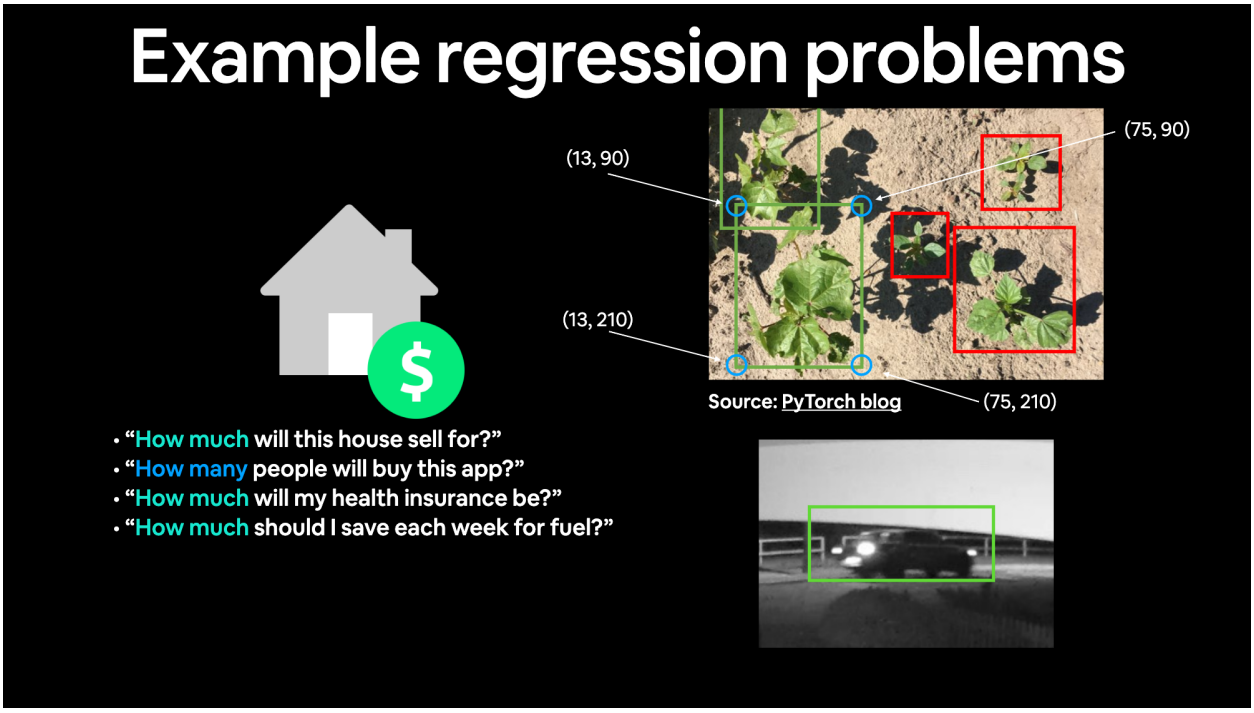
Neural Network Regression with TensorFlow

Table of contents:

- Introduction to Neural Network Regressions
  - Regression problems
  - Regression inputs and outputs
    - Inputs and Outputs
    - Input and Output shapes
    - What might be the shape of our inputs and outputs ?? 🤔
  - Anatomy and Architecture of a Neural Network Regression Model
  - Input and Shapes continued
  - Creating a data and view (visualize) it
  - Steps in Modelling with TensorFlow
    - Keras Sequential API
    - What's Keras? 🤔
  - Improving our model
  - Evaluating a model
    - Visualize, visualize, visualize
  - Split data into a training/validation/test sets
  - Visualize the data
  - Visualize model
  - Visualizing model's predictions
  - Evaluating our model's predictions with Regression Evaluation Metrics
  - Running Experiments to improve our Model
  - Comparing the results of our models
  - Tracking the experiments
  - Saving our model
  - Loading in saved model
  - How long I should train my model ? 🤔
  - Feature Scaling

Introduction to Neural Network Regressions

Regression problems



Regression problem - predicting a number of some sort.

🤔 Wikipedia info:

> In statistical modeling, **regression analysis** is a set of statistical processes for estimating the relationships between a dependent variable (often called the 'outcome' or 'response' variable) and one or more independent variables (often called 'predictors', 'covariates', 'explanatory variables' or 'features').
>
> https://en.wikipedia.org/wiki/Regression_analysis

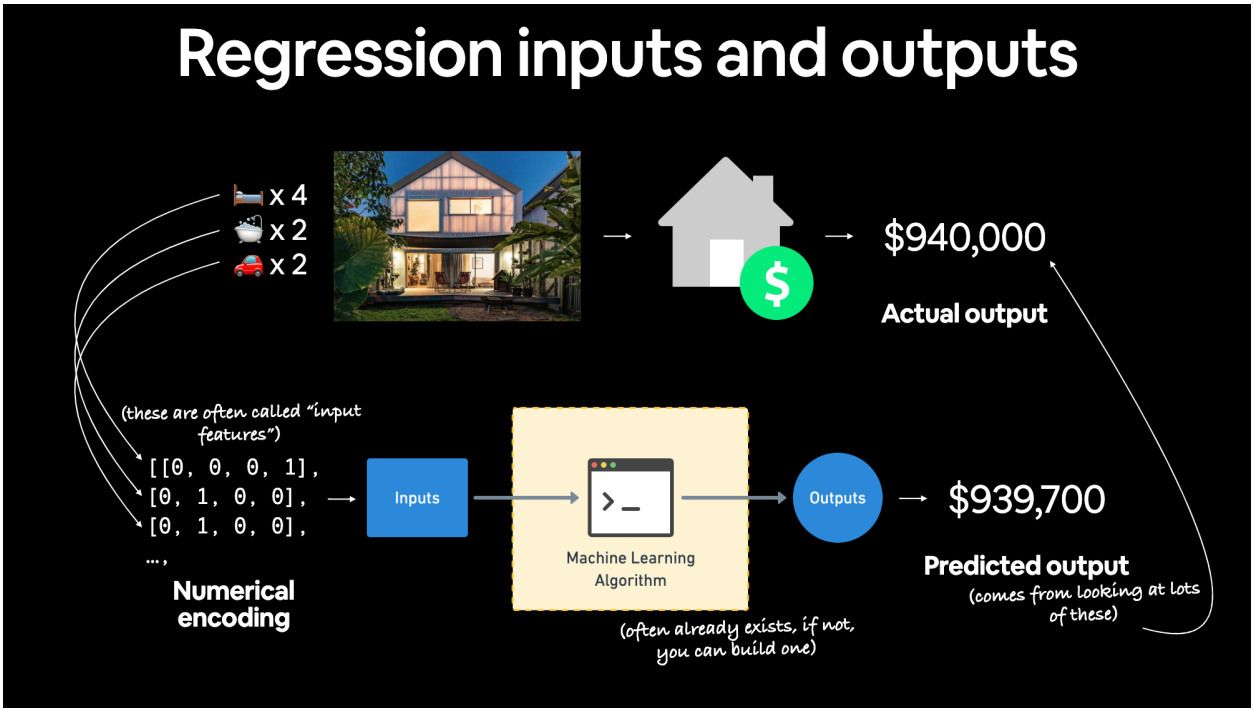
In our example problem of trying to predict the house price, our dependent variable might be the house price. The house price is the outcome we're trying to predict.

