01. Neural Network Regression with TensorFlow

There are many definitions for a [regression problem (https://en.wikipedia.org/wiki/Regression_analysis)](https://en.wikipedia.org/wiki/Regression_analysis) 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 outputting some kind of target number.

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

Hyperparameter	**Typical va**
Input layer shape	Same shape as number of features (e.g. 3 for # bedrooms, # bathroo # car spaces in housing price predicti
Hidden layer(s)	Problem specific, minimum = 1, maximum = unlimi
Neurons per hidden layer	Problem specific, generally 10 to
Output layer shape	Same shape as desired prediction shape (e.g. 1 for house pri
Hidden activation	Usually [ReLU (https://www.kaggle.com/dansbecker/rectified-line units-relu-in-deep-learning)](https://www.kaggle.com/dansbecker/rectified-line) (rectified linear un
Output activation	None, ReLU, logistic/t

Hyperparameter	Typical va
Loss function	[MSE (https://en.wikipedia.org/wiki/Mean_squared_error)](https://en.wikipedia.org/wiki/Mean_squared_error) (mean squ error) or [MAE (https://en.wikipedia.org/wiki/Mean_absolute_err](https://en.wikipedia.org/wiki/Mean_absolute_error) (mean absolute error)/Huber (combination of MAE/MSE) if outli
Optimizer	[(https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/S](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/S) (stochastic gradient descent), A [(https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Ad](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Ad)

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 (https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/)](https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/)

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.

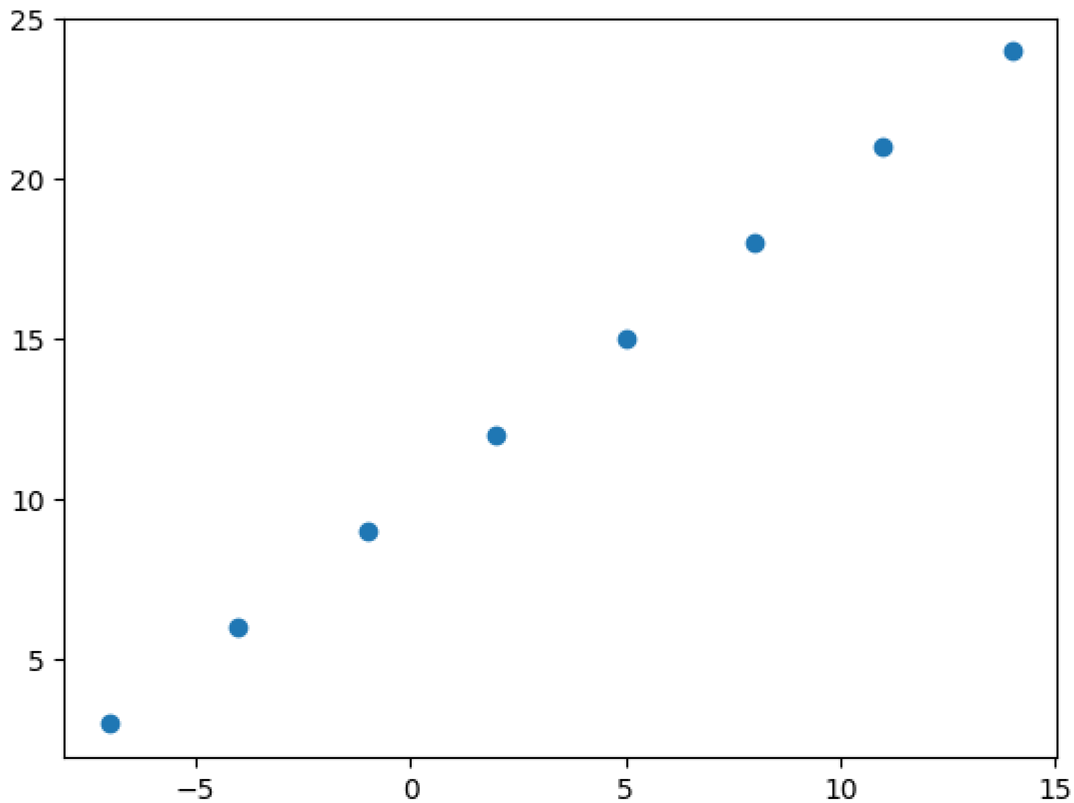
In [1]:
```
import tensorflow as tf
print(tf.__version__) # check the version (should be 2.x+)

import datetime
print(f"Notebook last run (end-to-end): {datetime.datetime.now()}")
```

```
2.12.0
Notebook last run (end-to-end): 2023-05-07 23:10:01.302908
```

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 [2]:
```
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 discovery 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 [3]:

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

Out[3]:

```
(<tf.Tensor: shape=(3,), dtype=string, numpy=array([b'bedroom', b'bath
room', b'garage'], dtype=object)>,
 <tf.Tensor: shape=(1,), dtype=int32, numpy=array([939700], dtype=int3
2)>)
```

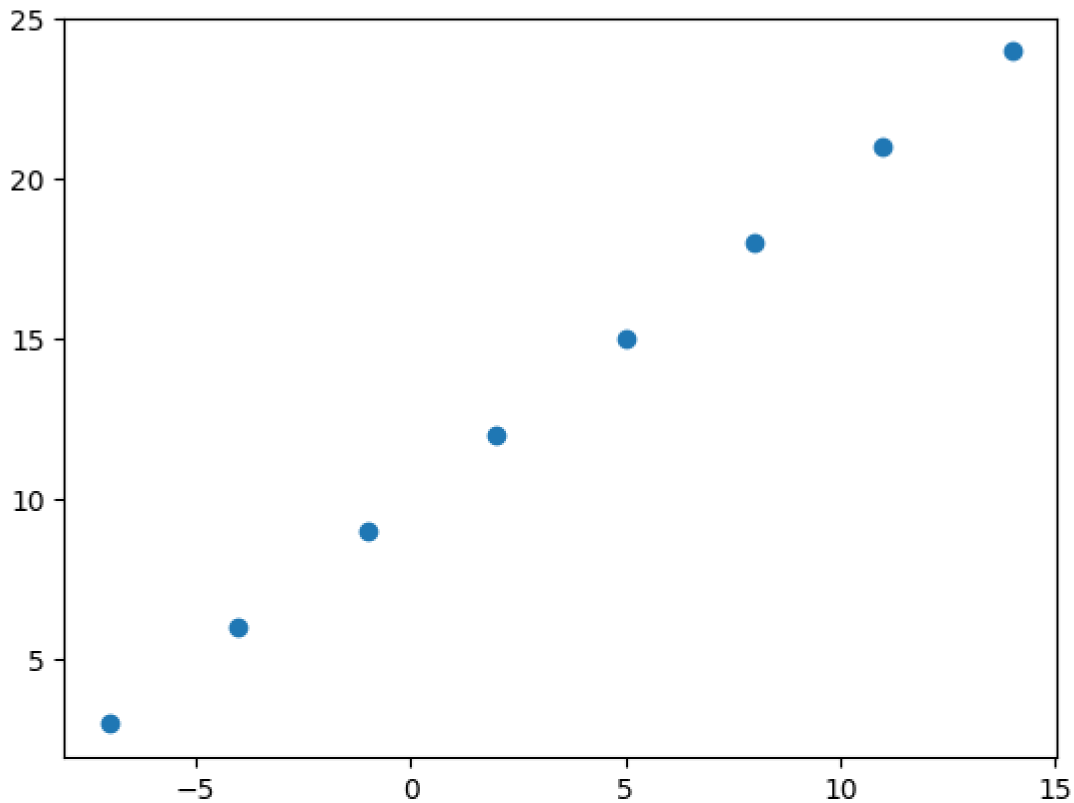
In [4]:

```python
house_info.shape
```

Out[4]:

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

In [5]:
```
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 [6]:
```
# 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[6]:
```
(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 output 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 specifically a rank 0 tensor or a scalar.

