activation="relu"),
  tf.keras.layers.Dense(100, activation="relu"),
  tf.keras.layers.Dense(100, activation="relu"),
  tf.keras.layers.Dense(10, activation="softmax")
])

# 2. Compile the model
model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
              optimizer=tf.keras.optimizers.Adam(lr=0.0001),
              metrics=["accuracy"])

# 3. Fit the model
model.fit(X_train_full, y_train_full, epochs=100)
```

Larger model

Common ways to improve a deep model:

- Adding layers
- Increase the number of hidden units
- Change the activation functions
- Change the optimization function
- Change the learning rate
- Fitting on more data
- Fitting for longer

(because you can alter each of these, they're hyperparameters)

There are many different ways to potentially improve a neural network. Some of the most common include: increasing the number of layers (making the network deeper), increasing the number of hidden units (making the network wider) and changing the learning rate. Because these values are all human-changeable, they're referred to as [hyperparameters](#)) and the practice of trying to find the best hyperparameters is referred to as [hyperparameter tuning](#).

Woah. We just introduced a bunch of possible steps. The important thing to remember is how you alter each of these will depend on the problem you're working on.

And the good thing is, over the next few problems, we'll get hands-on with all of them.

For now, let's keep it simple, all we'll do is train our model for longer (everything else will stay the same).

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a model (same as above)
model = tf.keras.Sequential([
  tf.keras.layers.Dense(1)
])

# Compile model (same as above)
model.compile(loss=tf.keras.losses.mae,
              optimizer=tf.keras.optimizers.SGD(),
              metrics=["mae"])

# Fit model (this time we'll train for longer)
model.fit(X, y, epochs=100) # train for 100 epochs not 10
```

```
Epoch 1/100
1/1 [==============================] - 0s 2ms/step - loss: 11.5048 - mae: 11.5048
Epoch 2/100
1/1 [==============================] - 0s 2ms/step - loss: 11.3723 - mae: 11.3723
Epoch 3/100
1/1 [==============================] - 0s 2ms/step - loss: 11.2398 - mae: 11.2398
Epoch 4/100
1/1 [==============================] - 0s 2ms/step - loss: 11.1073 - mae: 11.1073
Epoch 5/100
1/1 [==============================] - 0s 2ms/step - loss: 10.9748 - mae: 10.9748
Epoch 6/100
1/1 [==============================] - 0s 2ms/step - loss: 10.8423 - mae: 10.8423
Epoch 7/100
1/1 [==============================] - 0s 2ms/step - loss: 10.7098 - mae: 10.7098
Epoch 8/100
1/1 [==============================] - 0s 2ms/step - loss: 10.5773 - mae: 10.5773
Epoch 9/100
1/1 [==============================] - 0s 2ms/step - loss: 10.4448 - mae: 10.4448
Epoch 10/100
1/1 [==============================] - 0s 2ms/step - loss: 10.3123 - mae: 10.3123
Epoch 11/100
1/1 [==============================] - 0s 2ms/step - loss: 10.1798 - mae: 10.1798
Epoch 12/100
1/1 [==============================] - 0s 2ms/step - loss: 10.0473 - mae: 10.0473
Epoch 13/100
1/1 [==============================] - 0s 2ms/step - loss: 9.9148 - mae: 9.9148
Epoch 14/100
1/1 [==============================] - 0s 2ms/step - loss: 9.7823 - mae: 9.7823
Epoch 15/100
1/1 [==============================] - 0s 2ms/step - loss: 9.6498 - mae: 9.6498
Epoch 16/100
1/1 [==============================] - 0s 2ms/step - loss: 9.5173 - mae: 9.5173
Epoch 17/100
1/1 [==============================] - 0s 2ms/step - loss: 9.3848 - mae: 9.3848
Epoch 18/100
1/1 [==============================] - 0s 2ms/step - loss: 9.2523 - mae: 9.2523
Epoch 19/100
1/1 [==============================] - 0s 2ms/step - loss: 9.1198 - mae: 9.1198
Epoch 20/100
1/1 [==============================] - 0s 2ms/step - loss: 8.9873 - mae: 8.9873
Epoch 21/100
1/1 [==============================] - 0s 2ms/step - loss: 8.8548 - mae: 8.8548
Epoch 22/100
1/1 [==============================] - 0s 2ms/step - loss: 8.7223 - mae: 8.7223
Epoch 23/100
1/1 [==============================] - 0s 2ms/step - loss: 8.5898 - mae: 8.5898
Epoch 24/100
1/1 [==============================] - 0s 2ms/step - loss: 8.4573 - mae: 8.4573
Epoch 25/100
1/1 [==============================] - 0s 2ms/step - loss: 8.3248 - mae: 8.3248
Epoch 26/100
1/1 [==============================] - 0s 2ms/step - loss: 8.1923 - mae: 8.1923
Epoch 27/100
1/1 [==============================] - 0s 2ms/step - loss: 8.0598 - mae: 8.0598
Epoch 28/100
1/1 [==============================] - 0s 2ms/step - loss: 7.9273 - mae: 7.9273
Epoch 29/100
1/1 [==============================] - 0s 2ms/step - loss: 7.7948 - mae: 7.7948
Epoch 30/100
1/1 [==============================] - 0s 2ms/step - loss: 7.6623 - mae: 7.6623
Epoch 31/100
1/1 [==============================] - 0s 3ms/step - loss: 7.5298 - mae: 7.5298
Epoch 32/100
1/1 [==============================] - 0s 3ms/step - loss: 7.3973 - mae: 7.3973
Epoch 33/100
1/1 [==============================] - 0s 2ms/step - loss: 7.2648 - mae: 7.2648
Epoch 34/100
1/1 [==============================] - 0s 2ms/step - loss: 7.2525 - mae: 7.2525
Epoch 35/100
1/1 [==============================] - 0s 3ms/step - loss: 7.2469 - mae: 7.2469
Epoch 36/100
1/1 [==============================] - 0s 2ms/step - loss: 7.2413 - mae: 7.2413
```

```
Epoch 37/100
1/1 [==============================] - 0s 2ms/step - loss: 7.2356 - mae: 7.2356
Epoch 38/100
1/1 [==============================] - 0s 2ms/step - loss: 7.2300 - mae: 7.2300
Epoch 39/100
1/1 [==============================] - 0s 2ms/step - loss: 7.2244 - mae: 7.2244
Epoch 40/100
1/1 [==============================] - 0s 3ms/step - loss: 7.2188 - mae: 7.2188
Epoch 41/100
1/1 [==============================] - 0s 3ms/step - loss: 7.2131 - mae: 7.2131
Epoch 42/100
1/1 [==============================] - 0s 3ms/step - loss: 7.2075 - mae: 7.2075
Epoch 43/100
1/1 [==============================] - 0s 5ms/step - loss: 7.2019 - mae: 7.2019
Epoch 44/100
1/1 [==============================] - 0s 2ms/step - loss: 7.1963 - mae: 7.1963
Epoch 45/100
1/1 [==============================] - 0s 5ms/step - loss: 7.1906 - mae: 7.1906
Epoch 46/100
1/1 [==============================] - 0s 3ms/step - loss: 7.1850 - mae: 7.1850
Epoch 47/100
1/1 [==============================] - 0s 3ms/step - loss: 7.1794 - mae: 7.1794
Epoch 48/100
1/1 [==============================] - 0s 5ms/step - loss: 7.1738 - mae: 7.1738
Epoch 49/100
1/1 [==============================] - 0s 2ms/step - loss: 7.1681 - mae: 7.1681
Epoch 50/100
1/1 [==============================] - 0s 3ms/step - loss: 7.1625 - mae: 7.1625
Epoch 51/100
1/1 [==============================] - 0s 5ms/step - loss: 7.1569 - mae: 7.1569
Epoch 52/100
1/1 [==============================] - 0s 2ms/step - loss: 7.1512 - mae: 7.1512
Epoch 53/100
1/1 [==============================] - 0s 2ms/step - loss: 7.1456 - mae: 7.1456
Epoch 54/100
1/1 [==============================] - 0s 3ms/step - loss: 7.1400 - mae: 7.1400
Epoch 55/100
1/1 [==============================] - 0s 8ms/step - loss: 7.1344 - mae: 7.1344
Epoch 56/100
1/1 [==============================] - 0s 2ms/step - loss: 7.1287 - mae: 7.1287
Epoch 57/100
1/1 [==============================] - 0s 6ms/step - loss: 7.1231 - mae: 7.1231
Epoch 58/100
1/1 [==============================] - 0s 1ms/step - loss: 7.1175 - mae: 7.1175
Epoch 59/100
1/1 [==============================] - 0s 3ms/step - loss: 7.1119 - mae: 7.1119
Epoch 60/100
1/1 [==============================] - 0s 1ms/step - loss: 7.1063 - mae: 7.1063
Epoch 61/100
1/1 [==============================] - 0s 1ms/step - loss: 7.1006 - mae: 7.1006
Epoch 62/100
1/1 [==============================] - 0s 1ms/step - loss: 7.0950 - mae: 7.0950
Epoch 63/100
1/1 [==============================] - 0s 1ms/step - loss: 7.0894 - mae: 7.0894
Epoch 64/100
1/1 [==============================] - 0s 4ms/step - loss: 7.0838 - mae: 7.0838
Epoch 65/100
1/1 [==============================] - 0s 1ms/step - loss: 7.0781 - mae: 7.0781
Epoch 66/100
1/1 [==============================] - 0s 2ms/step - loss: 7.0725 - mae: 7.0725
Epoch 67/100
1/1 [==============================] - 0s 4ms/step - loss: 7.0669 - mae: 7.0669
Epoch 68/100
1/1 [==============================] - 0s 2ms/step - loss: 7.0613 - mae: 7.0613
Epoch 69/100
1/1 [==============================] - 0s 3ms/step - loss: 7.0556 - mae: 7.0556
Epoch 70/100
1/1 [==============================] - 0s 1ms/step - loss: 7.0500 - mae: 7.0500
Epoch 71/100
1/1 [==============================] - 0s 985us/step - loss: 7.0444 - mae: 7.0444
Epoch 72/100
1/1 [==============================] - 0s 1ms/step - loss: 7.0388 - mae: 7.0388
```

```
Epoch 73/100
1/1 [==============================] - 0s 1ms/step - loss: 7.0331 - mae: 7.0331
Epoch 74/100
1/1 [==============================] - 0s 1ms/step - loss: 7.0275 - mae: 7.0275
Epoch 75/100
1/1 [==============================] - 0s 4ms/step - loss: 7.0219 - mae: 7.0219
Epoch 76/100
1/1 [==============================] - 0s 4ms/step - loss: 7.0163 - mae: 7.0163
Epoch 77/100
1/1 [==============================] - 0s 4ms/step - loss: 7.0106 - mae: 7.0106
Epoch 78/100
1/1 [==============================] - 0s 3ms/step - loss: 7.0050 - mae: 7.0050
Epoch 79/100
1/1 [==============================] - 0s 3ms/step - loss: 6.9994 - mae: 6.9994
Epoch 80/100
1/1 [==============================] - 0s 1ms/step - loss: 6.9938 - mae: 6.9938
Epoch 81/100
1/1 [==============================] - 0s 2ms/step - loss: 6.9881 - mae: 6.9881
Epoch 82/100
1/1 [==============================] - 0s 3ms/step - loss: 6.9825 - mae: 6.9825
Epoch 83/100
1/1 [==============================] - 0s 2ms/step - loss: 6.9769 - mae: 6.9769
Epoch 84/100
1/1 [==============================] - 0s 9ms/step - loss: 6.9713 - mae: 6.9713
Epoch 85/100
1/1 [==============================] - 0s 2ms/step - loss: 6.9656 - mae: 6.9656
Epoch 86/100
1/1 [==============================] - 0s 1ms/step - loss: 6.9600 - mae: 6.9600
Epoch 87/100
1/1 [==============================] - 0s 2ms/step - loss: 6.9544 - mae: 6.9544
Epoch 88/100
1/1 [==============================] - 0s 2ms/step - loss: 6.9488 - mae: 6.9488
Epoch 89/100
1/1 [==============================] - 0s 3ms/step - loss: 6.9431 - mae: 6.9431
Epoch 90/100
1/1 [==============================] - 0s 3ms/step - loss: 6.9375 - mae: 6.9375
Epoch 91/100
1/1 [==============================] - 0s 2ms/step - loss: 6.9319 - mae: 6.9319
Epoch 92/100
1/1 [==============================] - 0s 2ms/step - loss: 6.9263 - mae: 6.9263
Epoch 93/100
1/1 [==============================] - 0s 2ms/step - loss: 6.9206 - mae: 6.9206
Epoch 94/100
1/1 [==============================] - 0s 2ms/step - loss: 6.9150 - mae: 6.9150
Epoch 95/100
1/1 [==============================] - 0s 3ms/step - loss: 6.9094 - mae: 6.9094
Epoch 96/100
1/1 [==============================] - 0s 3ms/step - loss: 6.9038 - mae: 6.9038
Epoch 97/100
1/1 [==============================] - 0s 5ms/step - loss: 6.8981 - mae: 6.8981
Epoch 98/100
1/1 [==============================] - 0s 3ms/step - loss: 6.8925 - mae: 6.8925
Epoch 99/100
1/1 [==============================] - 0s 2ms/step - loss: 6.8869 - mae: 6.8869
Epoch 100/100
1/1 [==============================] - 0s 2ms/step - loss: 6.8813 - mae: 6.8813
```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa24314c6d8>
```

You might've noticed the loss value decrease from before (and keep decreasing as the number of epochs gets higher).

What do you think this means for when we make a prediction with our model?

How about we try predict on 17.0 again?

In [ ]:

```
# Remind ourselves of what X and y are
```

```
X, y
```

Out[ ]:

```
(<tf.Tensor: shape=(8,), dtype=float32, numpy=array([-7., -4., -1.,  2.,  5.,  8., 11., 1
4.], dtype=float32)>,
 <tf.Tensor: shape=(8,), dtype=float32, numpy=array([ 3.,  6.,  9., 12., 15., 18., 21., 2
4.], dtype=float32)>)
```

In [ ]:

```
# Try and predict what y would be if X was 17.0
model.predict([17.0]) # the right answer is 27.0 (y = X + 10)
```

Out[ ]:

```
array([[30.158512]], dtype=float32)
```

Much better!

We got closer this time. But we could still be better.

Now we've trained a model, how could we evaluate it?

Evaluating a model

A typical workflow you'll go through when building neural networks is:

```
Build a model -> evaluate it -> build (tweak) a model -> evaulate it -> build (twe
ak) a model -> evaluate it...
```

The tweaking comes from maybe not building a model from scratch but adjusting an existing one.

Visualize, visualize, visualize

When it comes to evaluation, you'll want to remember the words: "visualize, visualize, visualize."

This is because you're probably better looking at something (doing) than you are thinking about something.

It's a good idea to visualize:

- **The data - what data are you working with? What does it look like?**
- **The model itself - what does the architecture look like? What are the different shapes?**
- **The training of a model - how does a model perform while it learns?**
- **The predictions of a model - how do the predictions of a model line up against the ground truth (the original labels)?**

Let's start by visualizing the model.

But first, we'll create a little bit of a bigger dataset and a new model we can use (it'll be the same as before, but the more practice the better).

In [ ]:

```
# Make a bigger dataset
X = np.arange(-100, 100, 4)
X
```

Out[ ]:

```
array([-100,  -96,  -92,  -88,  -84,  -80,  -76,  -72,  -68,  -64,  -60,
        -56,  -52,  -48,  -44,  -40,  -36,  -32,  -28,  -24,  -20,  -16,
        -12,   -8,   -4,    0,    4,    8,   12,   16,   20,   24,   28,
         32,   36,   40,   44,   48,   52,   56,   60,   64,   68,   72,
         76,   80,   84,   88,   92,   96])
```

In [ ]:

```
# Make labels for the dataset (adhering to the same pattern as before)
```

```
y = np.arange(-90, 110, 4)
y
```

Out[ ]:

```
array([-90, -86, -82, -78, -74, -70, -66, -62, -58, -54, -50, -46, -42,
       -38, -34, -30, -26, -22, -18, -14, -10,  -6,  -2,   2,   6,  10,
        14,  18,  22,  26,  30,  34,  38,  42,  46,  50,  54,  58,  62,
        66,  70,  74,  78,  82,  86,  90,  94,  98, 102, 106])
```

Since $y = X + 10$, we could make the labels like so:

In [ ]:

```
# Same result as above
y = X + 10
y
```

Out[ ]:

```
array([-90, -86, -82, -78, -74, -70, -66, -62, -58, -54, -50, -46, -42,
       -38, -34, -30, -26, -22, -18, -14, -10,  -6,  -2,   2,   6,  10,
        14,  18,  22,  26,  30,  34,  38,  42,  46,  50,  54,  58,  62,
        66,  70,  74,  78,  82,  86,  90,  94,  98, 102, 106])
```

Split data into training/test set

One of the other most common and important steps in a machine learning project is creating a training and test set (and when required, a validation set).

Each set serves a specific purpose:

- **Training set** - the model learns from this data, which is typically 70-80% of the total data available (like the course materials you study during the semester).
- **Validation set** - the model gets tuned on this data, which is typically 10-15% of the total data available (like the practice exam you take before the final exam).
- **Test set** - the model gets evaluated on this data to test what it has learned, it's typically 10-15% of the total data available (like the final exam you take at the end of the semester).

For now, we'll just use a training and test set, this means we'll have a dataset for our model to learn on as well as be evaluated on.

We can create them by splitting our `X` and `y` arrays.

> **Note:** When dealing with real-world data, this step is typically done right at the start of a project (the test set should always be kept separate from all other data). We want our model to learn on training data and then evaluate it on test data to get an indication of how well it **generalizes** to unseen examples.

In [ ]:

```
# Check how many samples we have
len(X)
```

Out[ ]:

```
50
```

In [ ]:

```
# Split data into train and test sets
X_train = X[:40] # first 40 examples (80% of data)
y_train = y[:40]
```

```
X_test = X[40:] # last 10 examples (20% of data)
y_test = y[40:]

len(X_train), len(X_test)
```

Out[ ]:

```
(40, 10)
```

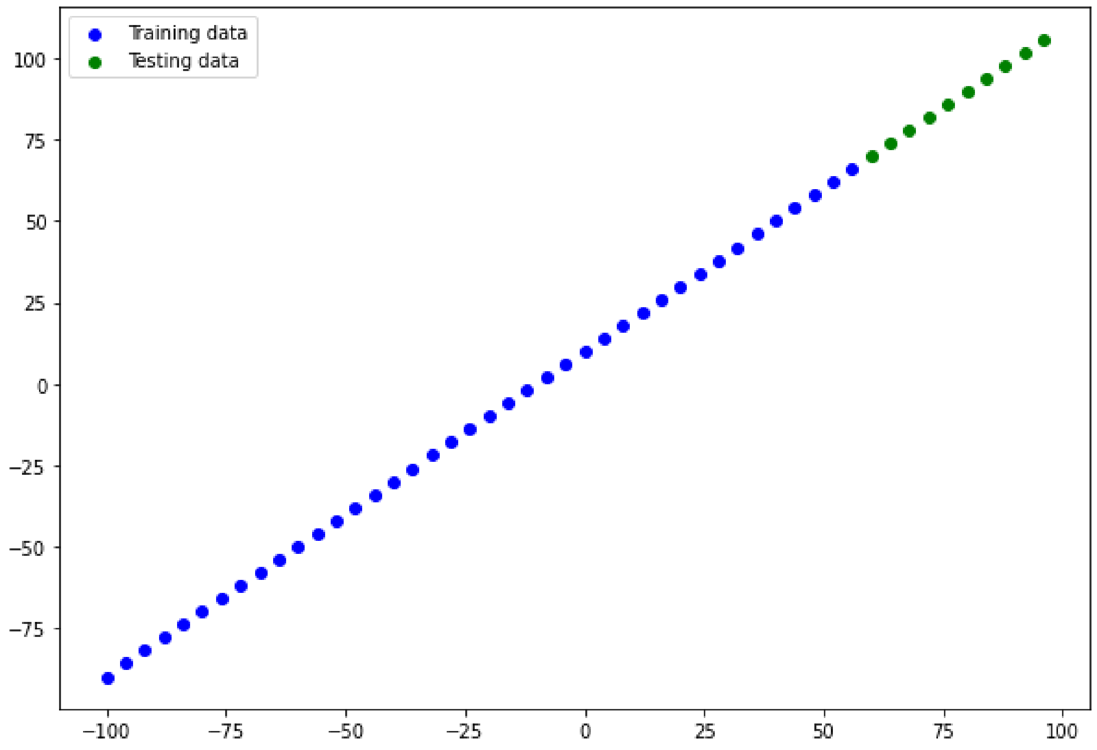
Visualizing the data

Now we've got our training and test data, it's a good idea to visualize it.

Let's plot it with some nice colours to differentiate what's what.

In [ ]:

```
plt.figure(figsize=(10, 7))
# Plot training data in blue
plt.scatter(X_train, y_train, c='b', label='Training data')
# Plot test data in green
plt.scatter(X_test, y_test, c='g', label='Testing data')
# Show the legend
plt.legend();
```



Beautiful! Any time you can visualize your data, your model, your anything, it's a good idea.

With this graph in mind, what we'll be trying to do is build a model which learns the pattern in the blue dots (`X_train`) to draw the green dots (`X_test`).

Time to build a model. We'll make the exact same one from before (the one we trained for longer).

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a model (same as above)
model = tf.keras.Sequential([
  tf.keras.layers.Dense(1)
])

# Compile model (same as above)
model.compile(loss=tf.keras.losses.mae,
              optimizer=tf.keras.optimizers.SGD(),
              metrics=["mae"])

# Fit model (same as above)
#model.fit(X_train, y_train, epochs=100) # commented out on purpose (not fitting it just
yet)
```

Visualizing the model

After you've built a model, you might want to take a look at it (especially if you haven't built many before).