Independent variables often called predictors, features might be:

- Number of rooms
- Number of bathrooms
- Number of garage spaces
- Etc..

So, in our records, we might have 10 different houses with 10 different numbers of rooms, bathrooms, garage spaces.

Regression inputs and outputs



Input features - some kind of information about the data we're using that goes into our Machine Learning algorithm.

Often an algorithm for our problem already exists. So that means that someone has built some algorithm that has worked before for their problem, might be very similar to our problem, and we can utilize that for whatever we are working on.

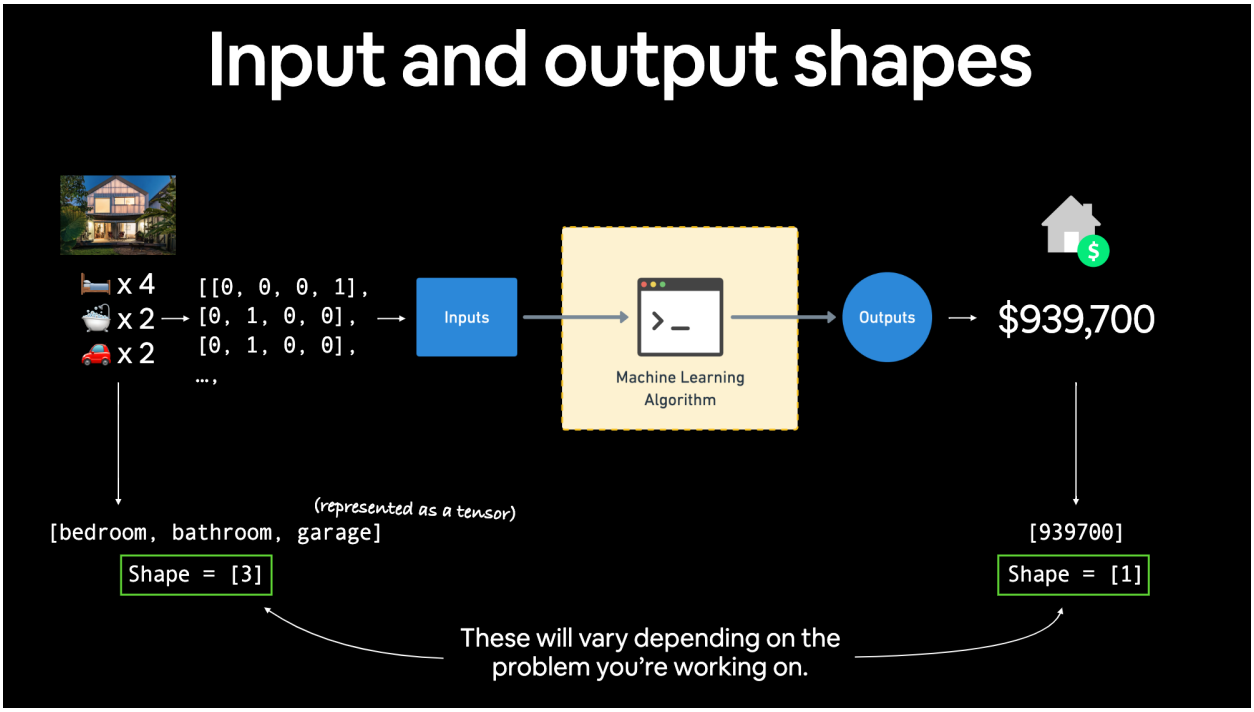
In our example, Machine Learning algorithm has looked at many, many examples that come from looking at input features and outputs. It is going to learn relationships between the input features and the outputs.

Then, for the use cases, where we would like to predict the potential output of a home that we don't know that actual sales price for. We can feed its input features into our algorithm and have suggested output of what the sale price.

Inputs and Outputs

We will be focusing a lot of time on inputs and outputs in Machine Learning, Deep Learning.

Input and Output shapes



- Take input features
- Numerically encode them
- Feed them into our Machine Learning algorithm as inputs
- Machine Learning algorithm works out the patterns or utilizes if it has already learned patterns to produce the output

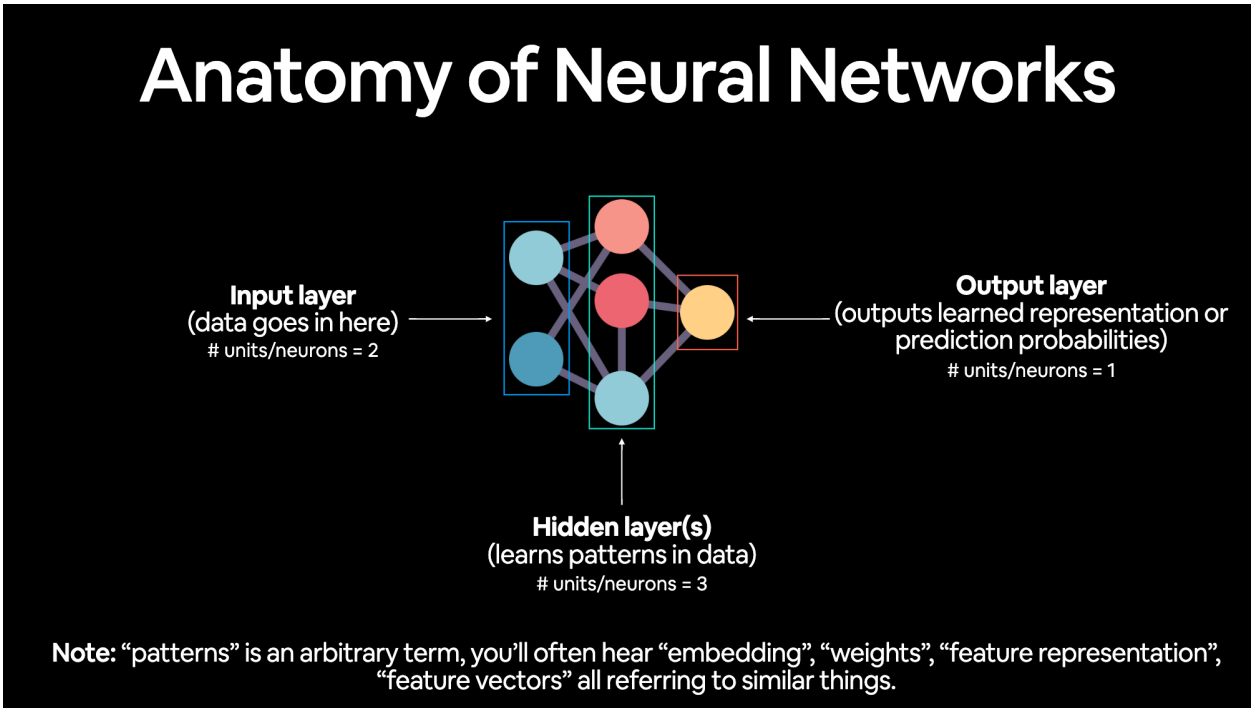
What might be the shape of our inputs and outputs ?? 🤔

Our numerical encoding will be on a form of a tensor, and output will be also in a form of a tensor.