In [7]:
```
# Let's take a look at the single examples invidually
X[0], y[0]
```

Out[7]:
```
(<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.

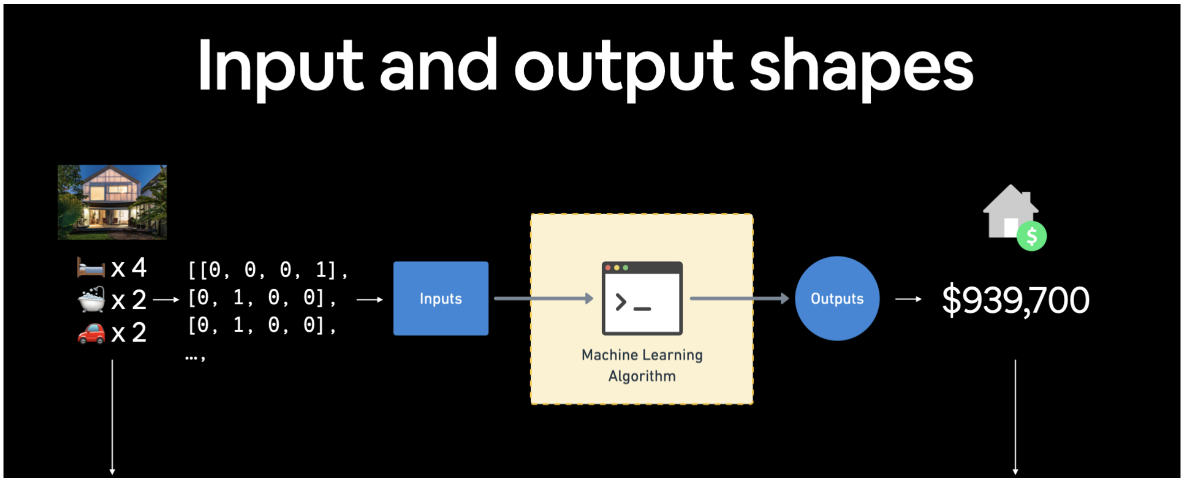
You might be thinking, "this seems pretty complicated for just predicting a straight line...".

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).


Input and output shapes
x 4
x 2
x 2
[[0, 0, 0, 1],
[0, 1, 0, 0],
[0, 1, 0, 0],
...,
Inputs
Machine Learning
Algorithm
Outputs
$939,700

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 (https://www.tensorflow.org/guide/keras/functional) or Sequential API (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential)) 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 (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential) to build a model for our regression data. And then we'll step through each.

> **Note:** If you're using TensorFlow 2.7.0 (https://github.com/tensorflow/tensorflow/releases/tag/v2.7.0)+, the `fit()` function no longer upscales input data to go from `(batch_size, )` to `(batch_size, 1)`. To fix this, you'll need to expand the dimension of input data using `tf.expand_dims(input_data, axis=-1)`.
>
> In our case, this means instead of using `model.fit(X, y, epochs=5)`, use `model.fit(tf.expand_dims(X, axis=-1), y, epochs=5)`.

In [8]:
```python
# 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 absolut
              optimizer=tf.keras.optimizers.SGD(), # SGD is short for s
              metrics=["mae"])

# Fit the model
# model.fit(X, y, epochs=5) # this will break with TensorFlow 2.7.0+
model.fit(tf.expand_dims(X, axis=-1), y, epochs=5)
```

```
Epoch 1/5
1/1 [==============================] - 6s 6s/step - loss: 19.2976 - ma
e: 19.2976
Epoch 2/5
1/1 [==============================] - 0s 18ms/step - loss: 19.0164 -
mae: 19.0164
Epoch 3/5
1/1 [==============================] - 0s 12ms/step - loss: 18.7351 -
mae: 18.7351
Epoch 4/5
1/1 [==============================] - 0s 13ms/step - loss: 18.4539 -
mae: 18.4539
Epoch 5/5
1/1 [==============================] - 0s 13ms/step - loss: 18.1726 -
mae: 18.1726
```

Out[8]:
```
<keras.callbacks.History at 0x7f00663d2680>
```

Boom!

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

How do you think it went?

In [9]:
```python
# Check out X and y
X, y
```

Out[9]:
```
(<tf.Tensor: shape=(8,), dtype=float32, numpy=array([-7., -4., -1.,
2.,  5.,  8., 11., 14.], dtype=float32)>,
 <tf.Tensor: shape=(8,), dtype=float32, numpy=array([ 3.,  6.,  9., 1
2., 15., 18., 21., 24.], dtype=float32)>)
```

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

In [10]:
```
# Make a prediction with the model
model.predict([17.0])
```

```
1/1 [==============================] - 0s 115ms/step
```

Out[10]:
```
array([[-16.701845]], 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 (https://keras.io/)](https://keras.io/) 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.

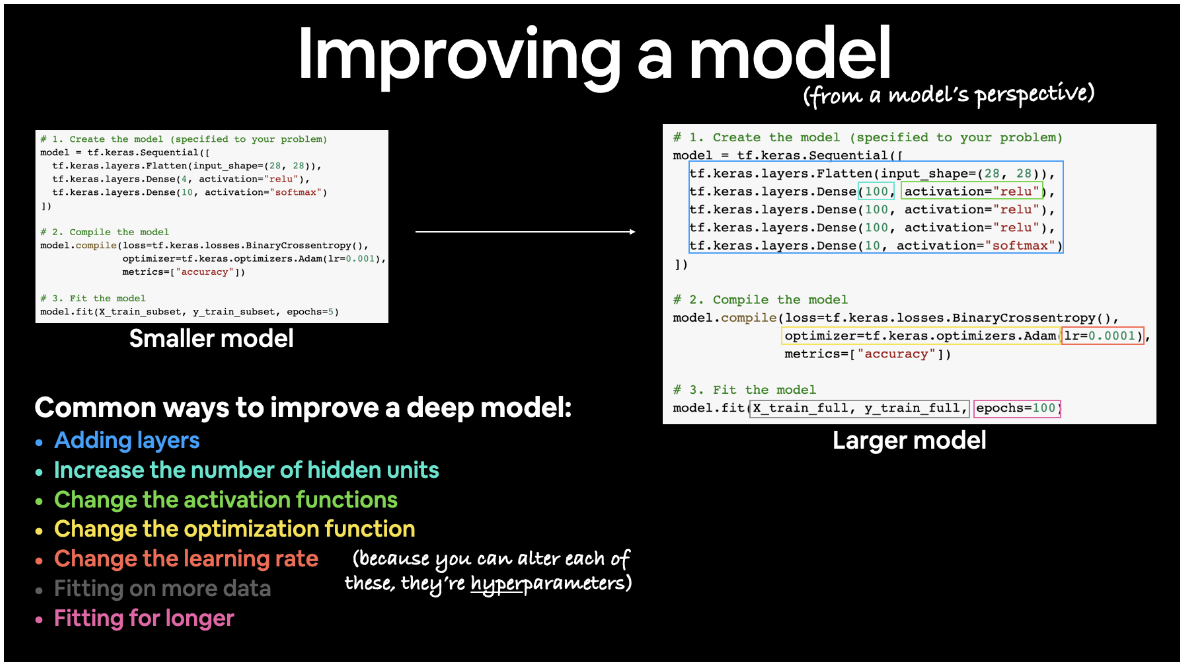
Improving a model

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).