You can take a look at the layers and shapes of your model by calling `summary()` on it.

> **🔑 Note: Visualizing a model is particularly helpful when you run into input and output shape mismatches.**

In [ ]:

```
# Doesn't work (model not fit/built)
model.summary()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-7d09d31d4e66> in <module>()
      1 # Doesn't work (model not fit/built)
----> 2 model.summary()

/usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/training.py in summ
ary(self, line_length, positions, print_fn)
   2349     """
   2350     if not self.built:
-> 2351       raise ValueError('This model has not yet been built. '
   2352                        'Build the model first by calling `build()` or calling '
   2353                        '`fit()` with some data, or specify '

ValueError: This model has not yet been built. Build the model first by calling `build()`
or calling `fit()` with some data, or specify an `input_shape` argument in the first laye
r(s) for automatic build.
```

Ahh, the cell above errors because we haven't fit our built our model.

We also haven't told it what input shape it should be expecting.

Remember above, how we discussed the input shape was just one number?

We can let our model know the input shape of our data using the `input_shape` parameter to the first layer (usually if `input_shape` isn't defined, Keras tries to figure it out automatically).

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a model (same as above)
model = tf.keras.Sequential([
  tf.keras.layers.Dense(1, input_shape=[1]) # define the input_shape to our model
])

# Compile model (same as above)
model.compile(loss=tf.keras.losses.mae,
              optimizer=tf.keras.optimizers.SGD(),
              metrics=["mae"])
```

In [ ]:

```
# This will work after specifying the input shape
```

```
model.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_7 (Dense)              (None, 1)                 2         
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

Calling `summary()` on our model shows us the layers it contains, the output shape and the number of parameters.

- **Total params** **- total number of parameters in the model.**
- **Trainable parameters** **- these are the parameters (patterns) the model can update as it trains.**
- **Non-trainable parameters** **- these parameters aren't updated during training (this is typical when you bring in the already learned patterns from other models during transfer learning).**

> ▯ **Resource:** **For a more in-depth overview of the trainable parameters within a layer, check out [MIT's introduction to deep learning video](). **
>
> ▯ **Exercise:** **Try playing around with the number of hidden units in the `Dense` layer (e.g. `Dense(2)`, `Dense(3)`). How does this change the Total/Trainable params? Investigate what's causing the change.**

For now, all you need to think about these parameters is that their learnable patterns in the data.

Let's fit our model to the training data.

In [ ]:

```
# Fit the model to the training data
model.fit(X_train, y_train, epochs=100, verbose=0) # verbose controls how much gets outp
ut
```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa238b42ac8>
```

In [ ]:

```
# Check the model summary
model.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_7 (Dense)              (None, 1)                 2         
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

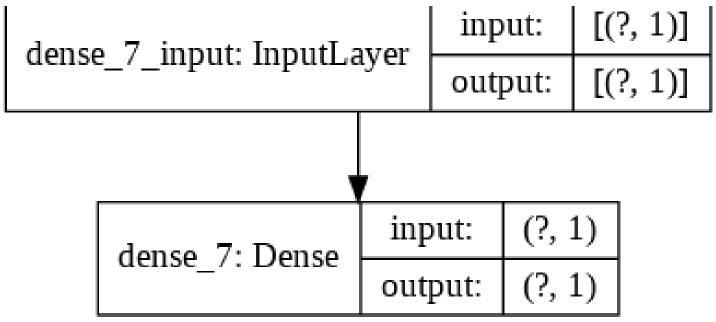
Alongside summary, you can also view a 2D plot of the model using [`plot_model()`]().

In [ ]:

```
from tensorflow.keras.utils import plot_model

plot_model(model, show_shapes=True)
```

Out[ ]:



In our case, the model we used only has an input and an output but visualizing more complicated models can be very helpful for debugging.

Visualizing the predictions

Now we've got a trained model, let's visualize some predictions.

To visualize predictions, it's always a good idea to plot them against the ground truth labels.

Often you'll see this in the form of `y_test` vs. `y_pred` (ground truth vs. predictions).

First, we'll make some predictions on the test data (`X_test`), remember the model has never seen the test data.

In [ ]:

```
# Make predictions
y_preds = model.predict(X_test)
```

In [ ]:

```
# View the predictions
y_preds
```

Out[ ]:

```
array([[53.57109 ],
       [57.05633 ],
       [60.541573],
       [64.02681 ],
       [67.512054],
       [70.99729 ],
       [74.48254 ],
       [77.96777 ],
       [81.45301 ],
       [84.938255]], dtype=float32)
```

Okay, we get a list of numbers but how do these compare to the ground truth labels?

Let's build a plotting function to find out.

> 🔑 **Note:** If you think you're going to be visualizing something a lot, it's a good idea to functionize it so you can use it later.

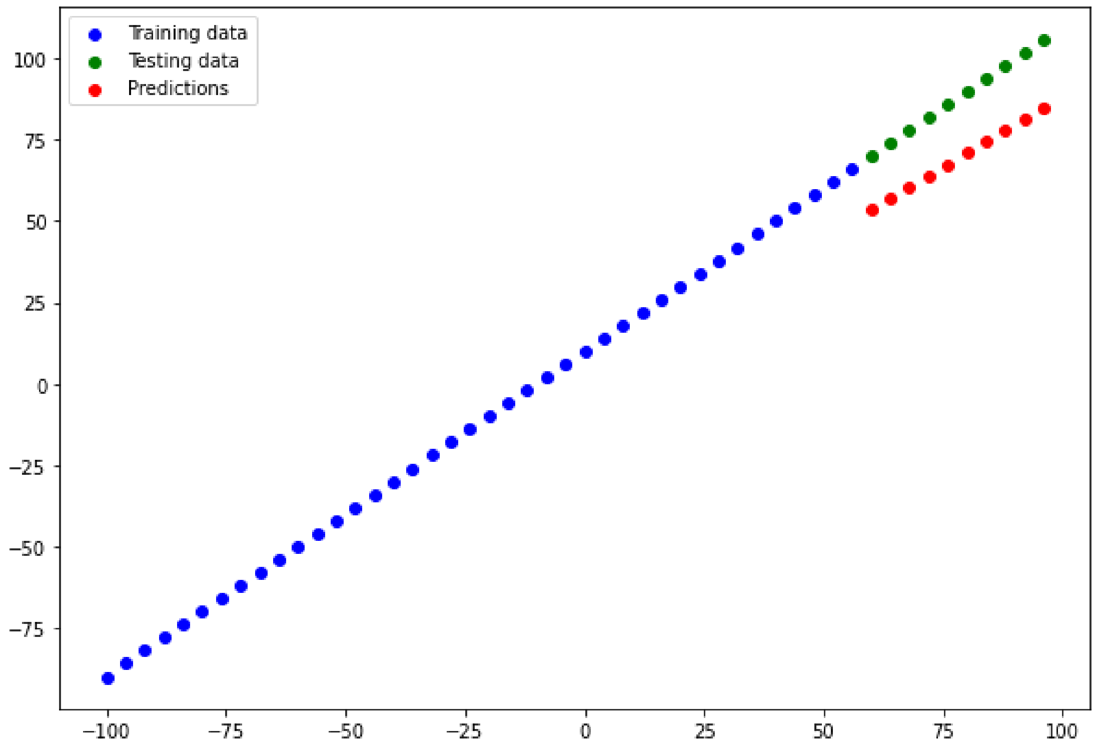
In [ ]:

```
def plot_predictions(train_data=X_train,
                     train_labels=y_train,
                     test_data=X_test,
                     test_labels=y_test,
                     predictions=y_preds):
  """
  Plots training data, test data and compares predictions.
  """
  plt.figure(figsize=(10, 7))
```

```
    # Plot training data in blue
    plt.scatter(train_data, train_labels, c="b", label="Training data")
    # Plot test data in green
    plt.scatter(test_data, test_labels, c="g", label="Testing data")
    # Plot the predictions in red (predictions were made on the test data)
    plt.scatter(test_data, predictions, c="r", label="Predictions")
    # Show the legend
    plt.legend();
```

In [ ]:

```
plot_predictions(train_data=X_train,
                 train_labels=y_train,
                 test_data=X_test,
                 test_labels=y_test,
                 predictions=y_preds)
```



From the plot we can see our predictions aren't totally outlandish but they definitely aren't anything special either.

Evaluating predictions

Alongisde visualizations, evaulation metrics are your alternative best option for evaluating your model.

Depending on the problem you're working on, different models have different evaluation metrics.

Two of the main metrics used for regression problems are:

- **Mean absolute error (MAE) - the mean difference between each of the predictions.**
- **Mean squared error (MSE) - the squared mean difference between of the predictions (use if larger errors are more detrimental than smaller errors).**

The lower each of these values, the better.

You can also use `model.evaluate()` which will return the loss of the model as well as any metrics setup during the compile step.