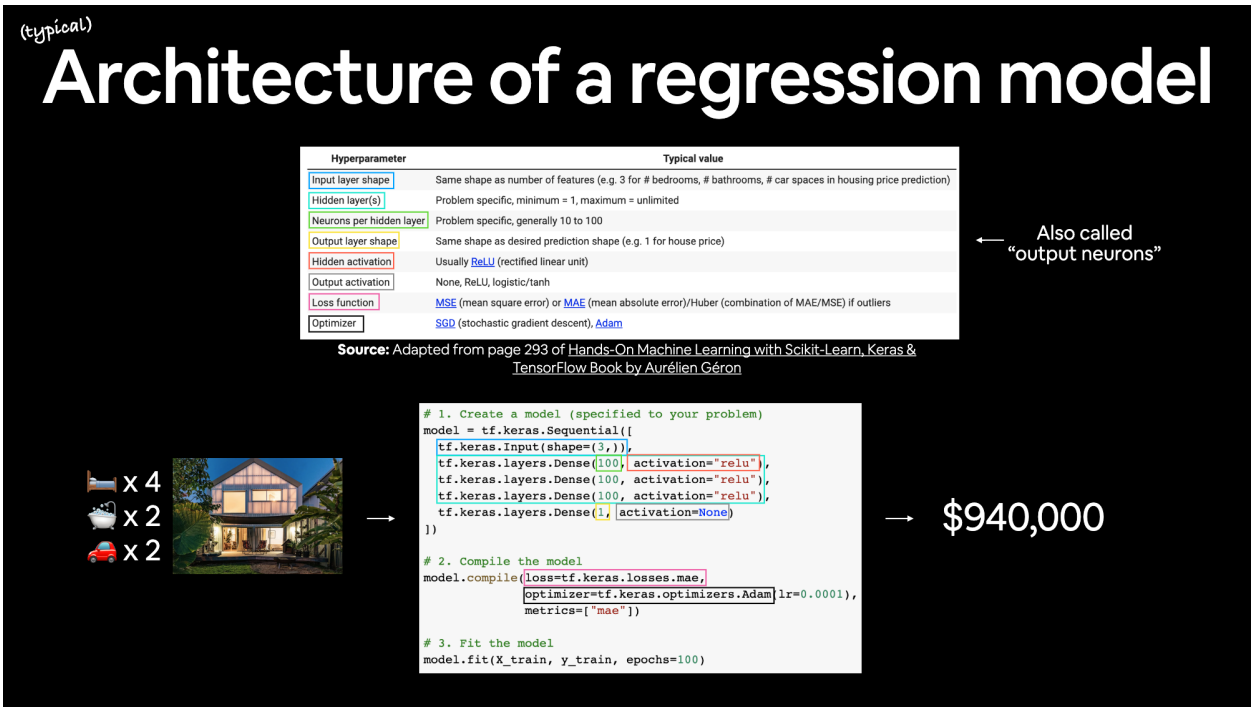
In our example, we have [bedroom, bathroom, garage] represented as a tensor, in this case, the shape is going to be [3] because we have got 3 input vectors. For the output, the shape will be [1]

For a regression problem usually, the output shape is [1] because we are trying to predict a some sort of number.

Anatomy and Architecture of a Neural Network Regression Model



Typical architecture of a Neural Network Regression model



- Input shape - we might have an input layer shape as the same as the number of input features that we have

- Hidden layers - the values are going to be problem specific. We can customize the number of hidden layers, and how many neurons per layer we have.
- Output layer - Same shape as desired prediction shape
- Hidden activation - Usually RELU (Rectified Linear Unit)
- Output activation - the values is going to be problem specific. Typical values: None, ReLu, logistic/tanh
- Loss function - MSE (Mean Squared Error) or MAE (Mean Absolute Error)/Huber (combination of MAE / MSE) if we have outliers.
- Optimizer - (A way of our neural network improves its predictions) Usually SGD (Stochastic Gradient Decent) or Adam optimizer.

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

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).

Input and Shapes continued

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.
- 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.**

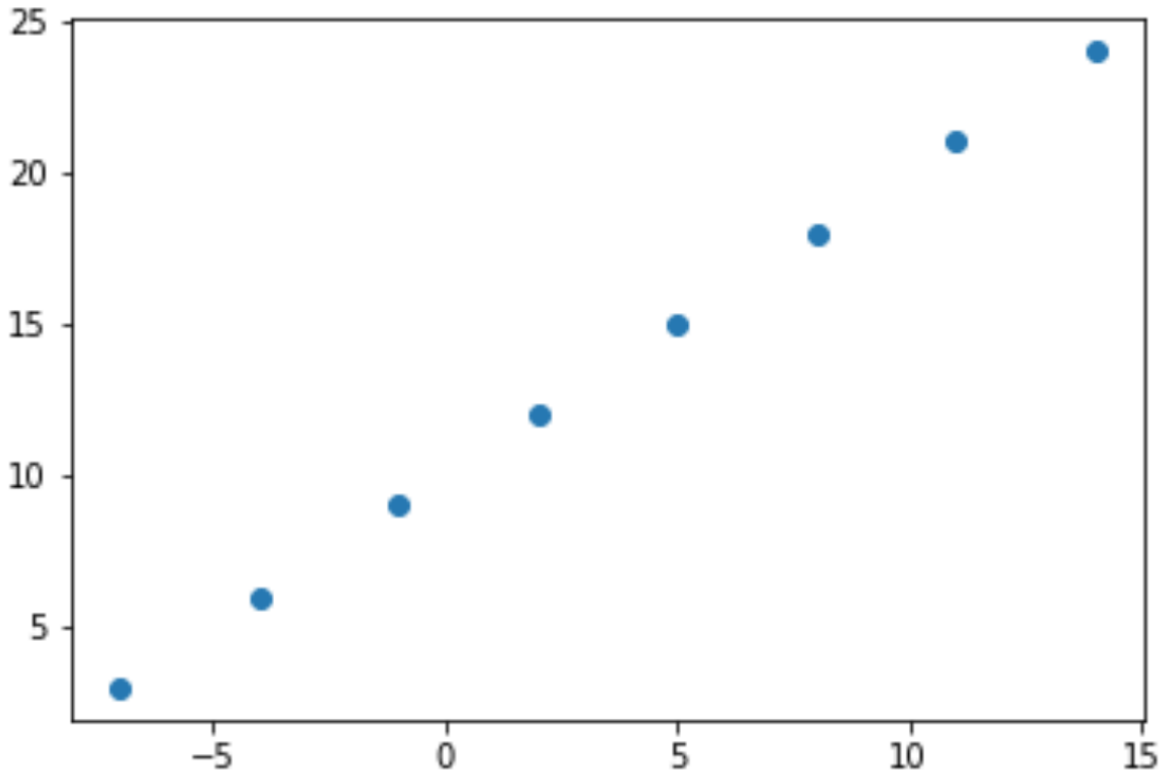
These will differ depending on the problem you're working on.