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](https://en.wikipedia.org/wiki/Hyperparameter_(machine_learning)) (https://en.wikipedia.org/wiki/Hyperparameter_(machine_learning)) and the practice of trying to find the best hyperparameters is referred

In [11]:
```
# 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(tf.expand_dims(X, axis=-1), y, epochs=100) # train for 100 ep
```

```
Epoch 1/100
1/1 [==============================] - 1s 604ms/step - loss: 12.9936
- mae: 12.9936
Epoch 2/100
1/1 [==============================] - 0s 14ms/step - loss: 12.8611
- mae: 12.8611
Epoch 3/100
1/1 [==============================] - 0s 16ms/step - loss: 12.7286
- mae: 12.7286
Epoch 4/100
1/1 [==============================] - 0s 16ms/step - loss: 12.5961
- mae: 12.5961
Epoch 5/100
1/1 [==============================] - 0s 15ms/step - loss: 12.4636
- mae: 12.4636
Epoch 6/100
1/1 [==============================] - 0s 15ms/step - loss: 12.3311
- mae: 12.3311
Epoch 7/100
```

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 [12]:
```
# Remind ourselves of what X and y are
X, y
```

Out[12]:
```
(<tf.Tensor: shape=(8,), dtype=float32, numpy=array([-7., -4., -1.,
2.,  5.,  8., 11., 14.], dtype=float32)>,
 <tf.Tensor: shape=(8,), dtype=float32, numpy=array([ 3.,  6.,  9., 1
2., 15., 18., 21., 24.], dtype=float32)>)
```

In [13]:
```python
# 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)
```

```
1/1 [==============================] - 0s 61ms/step
```

Out[13]:
```
array([[29.499516]], 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 -> evaluate it -> build (tweak) 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 [14]:
```
# Make a bigger dataset
X = np.arange(-100, 100, 4)
X
```

Out[14]:
```
array([-100,  -96,  -92,  -88,  -84,  -80,  -76,  -72,  -68,  -64,  -6
0,
        -56,  -52,  -48,  -44,  -40,  -36,  -32,  -28,  -24,  -20,  -1
6,
        -12,   -8,   -4,    0,    4,    8,   12,   16,   20,   24,   2
8,
         32,   36,   40,   44,   48,   52,   56,   60,   64,   68,   7
2,
         76,   80,   84,   88,   92,   96])
```

In [15]:
```
# Make labels for the dataset (adhering to the same pattern as before)
y = np.arange(-90, 110, 4)
y
```

Out[15]:
```
array([-90, -86, -82, -78, -74, -70, -66, -62, -58, -54, -50, -46, -4
2,
       -38, -34, -30, -26, -22, -18, -14, -10,  -6,  -2,   2,   6,  1
0,
        14,  18,  22,  26,  30,  34,  38,  42,  46,  50,  54,  58,  6
2,
        66,  70,  74,  78,  82,  86,  90,  94,  98, 102, 106])
```

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

In [16]:
```
# Same result as above
y = X + 10
y
```

Out[16]:
```
array([-90, -86, -82, -78, -74, -70, -66, -62, -58, -54, -50, -46, -4
2,
       -38, -34, -30, -26, -22, -18, -14, -10,  -6,  -2,   2,   6,  1
0,
        14,  18,  22,  26,  30,  34,  38,  42,  46,  50,  54,  58,  6
2,
        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 [17]:
```
# Check how many samples we have
len(X)
```

Out[17]: 50

In [18]:
```
# 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[18]: (40, 10)

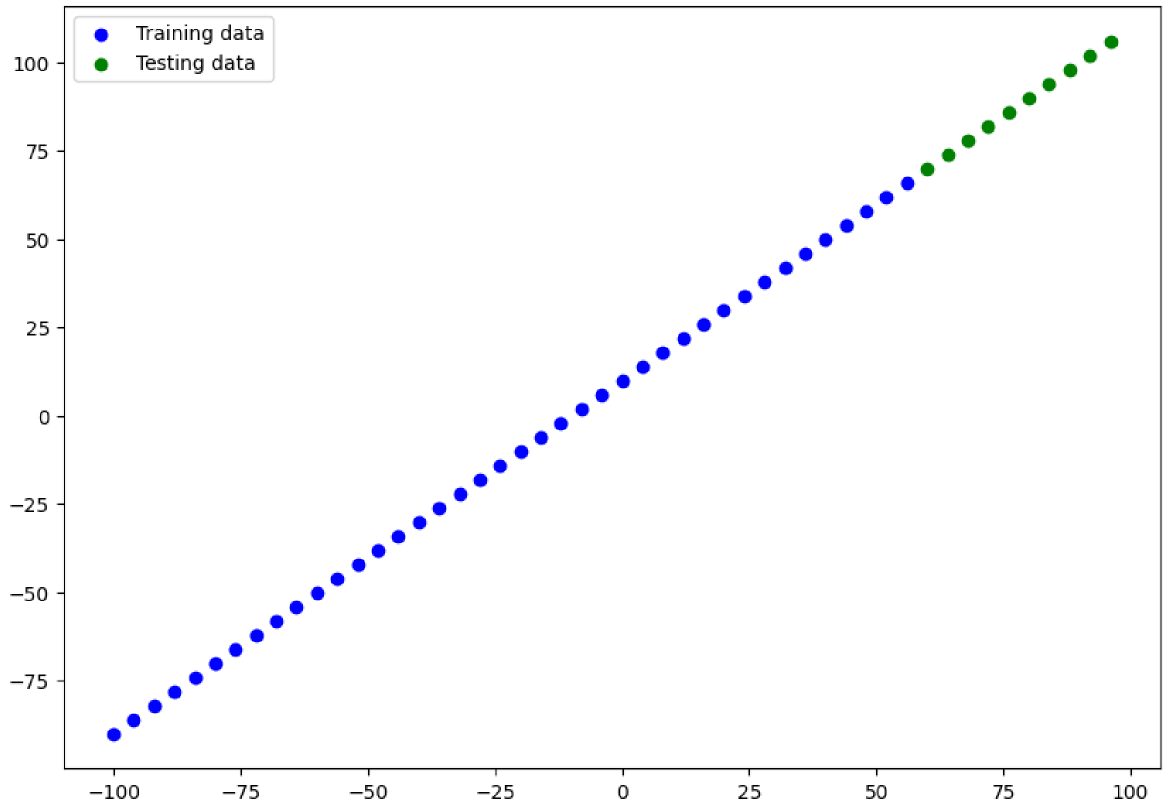
Visualizing the data

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 [19]:

```
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 [20]:
```
# 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 (no
```

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() (https://www.tensorflow.org/api_docs/python/tf/keras/Model#summary) on it.