In [ ]:

```
# Evaluate the model on the test set
model.evaluate(X_test, y_test)
```

```
1/1 [==============================] - 0s 1ms/step - loss: 18.7453 - mae: 18.7453
```

Out[ ]:

```
[18.74532699584961, 18.74532699584961]
```

In our case, since we used MAE for the loss function as well as MAE for the metrics, `model.evaulate()` returns them both.

TensorFlow also has built in functions for MSE and MAE.

For many evaluation functions, the premise is the same: compare predictions to the ground truth labels.

In [ ]:

```
# Calculate the mean absolute error
mae = tf.metrics.mean_absolute_error(y_true=y_test,
                                     y_pred=y_preds)
mae
```

Out[ ]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([34.42891 , 30.943668, 27.45843 , 23.97319 , 20.487946, 17.202168,
       14.510478, 12.419336, 11.018796, 10.212349], dtype=float32)>
```

Huh? That's strange, MAE should be a single output.

Instead, we get 10 values.

This is because our `y_test` and `y_preds` tensors are different shapes.

In [ ]:

```
# Check the test label tensor values
y_test
```

Out[ ]:

```
array([ 70,  74,  78,  82,  86,  90,  94,  98, 102, 106])
```

In [ ]:

```
# Check the predictions tensor values (notice the extra square brackets)
y_preds
```

Out[ ]:

```
array([[53.57109 ],
       [57.05633 ],
       [60.541573],
       [64.02681 ],
       [67.512054],
       [70.99729 ],
       [74.48254 ],
       [77.96777 ],
       [81.45301 ],
       [84.938255]], dtype=float32)
```

In [ ]:

```
# Check the tensor shapes
y_test.shape, y_preds.shape
```

Out[ ]:

```
((10,), (10, 1))
```

Remember how we discussed dealing with different input and output shapes is one the most common issues you'll come across, this is one of those times.

But not to worry.

We can fix it using `squeeze()`, it'll remove the the `1` dimension from our `y_preds` tensor, making it the same shape as `y_test`.

> **🔑 Note: If you're comparing two tensors, it's important to make sure they're the right shape(s) (you won't always have to manipulate the shapes, but always be on the look out, *many* errors are the result of mismatched tensors, especially mismatched input and output shapes).**

In [ ]:

```
# Shape before squeeze()
y_preds.shape
```

Out[ ]:

```
(10, 1)
```

In [ ]:

```
# Shape after squeeze()
y_preds.squeeze().shape
```

Out[ ]:

```
(10,)
```

In [ ]:

```
# What do they look like?
y_test, y_preds.squeeze()
```

Out[ ]:

```
(array([ 70,  74,  78,  82,  86,  90,  94,  98, 102, 106]),
 array([53.57109 , 57.05633 , 60.541573, 64.02681 , 67.512054, 70.99729 ,
        74.48254 , 77.96777 , 81.45301 , 84.938255], dtype=float32))
```

Okay, now we know how to make our `y_test` and `y_preds` tenors the same shape, let's use our evaluation metrics.

In [ ]:

```
# Calcuate the MAE
mae = tf.metrics.mean_absolute_error(y_true=y_test,
                                     y_pred=y_preds.squeeze()) # use squeeze() to make s
ame shape
mae
```

Out[ ]:

```
<tf.Tensor: shape=(), dtype=float32, numpy=18.745327>
```

In [ ]:

```
# Calculate the MSE
mse = tf.metrics.mean_squared_error(y_true=y_test,
                                    y_pred=y_preds.squeeze())
mse
```

Out[ ]:

```
<tf.Tensor: shape=(), dtype=float32, numpy=353.57336>
```

We can also calculate the MAE using pure TensorFlow functions.

In [ ]:

```
# Returns the same as tf.metrics.mean_absolute_error()
tf.reduce_mean(tf.abs(y_test-y_preds.squeeze()))
```

Out[ ]:

```
<tf.Tensor: shape=(), dtype=float64, numpy=18.745327377319335>
```

Again, it's a good idea to functionize anything you think you might use over again (or find yourself using over and over again).

Let's make functions for our evaluation metrics.

In [ ]:

```
def mae(y_test, y_pred):
  """
  Calculuates mean absolute error between y_test and y_preds.
  """
  return tf.metrics.mean_absolute_error(y_test,
                                        y_pred)