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

Creating a data and view (visualize) it

```
# Create a linear data

# Import libraries
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)
```



Our goal here will be to use `X` to predict `y`. So our **input** will be `X` and our **output** will be `y`

```
# 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)
```

This is 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.

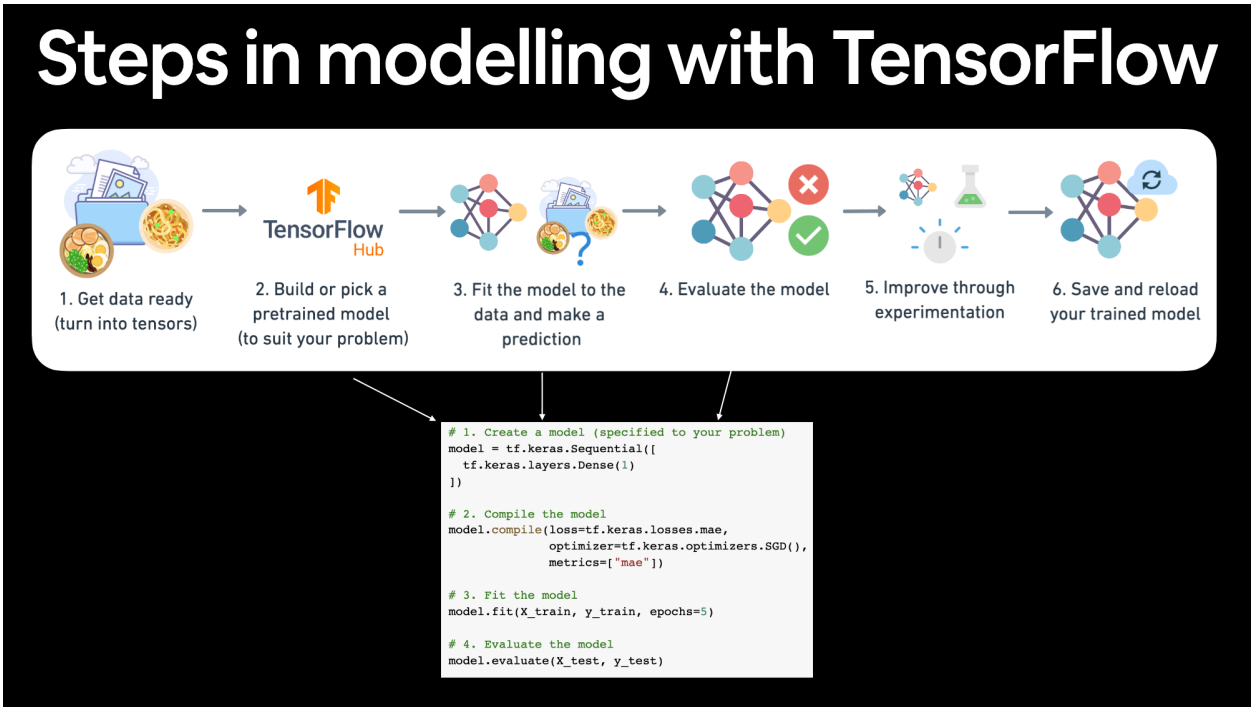
In our case, we're trying to build a model to predict the pattern between `X[0]` equalling `7.0` and `y[0]` equalling `3.0`.

Steps in Modelling with TensorFlow

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

1. Creating a model - define the input and output layers, as well as the hidden layers of a Deep Learning 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 - define the loss function (in other words, the function which tells our model how wrong it is) and the optimizer (tells our model how to improve the patterns its learning) and evaluation metircs (what we can use to interpret the performance of our model)
3. Fitting a model - letting the model try to find patterns between X & y (features and labels)



Keras Sequential API

To build our first TensorFlow regression model, we will use Keras Sequential API.

tf.keras.Sequential | TensorFlow Core v2.8.0
Sequential groups a linear stack of layers into a tf.keras.Model.
https://www.tensorflow.org/api_docs/python/tf/keras/Sequential

- Sequential groups a linear stack of layers into a tf.keras.Model.
- Sequential provides training and inference features on this model.

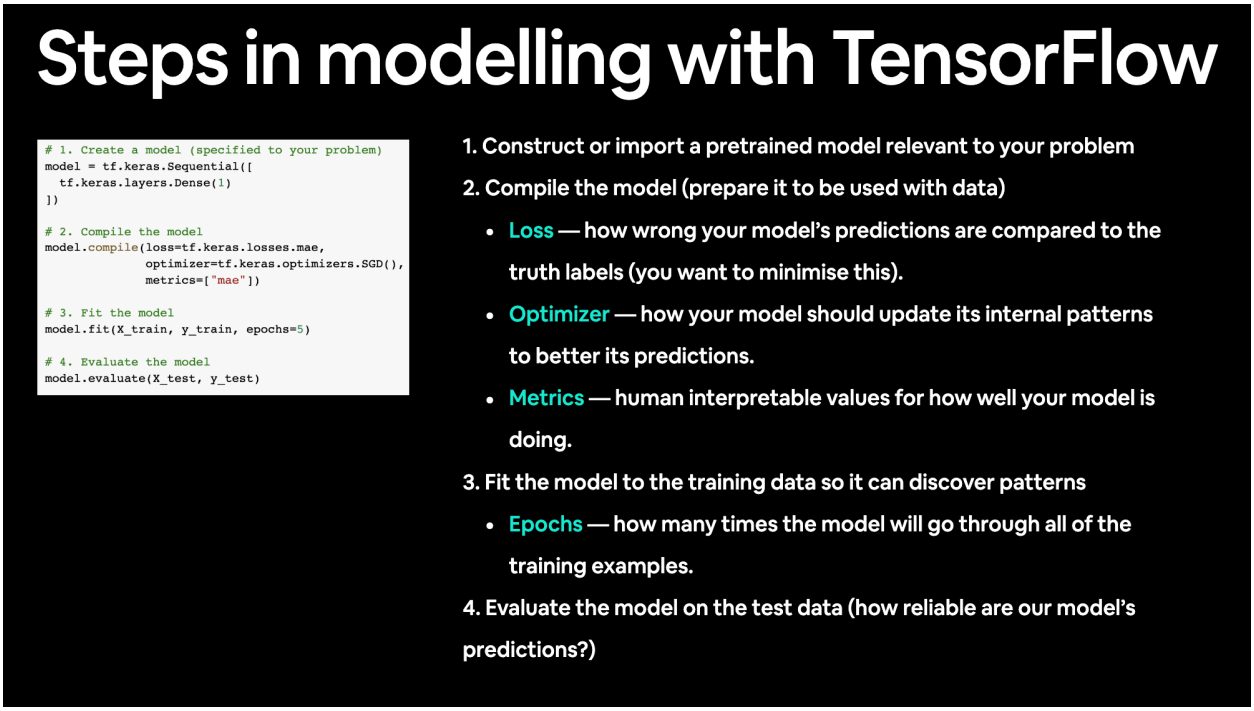
```
# Set random seed
tf.random.set_seed(42)

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

# 2. Compile the model
model.compile(loss=tf.keras.losses.mae, # MAE - Mean Absolute Error
              optimizer=tf.keras.optimizers.SGD(), # SGD - Stochastic Gradient Decent
              metrics=["mae"])

# 3. Fit the model
model.fit(tf.expand_dims(X,axis=-1), y, epochs=5)
```

```
# Try and make a prediction using our model
y_pred = model.predict([17.0])
y_pred
```