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

In [22]:
```
# Doesn't work (model not fit/built)
model.summary()
```

```
-----------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
<ipython-input-22-7d09d31d4e66> in <cell line: 2>()
      1 # Doesn't work (model not fit/built)
----> 2 model.summary()

/usr/local/lib/python3.10/dist-packages/keras/engine/training.py in su
mmary(self, line_length, positions, print_fn, expand_nested, show_trai
nable, layer_range)
   3227         """
   3228         if not self.built:
-> 3229             raise ValueError(
   3230                 "This model has not yet been built. "
   3231                 "Build the model first by calling `build()` or
by calling "

ValueError: This model has not yet been built. Build the model first b
y calling `build()` or by calling the model on a batch of data.
```

Ahh, the cell above errors because we haven't fit or 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 [23]:
```
# 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
])

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

In [24]:
```python
# This will work after specifying the input shape
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense_3 (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 (https://youtu.be/njKP3FqW3Sk)](https://youtu.be/njKP3FqW3Sk).

> 🛠 **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 [25]:
```python
# Fit the model to the training data
model.fit(X_train, y_train, epochs=100, verbose=0) # verbose controls h
```

Out[25]: <keras.callbacks.History at 0x7f0065dd80d0>

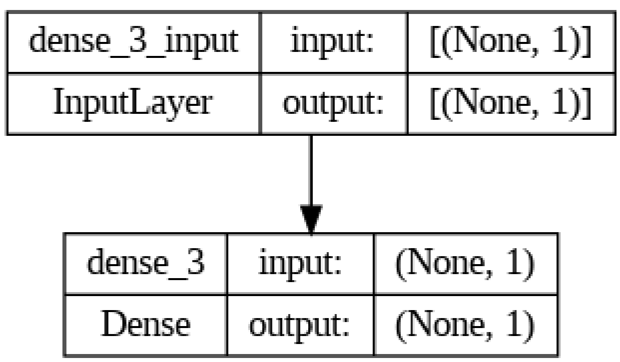
In [26]:
```python
# Check the model summary
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (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() (https://www.tensorflow.org/api_docs/python/tf/keras/utils/plot_model)](https://www.tensorflow.org/api_docs/python/tf/keras/utils/plot_model).

In [27]:
```python
from tensorflow.keras.utils import plot_model

plot_model(model, show_shapes=True)
```

Out[27]:



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 [28]:
```python
# Make predictions
y_preds = model.predict(X_test)
```

```
1/1 [==============================] - 0s 45ms/step
```

In [29]:
```python
# View the predictions
y_preds
```

Out[29]:
```
array([[44.544697],
       [47.427135],
       [50.309574],
       [53.192013],
       [56.07445 ],
       [58.95689 ],
       [61.839325],
       [64.72176 ],
       [67.60421 ],
       [70.48664 ]], dtype=float32)
```

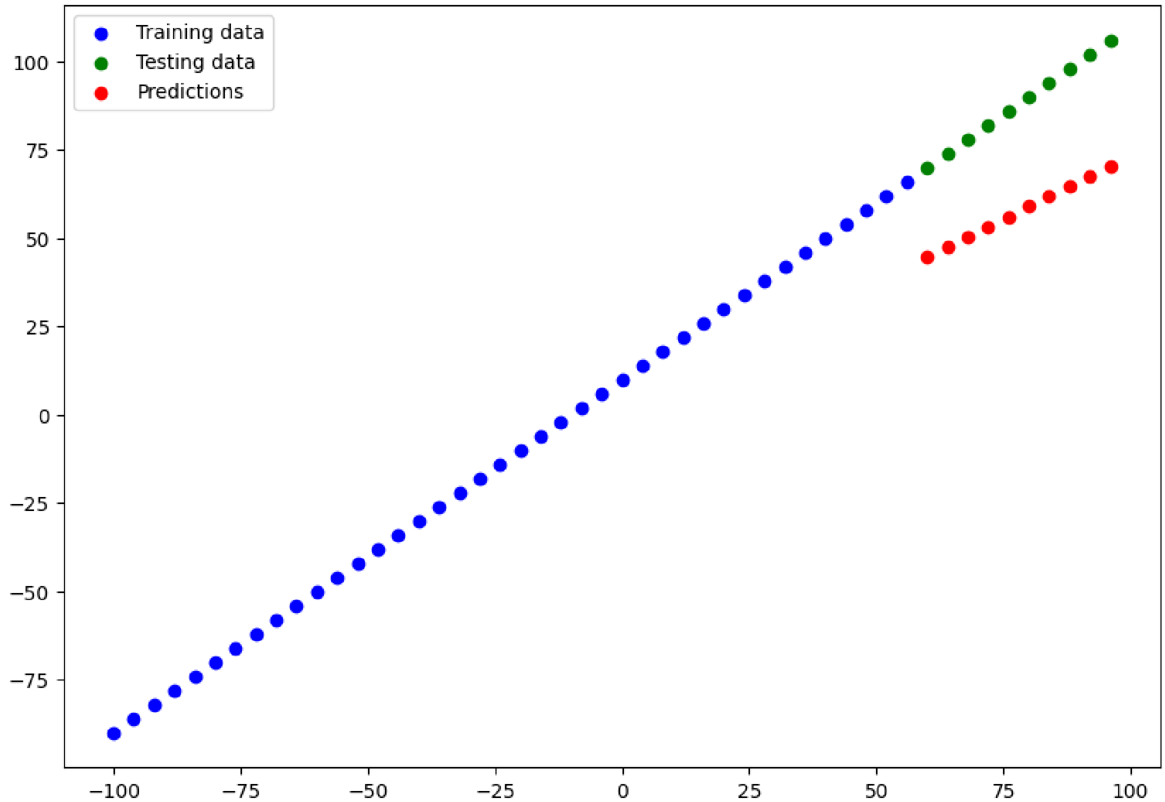
Okay, we get a list of numbers but how do these compare to the ground truth labels?

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 [30]:
```python
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 [31]:
```
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() (https://www.tensorflow.org/api_docs/python/tf/keras/Model#evaluate)](https://www.tensorflow.org/api_docs/python/tf/keras/Model#evaluate) which will return the loss of the model as well as any metrics setup

In [32]:
```
# Evaluate the model on the test set
model.evaluate(X_test, y_test)
```

```
1/1 [==============================] - 0s 132ms/step - loss: 30.4843 - mae: 30.4843
```

Out[32]:
```
[30.484329223632812, 30.484329223632812]
```

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 [33]:
```
# Calculate the mean absolute error
mae = tf.metrics.mean_absolute_error(y_true=y_test,
                                     y_pred=y_preds)
mae
```

Out[33]:
```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([43.455303, 40.572865, 37.690426, 34.807987, 31.925549, 29.04311
,
       26.160675, 23.278236, 20.39579 , 17.610687], 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 [34]:
```
# Check the test label tensor values
y_test
```

Out[34]:
```
array([ 70,  74,  78,  82,  86,  90,  94,  98, 102, 106])
```

In [35]:
```
# Check the predictions tensor values (notice the extra square brackets
y_preds
```