def mse(y_test, y_pred):
  """
  Calculates mean squared error between y_test and y_preds.
  """
  return tf.metrics.mean_squared_error(y_test,
                                       y_pred)
```

Running experiments to improve a model

After seeing the evaluation metrics and the predictions your model makes, it's likely you'll want to improve it.

Again, there are many different ways you can do this, but 3 of the main ones are:

1. **Get more data** - get more examples for your model to train on (more opportunities to learn patterns).
2. **Make your model larger (use a more complex model)** - this might come in the form of more layers or more hidden units in each layer.
3. **Train for longer** - give your model more of a chance to find the patterns in the data.

Since we created our dataset, we could easily make more data but this isn't always the case when you're working with real-world datasets.

So let's take a look at how we can improve our model using 2 and 3.

To do so, we'll build 3 models and compare their results:

1. `model_1` - same as original model, 1 layer, trained for 100 epochs.
2. `model_2` - 2 layers, trained for 100 epochs.
3. `model_3` - 2 layers, trained for 500 epochs.

Build `model_1`

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Replicate original model
model_1 = tf.keras.Sequential([
  tf.keras.layers.Dense(1)
])

# Compile the model
model_1.compile(loss=tf.keras.losses.mae,
                optimizer=tf.keras.optimizers.SGD(),
                metrics=['mae'])

# Fit the model
model_1.fit(X_train, y_train, epochs=100)
```

```
Epoch 1/100
```

```
2/2 [==============================] - 0s 2ms/step - loss: 15.9024 - mae: 15.9024
Epoch 2/100
2/2 [==============================] - 0s 2ms/step - loss: 11.2837 - mae: 11.2837
Epoch 3/100
2/2 [==============================] - 0s 2ms/step - loss: 11.1074 - mae: 11.1074
Epoch 4/100
2/2 [==============================] - 0s 2ms/step - loss: 9.2991 - mae: 9.2991
Epoch 5/100
2/2 [==============================] - 0s 2ms/step - loss: 10.1677 - mae: 10.1677
Epoch 6/100
2/2 [==============================] - 0s 1ms/step - loss: 9.4303 - mae: 9.4303
Epoch 7/100
2/2 [==============================] - 0s 1ms/step - loss: 8.5704 - mae: 8.5704
Epoch 8/100
2/2 [==============================] - 0s 2ms/step - loss: 9.0442 - mae: 9.0442
Epoch 9/100
2/2 [==============================] - 0s 2ms/step - loss: 18.7517 - mae: 18.7517
Epoch 10/100
2/2 [==============================] - 0s 1ms/step - loss: 10.1142 - mae: 10.1142
Epoch 11/100
2/2 [==============================] - 0s 2ms/step - loss: 8.3980 - mae: 8.3980
Epoch 12/100
2/2 [==============================] - 0s 2ms/step - loss: 10.6639 - mae: 10.6639
Epoch 13/100
2/2 [==============================] - 0s 1ms/step - loss: 9.7977 - mae: 9.7977
Epoch 14/100
2/2 [==============================] - 0s 2ms/step - loss: 16.0103 - mae: 16.0103
Epoch 15/100
2/2 [==============================] - 0s 2ms/step - loss: 11.4068 - mae: 11.4068
Epoch 16/100
2/2 [==============================] - 0s 2ms/step - loss: 8.5393 - mae: 8.5393
Epoch 17/100
2/2 [==============================] - 0s 2ms/step - loss: 13.6348 - mae: 13.6348
Epoch 18/100
2/2 [==============================] - 0s 2ms/step - loss: 11.4629 - mae: 11.4629
Epoch 19/100
2/2 [==============================] - 0s 2ms/step - loss: 17.9148 - mae: 17.9148
Epoch 20/100
2/2 [==============================] - 0s 2ms/step - loss: 15.0494 - mae: 15.0494
Epoch 21/100
2/2 [==============================] - 0s 2ms/step - loss: 11.0216 - mae: 11.0216
Epoch 22/100
2/2 [==============================] - 0s 2ms/step - loss: 8.1558 - mae: 8.1558
Epoch 23/100
2/2 [==============================] - 0s 2ms/step - loss: 9.5138 - mae: 9.5138
Epoch 24/100
2/2 [==============================] - 0s 2ms/step - loss: 7.6617 - mae: 7.6617
Epoch 25/100
2/2 [==============================] - 0s 3ms/step - loss: 13.1859 - mae: 13.1859
Epoch 26/100
2/2 [==============================] - 0s 3ms/step - loss: 16.4211 - mae: 16.4211
Epoch 27/100
2/2 [==============================] - 0s 2ms/step - loss: 13.1660 - mae: 13.1660
Epoch 28/100
2/2 [==============================] - 0s 2ms/step - loss: 14.2559 - mae: 14.2559
Epoch 29/100
2/2 [==============================] - 0s 2ms/step - loss: 10.0670 - mae: 10.0670
Epoch 30/100
2/2 [==============================] - 0s 2ms/step - loss: 16.3409 - mae: 16.3409
Epoch 31/100
2/2 [==============================] - 0s 1ms/step - loss: 23.6444 - mae: 23.6444
Epoch 32/100
2/2 [==============================] - 0s 2ms/step - loss: 7.6215 - mae: 7.6215
Epoch 33/100
2/2 [==============================] - 0s 2ms/step - loss: 9.3221 - mae: 9.3221
Epoch 34/100
2/2 [==============================] - 0s 2ms/step - loss: 13.7313 - mae: 13.7313
Epoch 35/100
2/2 [==============================] - 0s 2ms/step - loss: 11.1276 - mae: 11.1276
Epoch 36/100
2/2 [==============================] - 0s 2ms/step - loss: 13.3222 - mae: 13.3222
Epoch 37/100
```

```
2/2 [==============================] - 0s 1ms/step - loss: 9.4763 - mae: 9.4763
Epoch 38/100
2/2 [==============================] - 0s 2ms/step - loss: 10.1381 - mae: 10.1381
Epoch 39/100
2/2 [==============================] - 0s 2ms/step - loss: 10.1793 - mae: 10.1793
Epoch 40/100
2/2 [==============================] - 0s 2ms/step - loss: 10.9137 - mae: 10.9137
Epoch 41/100
2/2 [==============================] - 0s 2ms/step - loss: 7.9063 - mae: 7.9063
Epoch 42/100
2/2 [==============================] - 0s 2ms/step - loss: 10.0914 - mae: 10.0914
Epoch 43/100
2/2 [==============================] - 0s 2ms/step - loss: 8.7006 - mae: 8.7006
Epoch 44/100
2/2 [==============================] - 0s 2ms/step - loss: 12.2047 - mae: 12.2047
Epoch 45/100
2/2 [==============================] - 0s 2ms/step - loss: 13.7970 - mae: 13.7970
Epoch 46/100
2/2 [==============================] - 0s 2ms/step - loss: 8.4687 - mae: 8.4687
Epoch 47/100
2/2 [==============================] - 0s 2ms/step - loss: 9.1330 - mae: 9.1330
Epoch 48/100
2/2 [==============================] - 0s 2ms/step - loss: 10.6190 - mae: 10.6190
Epoch 49/100
2/2 [==============================] - 0s 2ms/step - loss: 7.7503 - mae: 7.7503
Epoch 50/100
2/2 [==============================] - 0s 2ms/step - loss: 9.5407 - mae: 9.5407
Epoch 51/100
2/2 [==============================] - 0s 2ms/step - loss: 9.1584 - mae: 9.1584
Epoch 52/100
2/2 [==============================] - 0s 2ms/step - loss: 16.3630 - mae: 16.3630
Epoch 53/100
2/2 [==============================] - 0s 2ms/step - loss: 14.1299 - mae: 14.1299
Epoch 54/100
2/2 [==============================] - 0s 2ms/step - loss: 21.1247 - mae: 21.1247
Epoch 55/100
2/2 [==============================] - 0s 2ms/step - loss: 16.3961 - mae: 16.3961
Epoch 56/100
2/2 [==============================] - 0s 2ms/step - loss: 9.9806 - mae: 9.9806
Epoch 57/100
2/2 [==============================] - 0s 2ms/step - loss: 9.9606 - mae: 9.9606
Epoch 58/100
2/2 [==============================] - 0s 8ms/step - loss: 9.2209 - mae: 9.2209
Epoch 59/100
2/2 [==============================] - 0s 2ms/step - loss: 8.4239 - mae: 8.4239
Epoch 60/100
2/2 [==============================] - 0s 2ms/step - loss: 9.4869 - mae: 9.4869
Epoch 61/100
2/2 [==============================] - 0s 2ms/step - loss: 11.4355 - mae: 11.4355
Epoch 62/100
2/2 [==============================] - 0s 2ms/step - loss: 11.6887 - mae: 11.6887
Epoch 63/100
2/2 [==============================] - 0s 2ms/step - loss: 7.0838 - mae: 7.0838
Epoch 64/100
2/2 [==============================] - 0s 2ms/step - loss: 16.9675 - mae: 16.9675
Epoch 65/100
2/2 [==============================] - 0s 1ms/step - loss: 12.4599 - mae: 12.4599
Epoch 66/100
2/2 [==============================] - 0s 2ms/step - loss: 13.0184 - mae: 13.0184
Epoch 67/100
2/2 [==============================] - 0s 2ms/step - loss: 8.0600 - mae: 8.0600
Epoch 68/100
2/2 [==============================] - 0s 2ms/step - loss: 10.1888 - mae: 10.1888
Epoch 69/100
2/2 [==============================] - 0s 2ms/step - loss: 12.3633 - mae: 12.3633
Epoch 70/100
2/2 [==============================] - 0s 2ms/step - loss: 9.0516 - mae: 9.0516
Epoch 71/100
2/2 [==============================] - 0s 2ms/step - loss: 10.0378 - mae: 10.0378
Epoch 72/100
2/2 [==============================] - 0s 1ms/step - loss: 10.0516 - mae: 10.0516
Epoch 73/100
```

```
2/2 [==============================] - 0s 1ms/step - loss: 12.6151 - mae: 12.6151
Epoch 74/100
2/2 [==============================] - 0s 1ms/step - loss: 10.3819 - mae: 10.3819
Epoch 75/100
2/2 [==============================] - 0s 1ms/step - loss: 9.7229 - mae: 9.7229
Epoch 76/100
2/2 [==============================] - 0s 2ms/step - loss: 11.2252 - mae: 11.2252
Epoch 77/100
2/2 [==============================] - 0s 2ms/step - loss: 8.3642 - mae: 8.3642
Epoch 78/100
2/2 [==============================] - 0s 2ms/step - loss: 9.1274 - mae: 9.1274
Epoch 79/100
2/2 [==============================] - 0s 2ms/step - loss: 19.5039 - mae: 19.5039
Epoch 80/100
2/2 [==============================] - 0s 1ms/step - loss: 14.8945 - mae: 14.8945
Epoch 81/100
2/2 [==============================] - 0s 1ms/step - loss: 9.0034 - mae: 9.0034
Epoch 82/100
2/2 [==============================] - 0s 2ms/step - loss: 13.0206 - mae: 13.0206
Epoch 83/100
2/2 [==============================] - 0s 2ms/step - loss: 7.9299 - mae: 7.9299
Epoch 84/100
2/2 [==============================] - 0s 2ms/step - loss: 7.6872 - mae: 7.6872
Epoch 85/100
2/2 [==============================] - 0s 2ms/step - loss: 10.0328 - mae: 10.0328
Epoch 86/100
2/2 [==============================] - 0s 2ms/step - loss: 9.2433 - mae: 9.2433
Epoch 87/100
2/2 [==============================] - 0s 2ms/step - loss: 12.0209 - mae: 12.0209
Epoch 88/100
2/2 [==============================] - 0s 1ms/step - loss: 10.6389 - mae: 10.6389
Epoch 89/100
2/2 [==============================] - 0s 2ms/step - loss: 7.2667 - mae: 7.2667
Epoch 90/100
2/2 [==============================] - 0s 2ms/step - loss: 12.7786 - mae: 12.7786
Epoch 91/100
2/2 [==============================] - 0s 1ms/step - loss: 7.3481 - mae: 7.3481
Epoch 92/100
2/2 [==============================] - 0s 2ms/step - loss: 7.7175 - mae: 7.7175
Epoch 93/100
2/2 [==============================] - 0s 1ms/step - loss: 7.1263 - mae: 7.1263
Epoch 94/100
2/2 [==============================] - 0s 3ms/step - loss: 12.6190 - mae: 12.6190
Epoch 95/100
2/2 [==============================] - 0s 2ms/step - loss: 10.0912 - mae: 10.0912
Epoch 96/100
2/2 [==============================] - 0s 1ms/step - loss: 9.3558 - mae: 9.3558
Epoch 97/100
2/2 [==============================] - 0s 1ms/step - loss: 12.6834 - mae: 12.6834
Epoch 98/100
2/2 [==============================] - 0s 1ms/step - loss: 8.6762 - mae: 8.6762
Epoch 99/100
2/2 [==============================] - 0s 2ms/step - loss: 9.4693 - mae: 9.4693
Epoch 100/100
2/2 [==============================] - 0s 1ms/step - loss: 8.7067 - mae: 8.7067
```

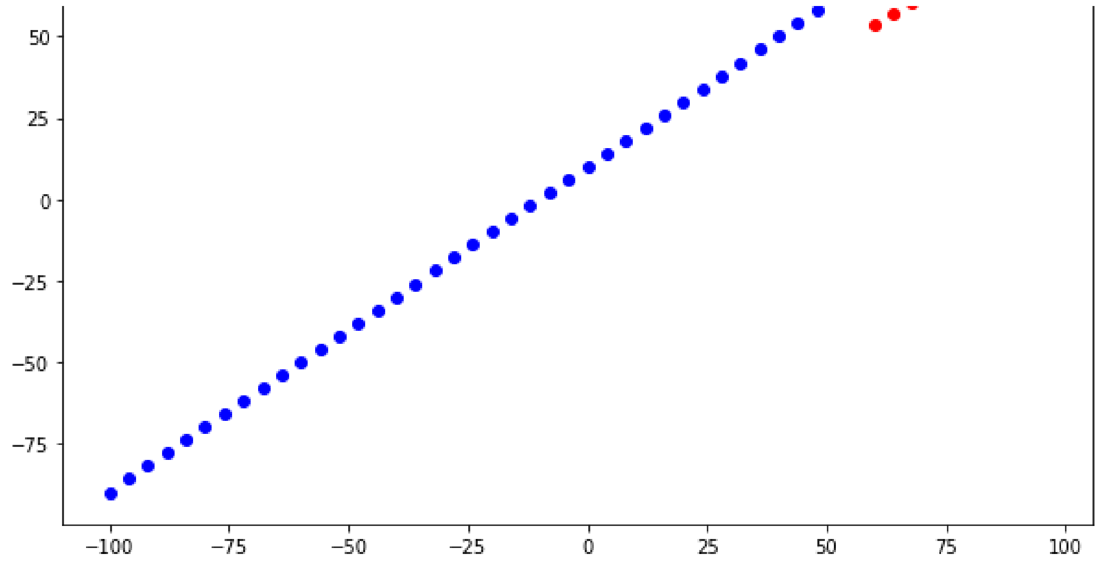
Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa2388fec50>
```

In [ ]:

```
# Make and plot predictions for model_1
y_preds_1 = model_1.predict(X_test)
plot_predictions(predictions=y_preds_1)
```

Training data
Testing data
Predictions
100
75



In [ ]:

```
# Calculate model_1 metrics
mae_1 = mae(y_test, y_preds_1.squeeze()).numpy()
mse_1 = mse(y_test, y_preds_1.squeeze()).numpy()
mae_1, mse_1
```

Out[ ]:

```
(18.745327, 353.57336)
```

Build `model_2`

This time we'll add an extra dense layer (so now our model will have 2 layers) whilst keeping everything else the same.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Replicate model_1 and add an extra layer
model_2 = tf.keras.Sequential([
  tf.keras.layers.Dense(1),
  tf.keras.layers.Dense(1) # add a second layer
])

# Compile the model
model_2.compile(loss=tf.keras.losses.mae,
                optimizer=tf.keras.optimizers.SGD(),
                metrics=['mae'])

# Fit the model
model_2.fit(X_train, y_train, epochs=100, verbose=0) # set verbose to 0 for less output
```

Out[ ]:

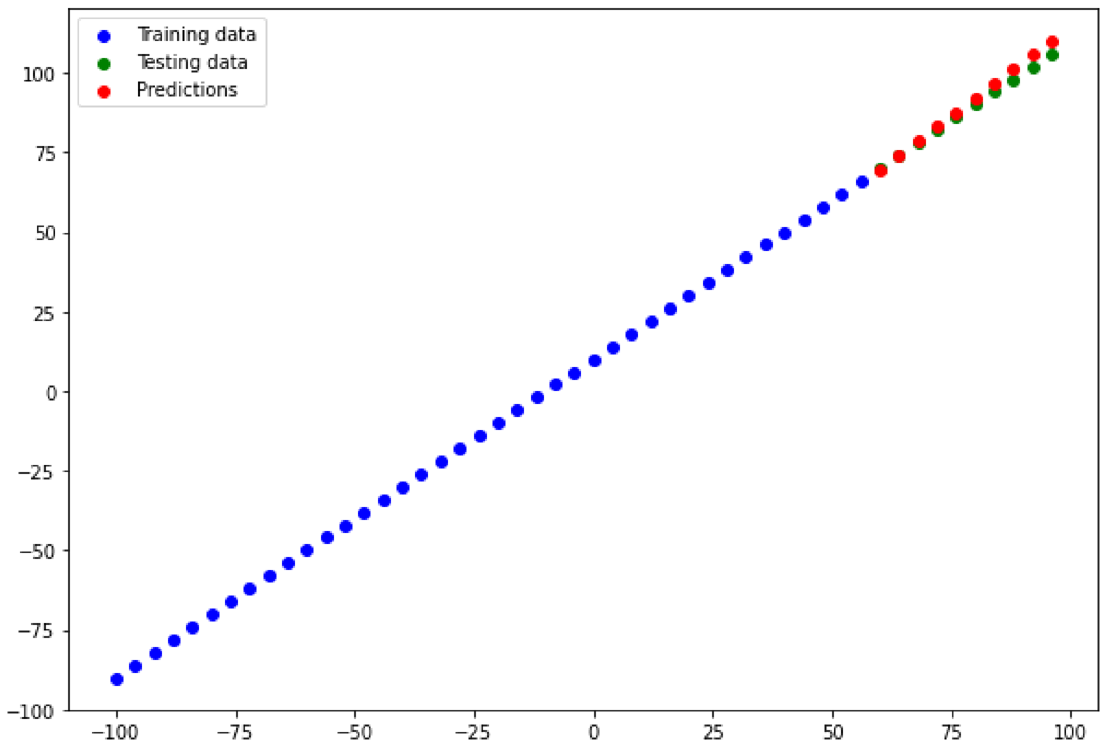
```
<tensorflow.python.keras.callbacks.History at 0x7fa23800bf98>
```

In [ ]:

```
# Make and plot predictions for model_2
y_preds_2 = model_2.predict(X_test)
plot_predictions(predictions=y_preds_2)
```

```
WARNING:tensorflow:5 out of the last 5 calls to <function Model.make_predict_function.<lo
cals>.predict_function at 0x7fa239dc71e0> triggered tf.function retracing. Tracing is exp
ensive and the excessive number of tracings could be due to (1) creating @tf.function rep
eatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the loop. For (2)
, @tf.function has experimental_relax_shapes=True option that relaxes argument shapes tha
t can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/tu
```

```
torials/customization/performance#python_or_tensor_args and https://www.tensorflow.org/ap
i_docs/python/tf/function for  more details.
```



Woah, that's looking better already! And all it took was an extra layer.

In [ ]:

```
# Calculate model_2 metrics
mae_2 = mae(y_test, y_preds_2.squeeze()).numpy()
mse_2 = mse(y_test, y_preds_2.squeeze()).numpy()
mae_2, mse_2
```

Out[ ]:

```
(1.9098114, 5.459232)
```

Build `model_3`

For our 3rd model, we'll keep everything the same as `model_2` **except this time we'll train for longer (500 epochs instead of 100).**

This will give our model more of a chance to learn the patterns in the data.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Replicate model_2
model_3 = tf.keras.Sequential([
  tf.keras.layers.Dense(1),
  tf.keras.layers.Dense(1)
])

# Compile the model
model_3.compile(loss=tf.keras.losses.mae,
                optimizer=tf.keras.optimizers.SGD(),
                metrics=['mae'])

# Fit the model (this time for 500 epochs, not 100)
model_3.fit(X_train, y_train, epochs=500, verbose=0) # set verbose to 0 for less output
```

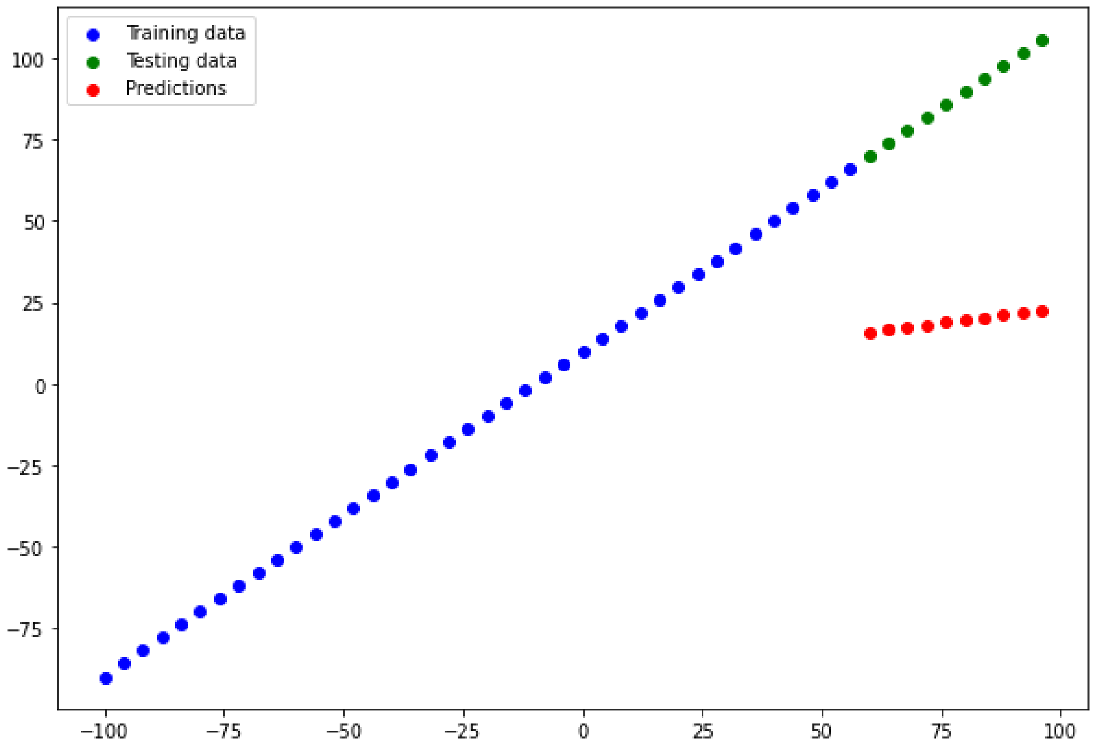
Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa239ce6cf8>
```

In [ ]:

```
# Make and plot predictions for model_3
y_preds_3 = model_3.predict(X_test)
plot_predictions(predictions=y_preds_3)
```

```
WARNING:tensorflow:6 out of the last 6 calls to <function Model.make_predict_function.<lo
cals>.predict_function at 0x7fa239c411e0> triggered tf.function retracing. Tracing is exp
ensive and the excessive number of tracings could be due to (1) creating @tf.function rep
eatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the loop. For (2)
, @tf.function has experimental_relax_shapes=True option that relaxes argument shapes tha
t can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/tu
torials/customization/performance#python_or_tensor_args and https://www.tensorflow.org/ap
i_docs/python/tf/function for  more details.
```



Strange, we trained for longer but our model performed worse?

As it turns out, our model might've trained too long and has thus resulted in worse results (we'll see ways to prevent training for too long later on).

In [ ]:

```
# Calculate model_3 metrics
mae_3 = mae(y_test, y_preds_3.squeeze()).numpy()
mse_3 = mse(y_test, y_preds_3.squeeze()).numpy()
mae_3, mse_3
```

Out[ ]:

```
(68.68786, 4804.4717)
```

Comparing results

Now we've got results for 3 similar but slightly different results, let's compare them.

In [ ]:

```
model_results = [["model_1", mae_1, mse_1],
                 ["model_2", mae_2, mse_2],
                 ["model_3", mae_3, mae_3]]
```

In [ ]:

```
import pandas as pd
all_results = pd.DataFrame(model_results, columns=["model", "mae", "mse"])
all_results
```

Out[ ]:

	model	mae	mse
0	model_1	18.745327	353.573364
1	model_2	1.909811	5.459232
2	model_3	68.687859	68.687859

From our experiments, it looks like `model_2` performed the best.

And now, you might be thinking, "wow, comparing models is tedious..." and it definitely can be, we've only compared 3 models here.

But this is part of what machine learning modelling is about, trying many different combinations of models and seeing which performs best.

Each model you build is a small experiment.

> **Note:** One of your main goals should be to minimize the time between your experiments. The more experiments you do, the more things you'll figure out which don't work and in turn, get closer to figuring out what does work. Remember the machine learning practitioner's motto: "experiment, experiment, experiment".

Another thing you'll also find is what you thought may work (such as training a model for longer) may not always work and the exact opposite is also often the case.

Tracking your experiments

One really good habit to get into is tracking your modelling experiments to see which perform better than others.

We've done a simple version of this above (keeping the results in different variables).

> **Resource:** But as you build more models, you'll want to look into using tools such as:
>
> - TensorBoard - a component of the TensorFlow library to help track modelling experiments (we'll see this later).
> - Weights & Biases - a tool for tracking all kinds of machine learning experiments (the good news for Weights & Biases is it plugs into TensorBoard).

Saving a model

Once you've trained a model and found one which performs to your liking, you'll probably want to save it for use elsewhere (like a web application or mobile device).

You can save a TensorFlow/Keras model using `model.save()`.

There are two ways to save a model in TensorFlow:

1. The SavedModel format (default).
2. The HDF5 format.

The main difference between the two is the SavedModel is automatically able to save custom objects (such as special layers) without additional modifications when loading the model back in.

Which one should you use?

It depends on your situation but the SavedModel format will suffice most of the time.

Both methods use the same method call.