What's Keras? 🤔

Every time we write TensorFlow code, `keras` comes after `tf` (e.g. `tf.keras.layers.Dense()`)

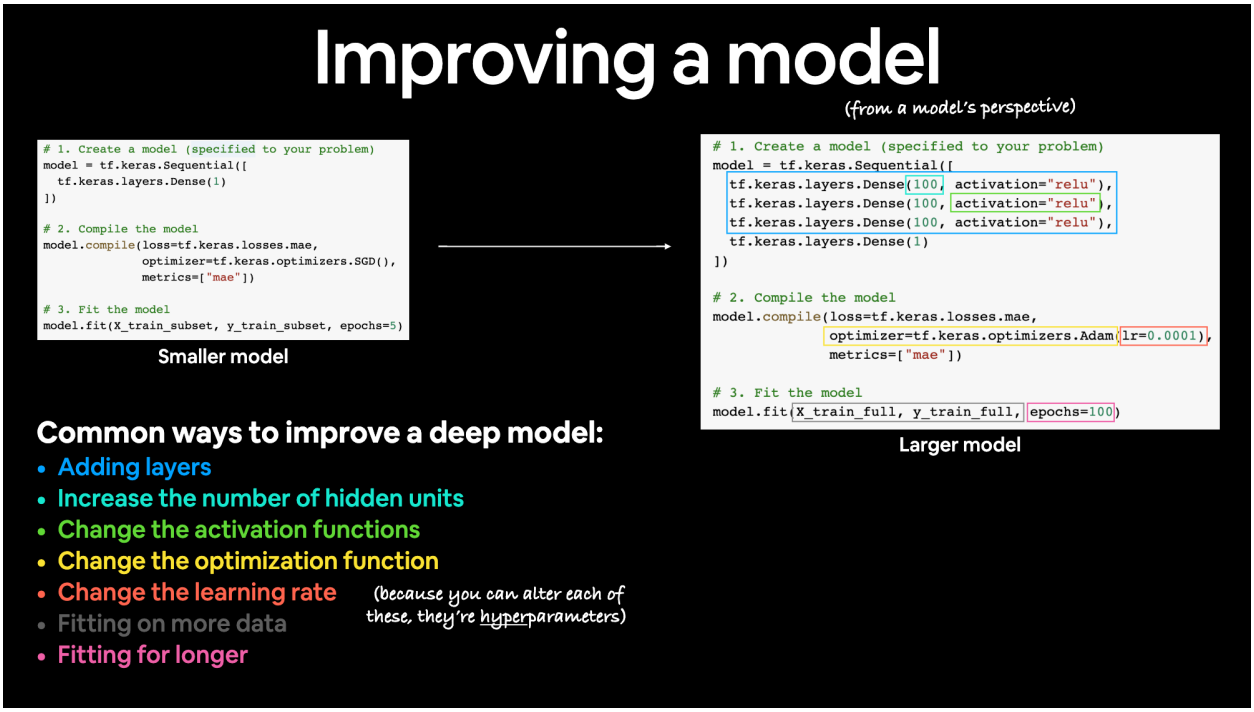
Before TensorFlow 2.0+, Keras was an API designed to be able to build deep learning models with ease. Since TensorFlow 2.0+, its functionality has been tightly integrated within the TensorFlow library.



Improving our model

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) and the practice of trying to find the best hyperparameters is referred to as hyperparameter tuning.

```
# Let's see if we can make another to improve our model
# Create a model (same as above)
model = tf.keras.Sequential([
  tf.keras.layers.Dense(100,activation="relu"),
  tf.keras.layers.Dense(1)
])

# Compile model (same as above)
model.compile(loss=tf.keras.losses.mae,
              optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              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 epochs not 10
```

Our resulted model performs really well on the training data. That means that our model overfitted on the training data, meaning that it learned the patterns to well from the training data. When it sees a new X, it's just relating back what it knows based on the training data and the error that it is producing is not really valid representation of what it is actually doing.

This code gives a better result

```
# Let's build another model
# Create a model (same as above)
model = tf.keras.Sequential([
  tf.keras.layers.Dense(50,activation=None), # Change to 50 units
  tf.keras.layers.Dense(1)
])

# Compile model (same as above)
model.compile(loss=tf.keras.losses.mae,
```

```
              optimizer=tf.keras.optimizers.Adam(lr=0.01), # Change to Adam optimizer and set learning_rate
              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 epochs not 10

# Let's try to make a prediction
model.predict([17.0])
```

How can we evaluate our model to ensure that it is working correctly on unseen data ?