Out[35]:
```
array([[44.544697],
       [47.427135],
       [50.309574],
       [53.192013],
       [56.07445 ],
       [58.95689 ],
       [61.839325],
       [64.72176 ],
       [67.60421 ],
       [70.48664 ]], dtype=float32)
```

In [36]:
```
# Check the tensor shapes
y_test.shape, y_preds.shape
```

Out[36]:
```
((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() (https://www.tensorflow.org/api_docs/python/tf/squeeze)](https://www.tensorflow.org/api_docs/python/tf/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 [37]:
```
# Shape before squeeze()
y_preds.shape
```

Out[37]:
```
(10, 1)
```

In [38]:
```
# Shape after squeeze()
y_preds.squeeze().shape
```

Out[38]:
```
(10,)
```

In [39]:
```python
# What do they look like?
y_test, y_preds.squeeze()
```

Out[39]:
```
(array([ 70,  74,  78,  82,  86,  90,  94,  98, 102, 106]),
 array([44.544697, 47.427135, 50.309574, 53.192013, 56.07445 , 58.9568
9 ,
        61.839325, 64.72176 , 67.60421 , 70.48664 ], 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 [40]:
```python
# Calcuate the MAE
mae = tf.metrics.mean_absolute_error(y_true=y_test,
                                     y_pred=y_preds.squeeze()) # use sq
mae
```

Out[40]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=30.48433>
```

In [41]:
```python
# Calculate the MSE
mse = tf.metrics.mean_squared_error(y_true=y_test,
                                    y_pred=y_preds.squeeze())
mse
```

Out[41]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=939.59827>
```

We can also calculate the MAE using pure TensorFlow functions.

In [42]:
```python
# Returns the same as tf.metrics.mean_absolute_error()
tf.reduce_mean(tf.abs(y_test-y_preds.squeeze()))
```

Out[42]:
```
<tf.Tensor: shape=(), dtype=float64, numpy=30.484329986572266>
```

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 [43]:
```python
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.

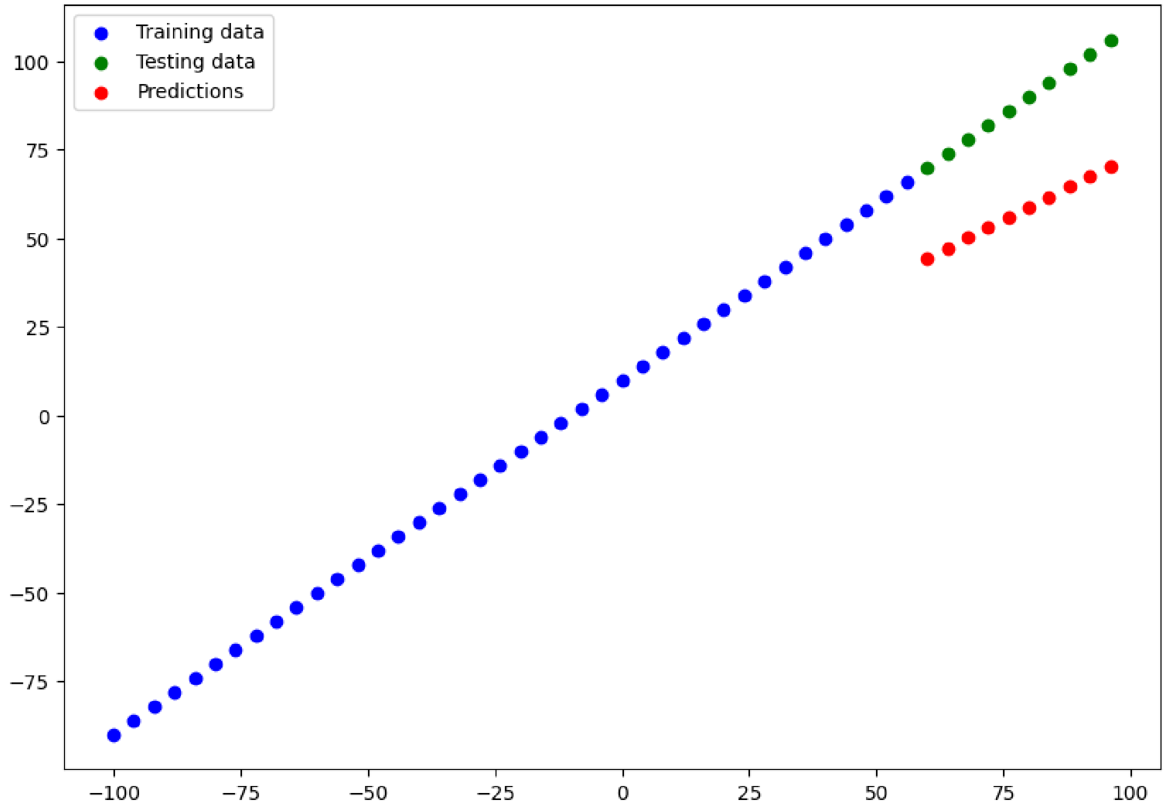
Build `model_1`

In [44]:
```
# 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(tf.expand_dims(X_train, axis=-1), y_train, epochs=100)
```

```
Epoch 1/100
2/2 [==============================] - 0s 19ms/step - loss: 30.0988
- mae: 30.0988
Epoch 2/100
2/2 [==============================] - 0s 8ms/step - loss: 8.4388 -
mae: 8.4388
Epoch 3/100
2/2 [==============================] - 0s 6ms/step - loss: 10.5960 -
mae: 10.5960
Epoch 4/100
2/2 [==============================] - 0s 6ms/step - loss: 13.1312 -
mae: 13.1312
Epoch 5/100
2/2 [==============================] - 0s 7ms/step - loss: 12.1970 -
mae: 12.1970
Epoch 6/100
2/2 [==============================] - 0s 6ms/step - loss: 9.4357 -
mae: 9.4357
Epoch 7/100
```

In [45]:
```python
# Make and plot predictions for model_1
y_preds_1 = model_1.predict(X_test)
plot_predictions(predictions=y_preds_1)
```

```
1/1 [==============================] - 0s 45ms/step
```



In [46]:
```python
# 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[46]:
```
(30.638134, 949.13086)
```