In [ ]:

```
# Save a model using the SavedModel format
model_2.save('best_model_SavedModel_format')
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/training
/tracking/tracking.py:111: Model.state_updates (from tensorflow.python.keras.engine.train
ing) is deprecated and will be removed in a future version.
Instructions for updating:
This property should not be used in TensorFlow 2.0, as updates are applied automatically.
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/training
/tracking/tracking.py:111: Layer.updates (from tensorflow.python.keras.engine.base_layer)
is deprecated and will be removed in a future version.
Instructions for updating:
This property should not be used in TensorFlow 2.0, as updates are applied automatically.
INFO:tensorflow:Assets written to: best_model_SavedModel_format/assets
```

In [ ]:

```
# Check it out - outputs a protobuf binary file (.pb) as well as other files
!ls best_model_SavedModel_format
```

```
assets saved_model.pb variables
```

Now let's save the model in the HDF5 format, we'll use the same method but with a different filename.

In [ ]:

```
# Save a model using the HDF5 format
model_2.save("best_model_HDF5_format.h5") # note the addition of '.h5' on the end
```

In [ ]:

```
# Check it out
!ls best_model_HDF5_format.h5
```

```
best_model_HDF5_format.h5
```

Loading a model

We can load a saved model using the `load_model()` method.

Loading a model for the different formats (SavedModel and HDF5) is the same (as long as the pathnames to the particuluar formats are correct).

In [ ]:

```
# Load a model from the SavedModel format
loaded_saved_model = tf.keras.models.load_model("best_model_SavedModel_format")
loaded_saved_model.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_9 (Dense)              (None, 1)                 2         
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 2         
=================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
_________________________________________________________________
```

Now let's test it out.

In [ ]:

```
# Compare model_2 with the SavedModel version (should return True)
model_2_preds = model_2.predict(X_test)
saved_model_preds = loaded_saved_model.predict(X_test)
mae(y_test, saved_model_preds.squeeze()).numpy() == mae(y_test, model_2_preds.squeeze())
.numpy()
```

```
WARNING:tensorflow:7 out of the last 8 calls to <function Model.make_predict_function.<lo
cals>.predict_function at 0x7fa239a2df28> triggered tf.function retracing. Tracing is exp
ensive and the excessive number of tracings could be due to (1) creating @tf.function rep
eatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the loop. For (2)
, @tf.function has experimental_relax_shapes=True option that relaxes argument shapes tha
t can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/tu
torials/customization/performance#python_or_tensor_args and https://www.tensorflow.org/ap
i_docs/python/tf/function for  more details.
```

Out[ ]:

```
True
```

Loading in from the HDF5 is much the same.

In [ ]:

```
# Load a model from the HDF5 format
loaded_h5_model = tf.keras.models.load_model("best_model_HDF5_format.h5")
loaded_h5_model.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_9 (Dense)              (None, 1)                 2         
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 2         
=================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

```
# Compare model_2 with the loaded HDF5 version (should return True)
h5_model_preds = loaded_h5_model.predict(X_test)
mae(y_test, h5_model_preds.squeeze()).numpy() == mae(y_test, model_2_preds.squeeze()).num
py()
```

```
WARNING:tensorflow:8 out of the last 9 calls to <function Model.make_predict_function.<lo
cals>.predict_function at 0x7fa239732e18> triggered tf.function retracing. Tracing is exp
ensive and the excessive number of tracings could be due to (1) creating @tf.function rep
eatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the loop. For (2)
, @tf.function has experimental_relax_shapes=True option that relaxes argument shapes tha
t can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/tu
torials/customization/performance#python_or_tensor_args and https://www.tensorflow.org/ap
i_docs/python/tf/function for  more details.
```

Out[ ]:

```
True
```

Downloading a model (from Google Colab)

Say you wanted to get your model from Google Colab to your local machine, you can do one of the following things:

- **Right click on the file in the files pane and click 'download'.**

- Use the code below.

In [ ]:

```
# Download the model (or any file) from Google Colab
from google.colab import files
files.download("best_model_HDF5_format.h5")
```

A larger example

Alright, we've seen the fundamentals of building neural network regression models in TensorFlow.

Let's step it up a notch and build a model for a more feature rich datase.

More specifically we're going to try predict the cost of medical insurance for individuals based on a number of different parameters such as, `age`, `sex`, `bmi`, `children`, `smoking_status` and `residential_region`.

To do, we'll leverage the pubically available [Medical Cost dataset] available from Kaggle and [hosted on GitHub].

> 🔑 **Note:** When learning machine learning paradigms, you'll often go through a series of foundational techniques and then practice them by working with open-source datasets and examples. Just as we're doing now, learn foundations, put them to work with different problems. Every time you work on something new, it's a good idea to search for something like "problem X example with Python/TensorFlow" where you substitute X for your problem.

In [ ]:

```
# Import required libraries
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt
```

In [ ]:

```
# Read in the insurance dataset
insurance = pd.read_csv("https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-
datasets/master/insurance.csv")
```

In [ ]:

```
# Check out the insurance dataset
insurance.head()
```

Out[ ]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

We're going to have to turn the non-numerical columns into numbers (because a neural network can't handle non-numerical inputs).

To do so, we'll use the `get_dummies()` method in pandas.

It converts categorical variables (like the `sex`, `smoker` and `region` columns) into numerical variables using one-hot encoding.

In [ ]:

```
# Turn all categories into numbers
insurance_one_hot = pd.get_dummies(insurance)
insurance_one_hot.head()  # view the converted columns
```

Out[ ]:

	age	bmi	children	charges	sex_female	sex_male	smoker_no	smoker_yes	region_northeast	region_northwest	reg
0	19	27.900	0	16884.92400	1	0	0	1	0	0	
1	18	33.770	1	1725.55230	0	1	1	0	0	0	
2	28	33.000	3	4449.46200	0	1	1	0	0	0	
3	33	22.705	0	21984.47061	0	1	1	0	0	1	
4	32	28.880	0	3866.85520	0	1	1	0	0	1	

Now we'll split data into features (`X`) and labels (`y`).

In [ ]:

```
# Create X & y values
X = insurance_one_hot.drop("charges", axis=1)
y = insurance_one_hot["charges"]
```

In [ ]:

```
# View features
X.head()
```

Out[ ]:

	age	bmi	children	sex_female	sex_male	smoker_no	smoker_yes	region_northeast	region_northwest	region_southeas
0	19	27.900	0	1	0	0	1	0	0	0
1	18	33.770	1	0	1	1	0	0	0	1
2	28	33.000	3	0	1	1	0	0	0	1
3	33	22.705	0	0	1	1	0	0	1	0
4	32	28.880	0	0	1	1	0	0	1	0

And create training and test sets. We could do this manually, but to make it easier, we'll leverage the already available `train_test_split` function available from Scikit-Learn.

In [ ]:

```
# Create training and test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)  # set random state
for reproducible splits
```

Now we can build and fit a model (we'll make it the same as `model_2`).

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a new model (same as model_2)
```

```
insurance_model = tf.keras.Sequential([
  tf.keras.layers.Dense(1),
  tf.keras.layers.Dense(1)
])

# Compile the model
insurance_model.compile(loss=tf.keras.losses.mae,
                        optimizer=tf.keras.optimizers.SGD(),
                        metrics=['mae'])

# Fit the model
insurance_model.fit(X_train, y_train, epochs=100)
```

In [ ]:

```
# Check the results of the insurance model
insurance_model.evaluate(X_test, y_test)
```

```
9/9 [==============================] - 0s 1ms/step - loss: 8628.2363 - mae: 8628.2363
```

Out[ ]:

```
[8628.236328125, 8628.236328125]
```

Our model didn't perform very well, let's try a bigger model.

We'll try 3 things:

- **Increasing the number of layers (2 -> 3).**
- **Increasing the number of units in each layer (except for the output layer).**
- **Changing the optimizer (from SGD to Adam).**

Everything else will stay the same.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Add an extra layer and increase number of units
insurance_model_2 = tf.keras.Sequential([
  tf.keras.layers.Dense(100), # 100 units
  tf.keras.layers.Dense(10), # 10 units
  tf.keras.layers.Dense(1) # 1 unit (important for output layer)
])

# Compile the model
insurance_model_2.compile(loss=tf.keras.losses.mae,
                          optimizer=tf.keras.optimizers.Adam(), # Adam works but SGD doe
sn't
                          metrics=['mae'])

# Fit the model and save the history (we can plot this)
history = insurance_model_2.fit(X_train, y_train, epochs=100, verbose=0)
```

```
WARNING:tensorflow:Layer dense_37 is casting an input tensor from dtype float64 to the la
yer's dtype of float32, which is new behavior in TensorFlow 2.  The layer has dtype float
32 because its dtype defaults to floatx.

If you intended to run this layer in float32, you can safely ignore this warning. If in d
oubt, this warning is likely only an issue if you are porting a TensorFlow 1.X model to T
ensorFlow 2.

To change all layers to have dtype float64 by default, call `tf.keras.backend.set_floatx(
'float64')`. To change just this layer, pass dtype='float64' to the layer constructor. If
you are the author of this layer, you can disable autocasting by passing autocast=False t
o the base Layer constructor.
```

In [ ]:

```
# Evaluate our larger model
insurance_model_2.evaluate(X_test, y_test)
```

```
9/9 [==============================] - 0s 1ms/step - loss: 4924.3477 - mae: 4924.3477
```

Out[ ]:

```
[4924.34765625, 4924.34765625]
```

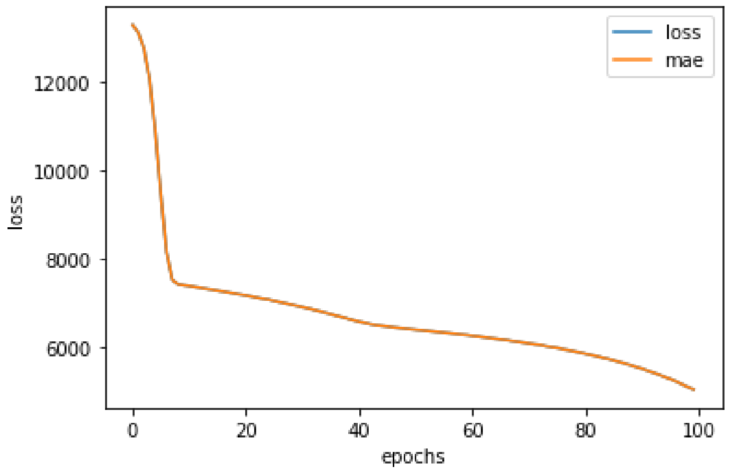
Much better! Using a larger model and the Adam optimizer results in almost half the error as the previous model.

> **🔑 Note:** For many problems, the [Adam optimizer](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam) is a great starting choice. See Andrei Karpathy's "Adam is safe" point from [*A Recipe for Training Neural Networks*](http://karpathy.github.io/2019/04/25/recipe/) for more.

Let's check out the loss curves of our model, we should see a downward trend.

In [ ]:

```
# Plot history (also known as a loss curve)
pd.DataFrame(history.history).plot()
plt.ylabel("loss")
plt.xlabel("epochs");
```



From this, it looks like our model's loss (and MAE) were both still decreasing (in our case, MAE and loss are the same, hence the lines in the plot overlap eachother).

What this tells us is the loss might go down if we try training it for longer.

> **🤔 Question:** How long should you train for?
>
> It depends on what problem you're working on. Sometimes training won't take very long, other times it'll take longer than you expect. A common method is to set your model training for a very long time (e.g. 1000's of epochs) but set it up with an [EarlyStopping callback](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping) so it stops automatically when it stops improving. We'll see this in another module.

Let's train the same model as above for a little longer. We can do this but calling fit on it again.

In [ ]:

```
# Try training for a little longer (100 more epochs)
history_2 = insurance_model_2.fit(X_train, y_train, epochs=100, verbose=0)
```

How did the extra training go?

In [ ]:

```
# Evaluate the model trained for 200 total epochs
insurance_model_2_loss, insurance_model_2_mae = insurance_model_2.evaluate(X_test, y_tes
t)
```

```
insurance_model_2_loss, insurance_model_2_mae
```

```
9/9 [==============================] - 0s 1ms/step - loss: 3494.7285 - mae: 3494.7285
```

Out[ ]:

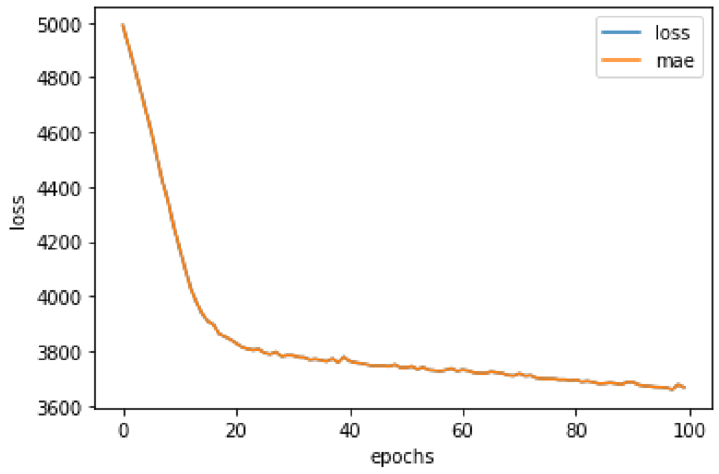
```
(3494.728515625, 3494.728515625)
```

Boom! Training for an extra 100 epochs we see about a 10% decrease in error.

How does the visual look?

In [ ]:

```
# Plot the model trained for 200 total epochs loss curves
pd.DataFrame(history_2.history).plot()
plt.ylabel("loss")
plt.xlabel("epochs"); # note: epochs will only show 100 since we overrid the history vari
able
```



Preprocessing data (normalization and standardization)

A common practice when working with neural networks is to make sure all of the data you pass to them is in the range 0 to 1.

This practice is called normalization (scaling all values from their original range to, e.g. between 0 and 100,000 to be between 0 and 1).

There is another process call standardization which converts all of your data to unit variance and 0 mean.

These two practices are often part of a preprocessing pipeline (a series of functions to prepare your data for use with neural networks).

Knowing this, some of the major steps you'll take to preprocess your data for a neural network include:

- **Turning all of your data to numbers (a neural network can't handle strings).**
- **Making sure your data is in the right shape (verifying input and output shapes).**
- **Feature scaling:**
  - **Normalizing data (making sure all values are between 0 and 1). This is done by subtracting the minimum value then dividing by the maximum value minus the minmum. This is also referred to as min-max scaling.**
  - **Standardization (making sure all values have a mean of 0 and a variance of 1). This is done by substracting the mean value from the target feature and then dividing it by the standard deviation.**
  - **Which one should you use?**
    - **With neural networks you'll tend to favour normalization as they tend to prefer values between 0 and 1 (you'll see this espcially with image processing), however, you'll often find a neural network can perform pretty well with minimal feature scaling.**

> **Resource:** For more on preprocessing data, I'd recommend reading the following resources:
>
> - Scikit-Learn's documentation on preprocessing data.

- Scikit-Learn's documentation on preprocessing data.
- Scale, Standardize or Normalize with Scikit-Learn by Jeff Hale .

We've already turned our data into numbers using `get_dummies()`, let's see how we'd normalize it as well.

In [ ]:

```
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf

# Read in the insurance dataset
insurance = pd.read_csv("https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-
datasets/master/insurance.csv")
```

In [ ]:

```
# Check out the data
insurance.head()
```

Out[ ]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

Now, just as before, we need to transform the non-numerical columns into numbers and this time we'll also be normalizing the numerical columns with different ranges (to make sure they're all between 0 and 1).

To do this, we're going to use a few classes from Scikit-Learn:

- `make_column_transformer` **- build a multi-step data preprocessing function for the folllowing trnasformations:**
  - `MinMaxScaler` **- make sure all numerical columns are normalized (between 0 and 1).**
  - `OneHotEncoder` **- one hot encode the non-numerical columns.**

Let's see them in action.

In [ ]:

```
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder

# Create column transformer (this will help us normalize/preprocess our data)
ct = make_column_transformer(
    (MinMaxScaler(), ["age", "bmi", "children"]), # get all values between 0 and 1
    (OneHotEncoder(handle_unknown="ignore"), ["sex", "smoker", "region"])
)

# Create X & y
X = insurance.drop("charges", axis=1)
y = insurance["charges"]

# Build our train and test sets (use random state to ensure same split as before)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)

# Fit column transformer on the training data only (doing so on test data would result in
data leakage)
ct.fit(X_train)

# Transform training and test data with normalization (MinMaxScalar) and one hot encoding
```

```
(OneHotEncoder)
X_train_normal = ct.transform(X_train)
X_test_normal = ct.transform(X_test)
```

Now we've normalized it and one-hot encoding it, what does our data look like now?

In [ ]:

```
# Non-normalized and non-one-hot encoded data example
X_train.loc[0]
```

Out[ ]:

```
age                19
sex            female
bmi              27.9
children            0
smoker            yes
region      southwest
Name: 0, dtype: object
```

In [ ]:

```
# Normalized and one-hot encoded example
X_train_normal[0]
```

Out[ ]:

```
array([0.60869565, 0.10734463, 0.4       , 1.        , 0.        ,
       1.        , 0.        , 0.        , 1.        , 0.        ,
       0.        ])
```

How about the shapes?

In [ ]:

```
# Notice the normalized/one-hot encoded shape is larger because of the extra columns
X_train_normal.shape, X_train.shape
```

Out[ ]:

```
((1070, 11), (1070, 6))
```

Our data is normalized and numerical, let's model it.

We'll use the same model as `insurance_model_2`.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Build the model (3 layers, 100, 10, 1 units)
insurance_model_3 = tf.keras.Sequential([
  tf.keras.layers.Dense(100),
  tf.keras.layers.Dense(10),
  tf.keras.layers.Dense(1)
])

# Compile the model
insurance_model_3.compile(loss=tf.keras.losses.mae,
                          optimizer=tf.keras.optimizers.Adam(),
                          metrics=['mae'])

# Fit the model for 200 epochs (same as insurance_model_2)
insurance_model_3.fit(X_train_normal, y_train, epochs=200, verbose=0)
```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa236315b00>
```

Let's evaluate the model on normalized test set.

In [ ]:

```
# Evaulate 3rd model
insurance_model_3_loss, insurance_model_3_mae = insurance_model_3.evaluate(X_test_normal,
y_test)
```

```
9/9 [==============================] - 0s 1ms/step - loss: 3171.7632 - mae: 3171.7632
```

And finally, let's compare the results from `insurance_model_2` (trained on non-normalized data) and `insurance_model_3` (trained on normalized data).

In [ ]:

```
# Compare modelling results from non-normalized data and normalized data
insurance_model_2_mae, insurance_model_3_mae
```

Out[ ]:

```
(3494.728515625, 3171.76318359375)
```

From this we can see normalizing the data results in 10% less error using the same model than not normalizing the data.

This is one of the main benefits of normalization: faster convergence time (a fancy way of saying, your model gets to better results faster).

`insurance_model_2` may have eventually achieved the same results as `insurance_model_3` if we left it training for longer.

Also, the results may change if we were to alter the architectures of the models, e.g. more hidden units per layer or more layers.

But since our main goal as neural network practioners is to decrease the time between experiments, anything that helps us get better results sooner is a plus.

▯ Exercises

We've a covered a whole lot pretty quickly.

So now it's time to have a play around with a few things and start to build up your initution.

I emphasise the words play around because that's very important. Try a few things out, run the code and see what happens.

1. **Create your own regression dataset (or make the one we created in "Create data to view and fit" bigger) and build fit a model to it.**
2. **Try building a neural network with 4 Dense layers and fitting it to your own regression dataset, how does it perform?**
3. **Try and improve the results we got on the insurance dataset, some things you might want to try include:**
   - **Building a larger model (how does one with 4 dense layers go?).**
   - **Increasing the number of units in each layer.**
   - **Lookup the documentation of Adam and find out what the first parameter is, what happens if you increase it by 10x?**
   - **What happens if you train for longer (say 300 epochs instead of 200)?**
4. **Import the Boston pricing dataset from TensorFlow `tf.keras.datasets` and model it.**

▯ Extra curriculum

If you're looking for extra materials relating to this notebook, I'd check out the following:

- **MIT introduction deep learning lecture 1 - gives a great overview of what's happening behind all of the code** we're running.

- Reading: 1-hour of Chapter 1 of Neural Networks and Deep Learning by Michael Nielson - a great in-depth and hands-on example of the intuition behind neural networks.

To practice your regression modelling with TensorFlow, I'd also encourage you to look through Lion Bridge's collection of datasets or Kaggle's datasets, find a regression dataset which sparks your interest and try to model.