Evaluating a model

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

```
Build a model -> fit it -> evaluate it -> tweak a model -> fit it -> evaluate it -> tweak a model -> fit it -> 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.

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)?

We'll create a little bit of a bigger dataset and a new model we can use

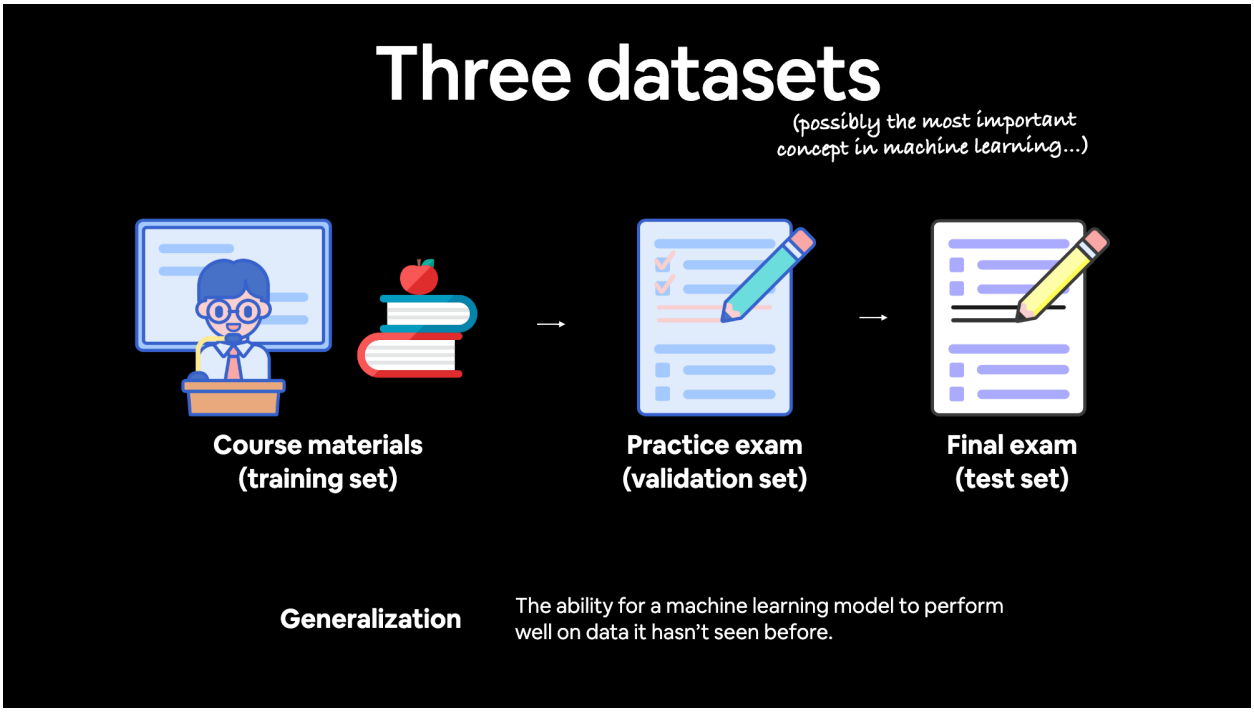
```
# Make a bigger dataset
X = tf.range(-100, 100, 4)
X

# Make labels for the dataset
y = X + 10 # This is the pattern for our model to learn
y
```

Split data into a training/validation/test sets

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).



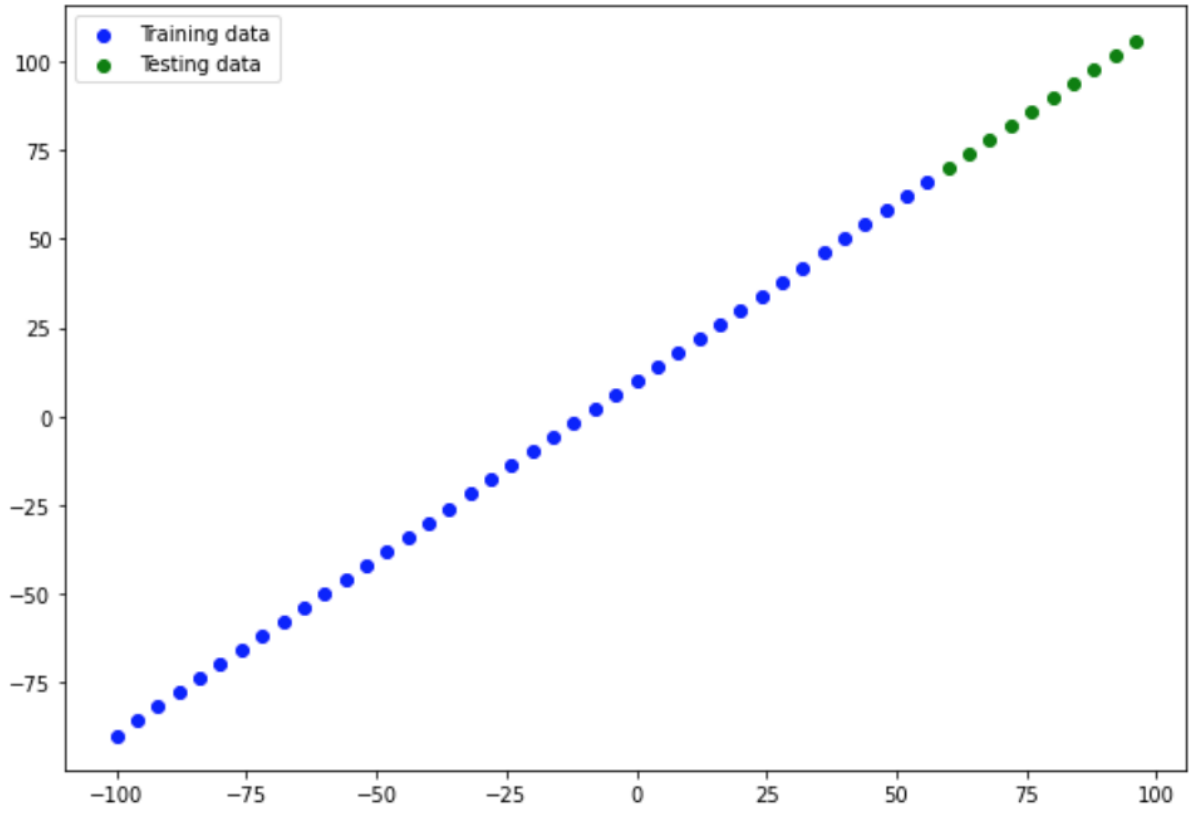
📝 **Note:** When dealing with real-world data, this step is typically done right at the start of a project (the test set should always be kept separate from all other data). We want our model to learn on training data and then evaluate it on test data to get an indication of how well it **generalizes**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

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

# Split the data into train and test sets
X_train = X[:40] # First 40 are training samples (80% of the data)
y_train = y[:40]
X_test = X[40:] # last 10 are testing samples (20% of the data)
y_test = y[40:]

len(X_train), len(X_test), len(y_train), len(y_test)
```

Visualize the data

```
plt.figure(figsize=(10,7))
# Plot traning 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 a legend
plt.legend();
```