Build `model_2`

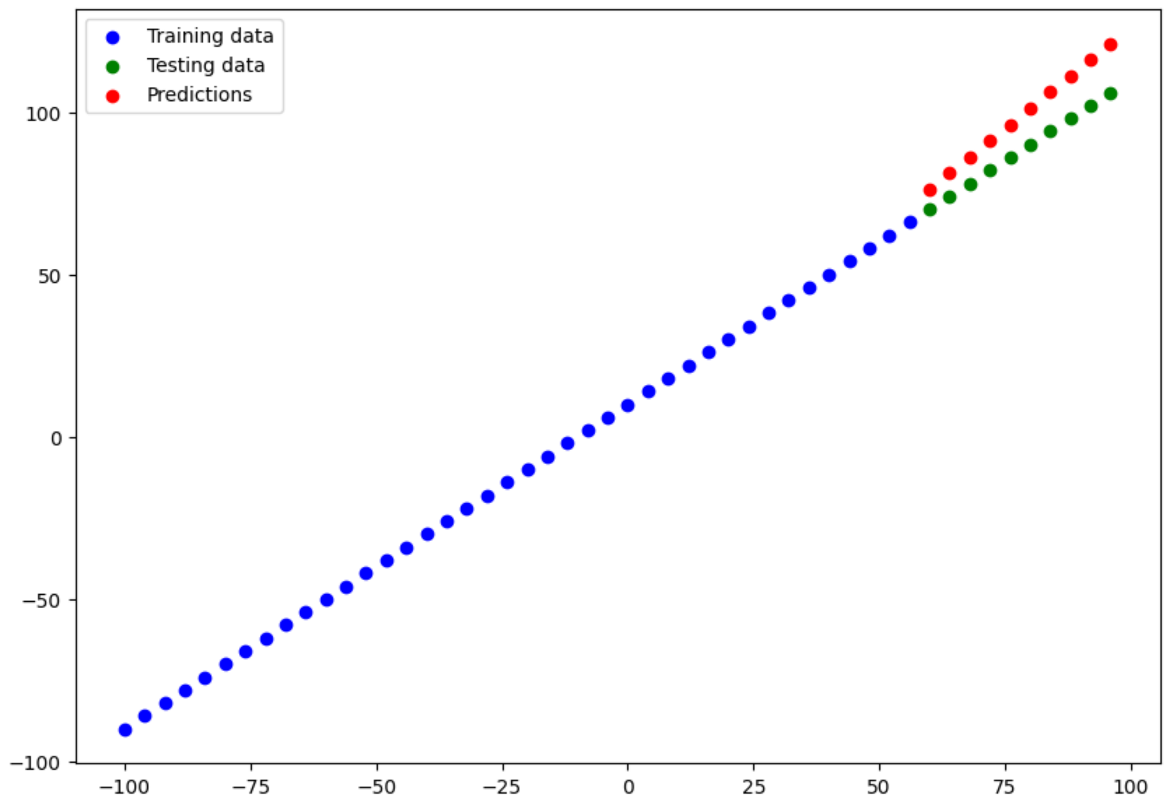
This time we'll add an extra dense layer (so now our model will have 2 layers) whilst keeping everything else the same.

In [47]:
```
# 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(tf.expand_dims(X_train, axis=-1), y_train, epochs=100, verb
```

Out[47]:
```
<keras.callbacks.History at 0x7f00643ba560>
```

In [48]:
```
# 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.<locals>.predict_function at 0x7f006436f880> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly 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 reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing (https://www.tensorflow.org/guide/function#controlling_retracing) and https://www.tensorflow.org/api_docs/python/tf/function (https://www.tensorflow.org/api_docs/python/tf/function) for  more details.

1/1 [==============================] - 0s 51ms/step
```



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

In [49]:
```
# 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[49]:
```
(10.610324, 120.35542)
```

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.

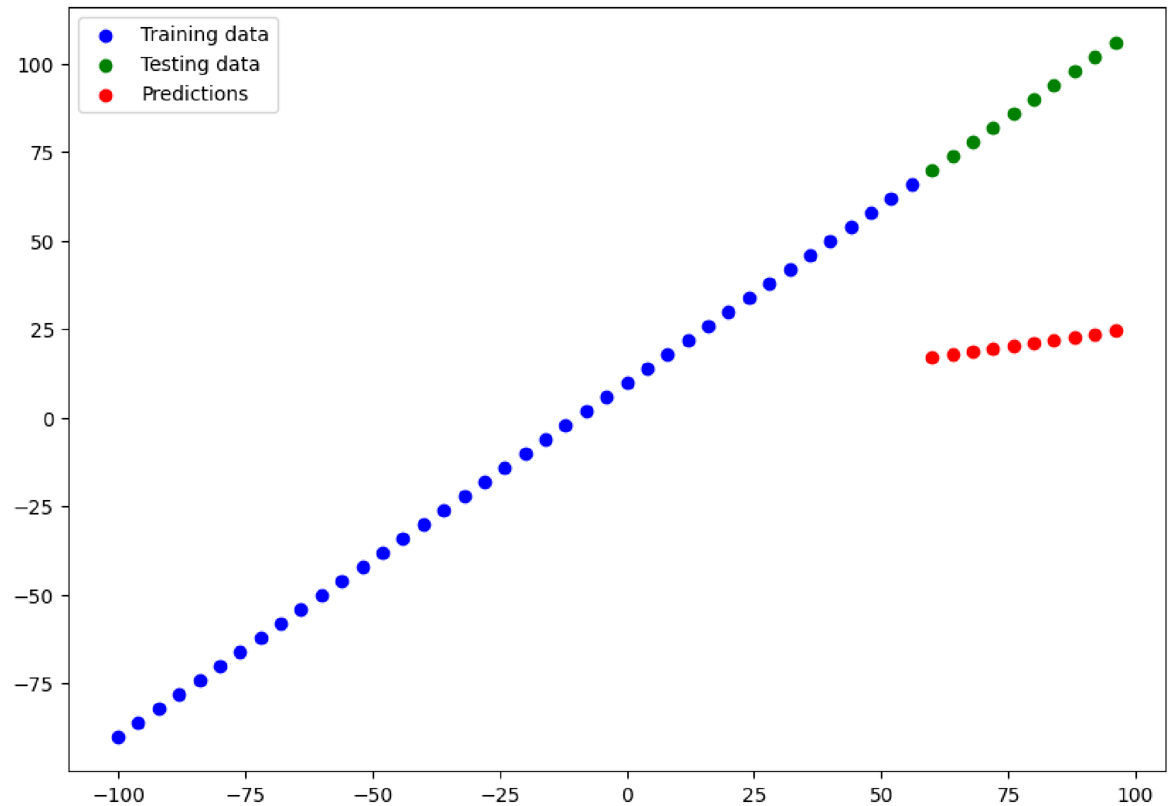
In [50]:
```
# Set random seed
tf.random.set_seed(42)

# 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(tf.expand_dims(X_train, axis=-1), y_train, epochs=500, verb
```

Out[50]: <keras.callbacks.History at 0x7f0065a5c8e0>

In [51]:
```
# 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.<locals>.predict_function at 0x7f00641280d0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly 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 reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing (https://www.tensorflow.org/guide/function#controlling_retracing) and https://www.tensorflow.org/api_docs/python/tf/function (https://www.tensorflow.org/api_docs/python/tf/function) for  more details.
```

```
1/1 [==============================] - 0s 55ms/step
```



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 [52]:
```python
# 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[52]:
```
(67.224594, 4601.822)
```

Comparing results

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

In [53]:
```python
model_results = [["model_1", mae_1, mse_1],
                 ["model_2", mae_2, mse_2],
                 ["model_3", mae_3, mae_3]]
```

In [54]:
```python
import pandas as pd
all_results = pd.DataFrame(model_results, columns=["model", "mae", "mse
all_results
```

Out[54]:

	model	mae	mse
0	model_1	30.638134	949.130859
1	model_2	10.610324	120.355423
2	model_3	67.224594	67.224594

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](https://tensorboard.dev/)** (https://tensorboard.dev/) - a component of the TensorFlow library to help track modelling experiments (we'll see this later).
- **[Weights & Biases](https://www.wandb.com/)** (https://www.wandb.com/) - a tool for tracking all kinds of machine learning experiments (the good news for

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()` (https://www.tensorflow.org/tutorials/keras/save_and_load#save_the_enti

There are two ways to save a model in TensorFlow:

1. The SavedModel format (https://www.tensorflow.org/tutorials/keras/save_and_load#savedmodel (default).
2. The HDF5 format (https://www.tensorflow.org/tutorials/keras/save_and_load#hdf5_forma

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 [55]:
```
# Save a model using the SavedModel format
model_2.save('best_model_SavedModel_format')
```

```
WARNING:absl:Found untraced functions such as _update_step_xla while s
aving (showing 1 of 1). These functions will not be directly callable
after loading.
```

In [56]:
```
# Check it out - outputs a protobuf binary file (.pb) as well as other
!ls best_model_SavedModel_format
```

```
assets  fingerprint.pb  keras_metadata.pb  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 [57]:
```
# Save a model using the HDF5 format
model_2.save("best_model_HDF5_format.h5") # note the addition of '.h5'
```

In [58]:
```
# 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() (https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model) method.