Visualize model

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).

```
# Let's create a model which builds automatically by defining the input_shape argument in the first layer
tf.random.set_seed(42)

# 1. Create a model (same as above)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[1]) # The shape might be different depending on what shape of input data we are passing
])

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

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

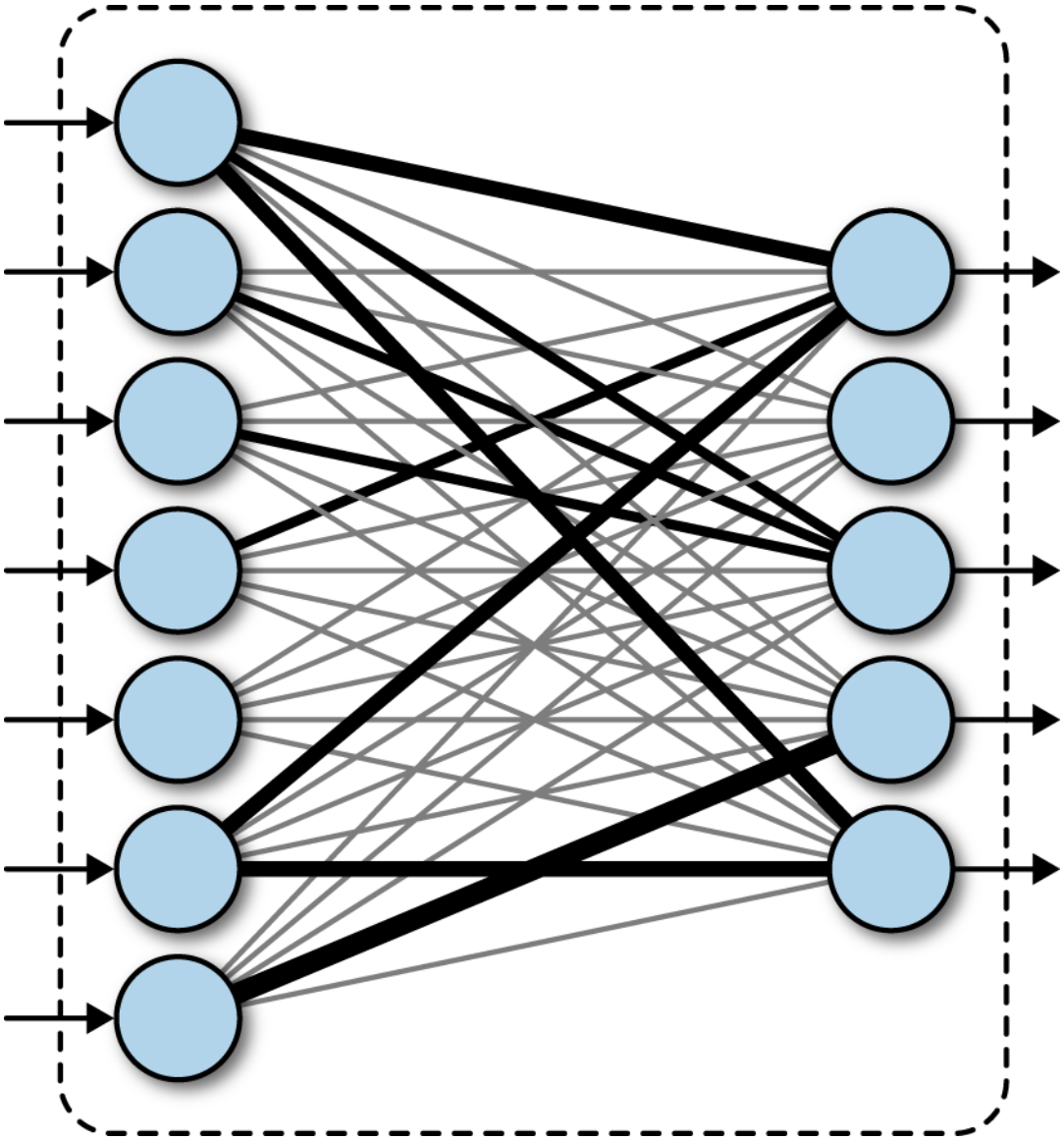
```
Model: "sequential_20"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_44 (Dense)            (None, 1)                 2

=================================================================
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 of each layer.

- **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**). When we import the model that is already learned patterns of the data, we might freeze those learned patterns.

🤔 What is a type of Dense ??

Another word for Dense is fully-connected. It refers to fully-connected layer. In TensorFlow, fully-connected layers is same as Dense layer.



Resources:

Fundamentals of Neural Networks on Weights & Biases

Training neural networks can be very confusing. What's a good learning rate? How many hidden layers should your network have? Is dropout actually useful? Why are your gradients vanishing? In this post we'll peel the curtain behind some of the more confusing aspects of neural nets, and help you make smart decisions about

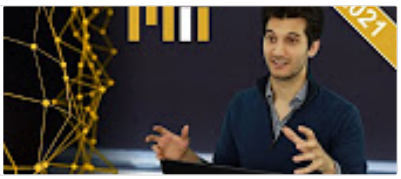
https://wandb.ai/site/articles/fundamentals-of-neural-networks

https://www.youtube.com/watch?v=njKP3FqW3Sk

MIT 6.S191: Introduction to Deep Learning

Course lectures for MIT Introduction to Deep Learning. http://introtodeeplearning.com

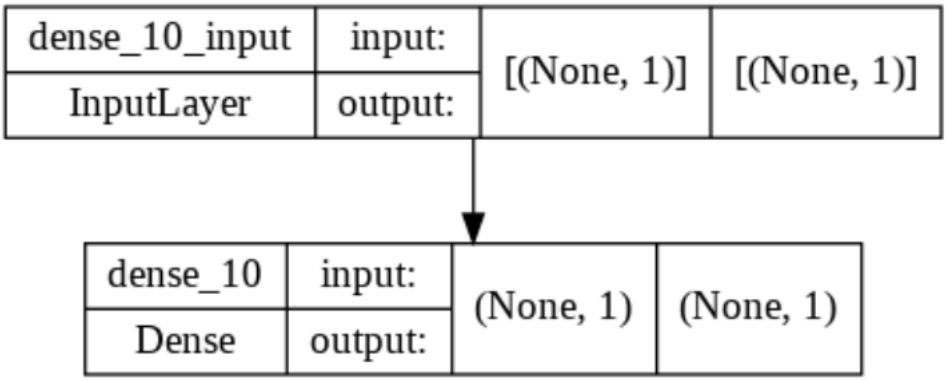
https://www.youtube.com/playlist?list=PLtBw6njQRU-rwp5__7C0oIVt26ZgjG9NI



When we are increasing the number of hidden units in the Dense (fully-connected) layer, the number of trainable parameters of the models is increased by 2. There are 2 trainable parameters per Dense hidden unit.