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

In [59]:
```
# Load a model from the SavedModel format
loaded_saved_model = tf.keras.models.load_model("best_model_SavedModel_
loaded_saved_model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense_5 (Dense)             (None, 1)                 2         
                                                                 
 dense_6 (Dense)             (None, 1)                 2         
                                                                 
=================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
_________________________________________________________________
```

Now let's test it out.

In [60]:
```
# 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
```

```
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 56ms/step
```

Out[60]: True

Loading in from the HDF5 is much the same.

In [61]:
```
# 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_5"

_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense_5 (Dense)             (None, 1)                 2         
                                                                 
 dense_6 (Dense)             (None, 1)                 2         
                                                                 
=================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
_________________________________________________________________
```

In [62]:
```
# 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_pr
```

```
1/1 [==============================] - 0s 53ms/step
```

Out[62]: 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 [63]:
```
# Download the model (or any file) from Google Colab
from google.colab import files
files.download("best_model_HDF5_format.h5")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

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 dataset.

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 (https://www.kaggle.com/mirichoi0218/insurance)](https://www.kaggle.com/mirichoi0218/insurance) available from Kaggle and [hosted on GitHub (https://github.com/stedy/Machine-Learning-with-R-datasets/blob/master/insurance.csv)](https://github.com/stedy/Machine-Learning-with-R-datasets/blob/master/insurance.csv).

> 🔑 **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 [64]:
```
# Import required libraries
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt
```

In [65]:
```
# Read in the insurance dataset
insurance = pd.read_csv("https://raw.githubusercontent.com/stedy/Machin
```

In [66]:

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

Out[66]:

	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() (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html) method in pandas.

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

In [67]:

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

Out[67]:

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

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

In [68]:

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

In [69]:
```python
# View features
X.head()
```

Out[69]:

	age	bmi	children	sex_female	sex_male	smoker_no	smoker_yes	region_northea
0	19	27.900	0	1	0	0	1	
1	18	33.770	1	0	1	1	0	
2	28	33.000	3	0	1	1	0	
3	33	22.705	0	0	1	1	0	
4	32	28.880	0	0	1	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` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_s function available from Scikit-Learn.

In [70]:
```python
# 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) #
```

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

In [71]:
```
# 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)
```

```
Epoch 1/100
34/34 [==============================] - 1s 3ms/step - loss: 9369.49
71 - mae: 9369.4971
Epoch 2/100
34/34 [==============================] - 0s 3ms/step - loss: 7851.06
15 - mae: 7851.0615
Epoch 3/100
34/34 [==============================] - 0s 3ms/step - loss: 7567.60
06 - mae: 7567.6006
Epoch 4/100
34/34 [==============================] - 0s 3ms/step - loss: 7540.68
12 - mae: 7540.6812
Epoch 5/100
34/34 [==============================] - 0s 3ms/step - loss: 7695.30
62 - mae: 7695.3062
Epoch 6/100
34/34 [==============================] - 0s 3ms/step - loss: 7608.97
12 - mae: 7608.9712
Epoch 7/100
```

In [72]:
```
# Check the results of the insurance model
insurance_model.evaluate(X_test, y_test)
```

```
9/9 [==============================] - 0s 2ms/step - loss: 6392.2939 -
mae: 6392.2939
```

Out[72]:
```
[6392.2939453125, 6392.2939453125]
```

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 [73]:
```
# 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
                          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
```

In [74]:
```
# Evaluate our larger model
insurance_model_2.evaluate(X_test, y_test)
```

```
9/9 [==============================] - 0s 2ms/step - loss: 4629.1626 -
mae: 4629.1626
```

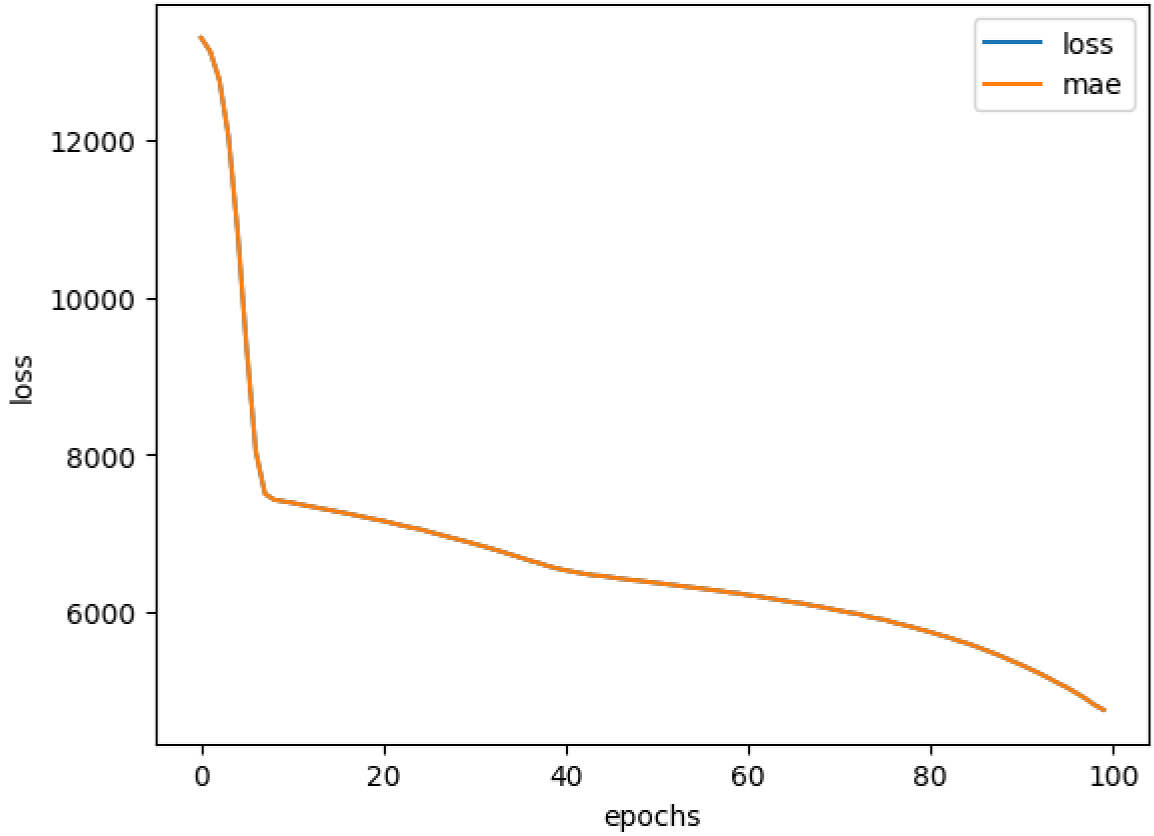
Out[74]:
```
[4629.16259765625, 4629.16259765625]
```

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/A (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/A) 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/) (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 [75]:
```python
# 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/Ea](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/Ea) so it stops automatically when it stops improving. We'll see this in another module.

In [76]:
```
# Try training for a little longer (100 more epochs)
history_2 = insurance_model_2.fit(X_train, y_train, epochs=100, verbose
```

How did the extra training go?

In [77]:
```
# Evaluate the model trained for 200 total epochs
insurance_model_2_loss, insurance_model_2_mae = insurance_model_2.evalu
insurance_model_2_loss, insurance_model_2_mae
```

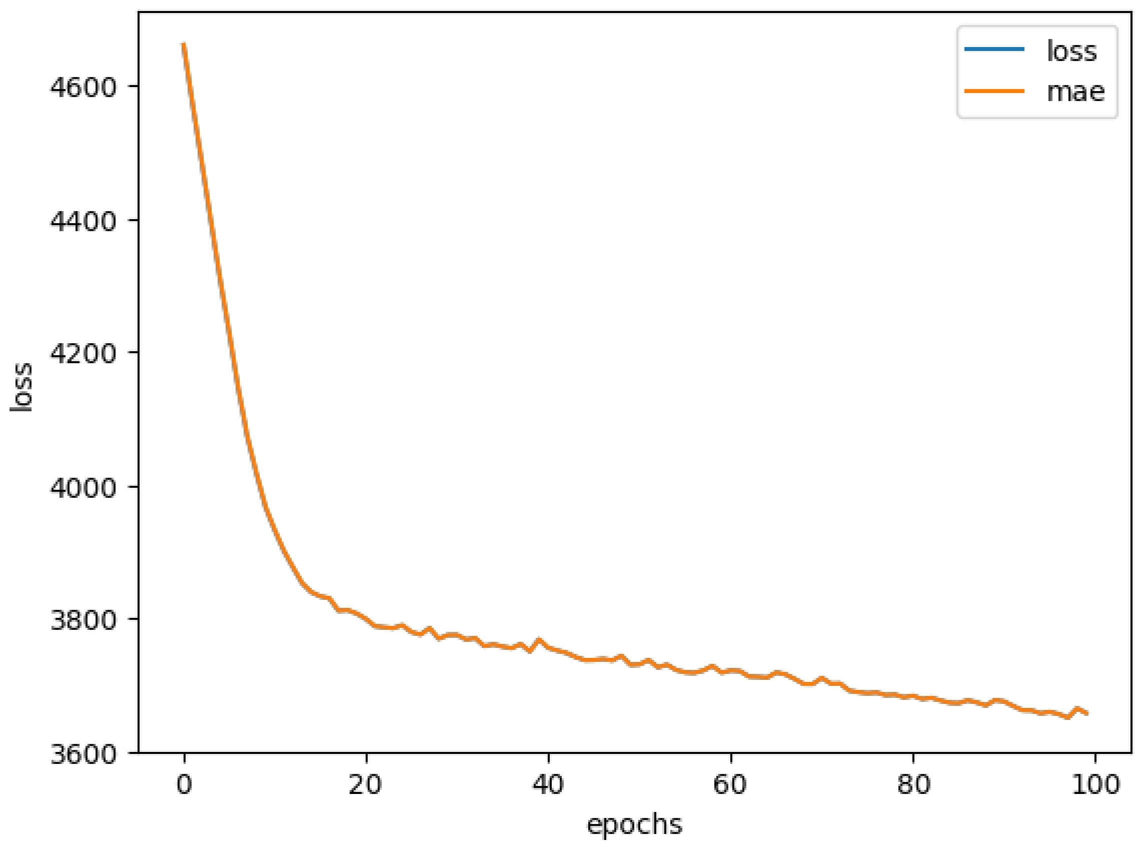
```
9/9 [==============================] - 0s 3ms/step - loss: 3483.4031 -
mae: 3483.4031
```

Out[77]:
```
(3483.403076171875, 3483.403076171875)
```

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

How does the visual look?

In [78]:
```
# 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 overri
```



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** (https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler)](https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler):
  - 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 minimum. 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 subtracting 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 (https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-data)](https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-data).
- [Scale, Standardize or Normalize with Scikit-Learn by Jeff Hale (https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02)](https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02).

We've already turned our data into numbers using `get_dummies()` ,

In [79]:

```python
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/Machin
```

In [80]:

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

Out[80]:

	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 (https://scikit-learn.org/stable/modules/generated/sklearn.compose.make_column_trans - build a multi-step data preprocessing function for the folllowing transformations:
  - MinMaxScaler (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxS - make sure all numerical columns are normalized (between 0 and 1).
  - OneHotEncoder (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotE - one hot encode the non-numerical columns.

Let's see them in action.

In [81]:
```python
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder

# Create column transformer (this will help us normalize/preprocess our
ct = make_column_transformer(
    (MinMaxScaler(), ["age", "bmi", "children"]), # get all values betw
    (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
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2

# Fit column transformer on the training data only (doing so on test da
ct.fit(X_train)

# Transform training and test data with normalization (MinMaxScalar) an
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 [82]:
```python
# Non-normalized and non-one-hot encoded data example
X_train.loc[0]
```

Out[82]:
```
age                19
sex            female
bmi              27.9
children            0
smoker            yes
region      southwest
Name: 0, dtype: object
```

In [83]:
```python
# Normalized and one-hot encoded example
X_train_normal[0]
```

Out[83]:
```
array([0.60869565, 0.10734463, 0.4       , 1.        , 0.        ,
       1.        , 0.        , 0.        , 1.        , 0.        ,
       0.        ])
```

How about the shapes?

In [84]:
```python
# Notice the normalized/one-hot encoded shape is larger because of the
X_train_normal.shape, X_train.shape
```

Out[84]:
```
((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 [85]:
```
# 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[85]:
```
<keras.callbacks.History at 0x7f004cd64af0>
```

Let's evaluate the model on normalized test set.

In [86]:
```
# Evaulate 3rd model
insurance_model_3_loss, insurance_model_3_mae = insurance_model_3.evalu
```

```
9/9 [==============================] - 0s 3ms/step - loss: 3171.2595 -
mae: 3171.2595
```

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 [87]:
```
# Compare modelling results from non-normalized data and normalized dat
insurance_model_2_mae, insurance_model_3_mae
```

Out[87]:
```
(3483.403076171875, 3171.259521484375)
```

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 practitioners is to

🛠 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 intuition.

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 (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/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 (https://www.tensorflow.org/api_docs/python/tf/keras/datasets/boston from TensorFlow `tf.keras.datasets` (https://www.tensorflow.org/api_docs/python/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 (https://youtu.be/njKP3FqW3Sk) - 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 (http://neuralnetworksanddeeplearning.com/chap1.html) 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 (https://lionbridge.ai/datasets/)](https://lionbridge.ai/datasets/) or [Kaggle's datasets (https://www.kaggle.com/data)](https://www.kaggle.com/data), find a regression dataset which sparks