Another way of visualizing our model -> using TensorFlow plot function.

```
from tensorflow.keras.utils import plot_model
# Plot the model
plot_model(model=model, show_shapes=True)
```

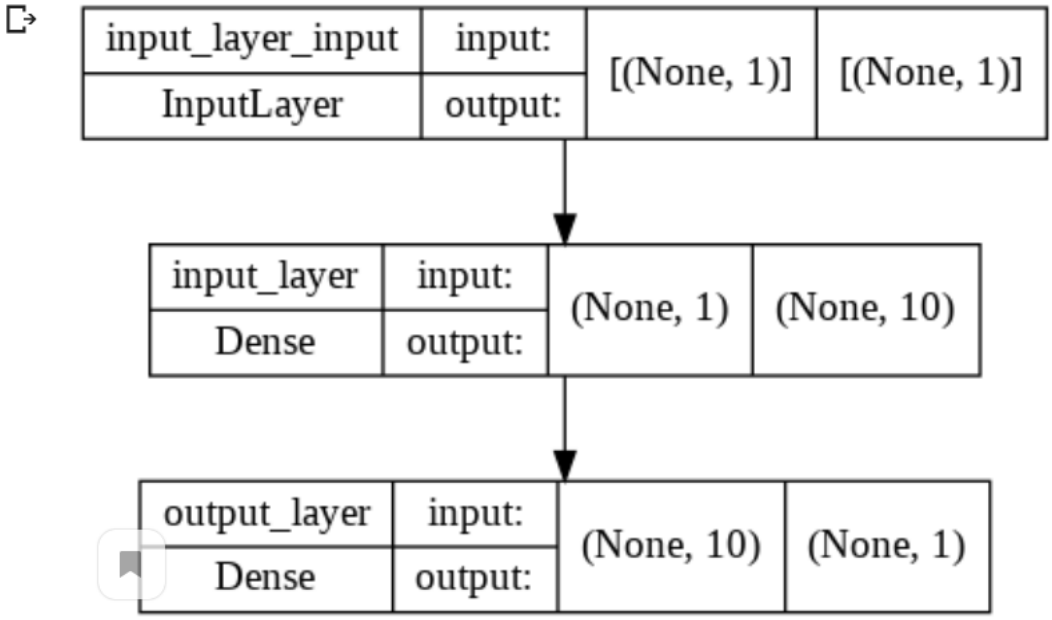


We can specify the names for:

- Model
- Hidden Layer (Dense / Fully-connected) layer

```
# Let's create a model which builds automatically by defining the input_shape argument in the first layer
tf.random.set_seed(42)

# 1. Create a model (same as above)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, input_shape=[1], name="input_layer"), # The shape might be different depending on what shape of input data we
    tf.keras.layers.Dense(1, input_shape=[1], name="output_layer")
],name="Models_1")

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

# Show model summary
model.summary()
```

```
Model: "Models name is specified..."
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 input_layer (Dense)          (None, 10)                20

 output_layer (Dense)         (None, 1)                 11

=================================================================
Total params: 31
Trainable params: 31
Non-trainable params: 0
_________________________________________________________________
```



Using TensorFlow plot function will be very helpful for debugging when we will visualize more complicated models.

Visualizing model's predictions

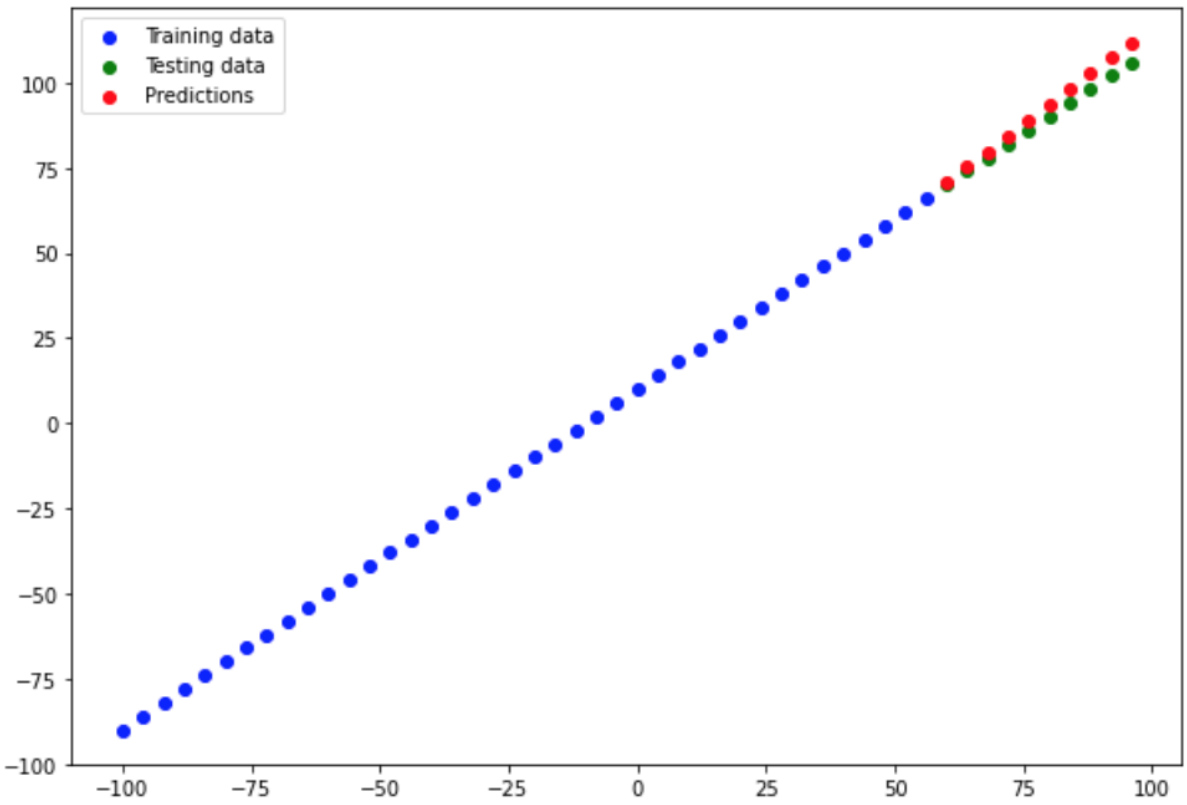
To visualize predictions, it's always a good practice to plot them against the ground truth labels.

Often we'll see this in the form of `y_test` vs. `y_pred` (ground truth vs. model's **predictions**).

```
# Make some predictions
y_pred = model.predict(X_test)
# Model's predictions
y_pred

# Ground truth
y_test
```

Note:
If you feel like you're going to reuse some kind of functionality in the future, it's a good idea to turn it into a function.

```
# Let's create a plotting function
def plot_predictions(train_data=X_train, train_labels=y_train, test_data=X_test, test_labels=y_test, predictions=y_pred):
  """
  Plots training, test data and compres predictions to ground truth labels
  """
  plt.figure(figsize=(10,7))
  # Plot training data in blue
  plt.scatter(train_data,train_labels,c="b", label="Training data")
  # Plot testing data in green
  plt.scatter(test_data, test_labels, c="g", label="Testing data")
  # Plot model's predictions in red
  plt.scatter(test_data, predictions, c="r",label="Predictions")
  # Show the legend
  plt.legend();


# Call the plot function
plot_predictions(train_data=X_train,
                 train_labels=y_train,
                 test_data=X_test,
                 test_labels=y_test,
                 predictions=y_pred)
```

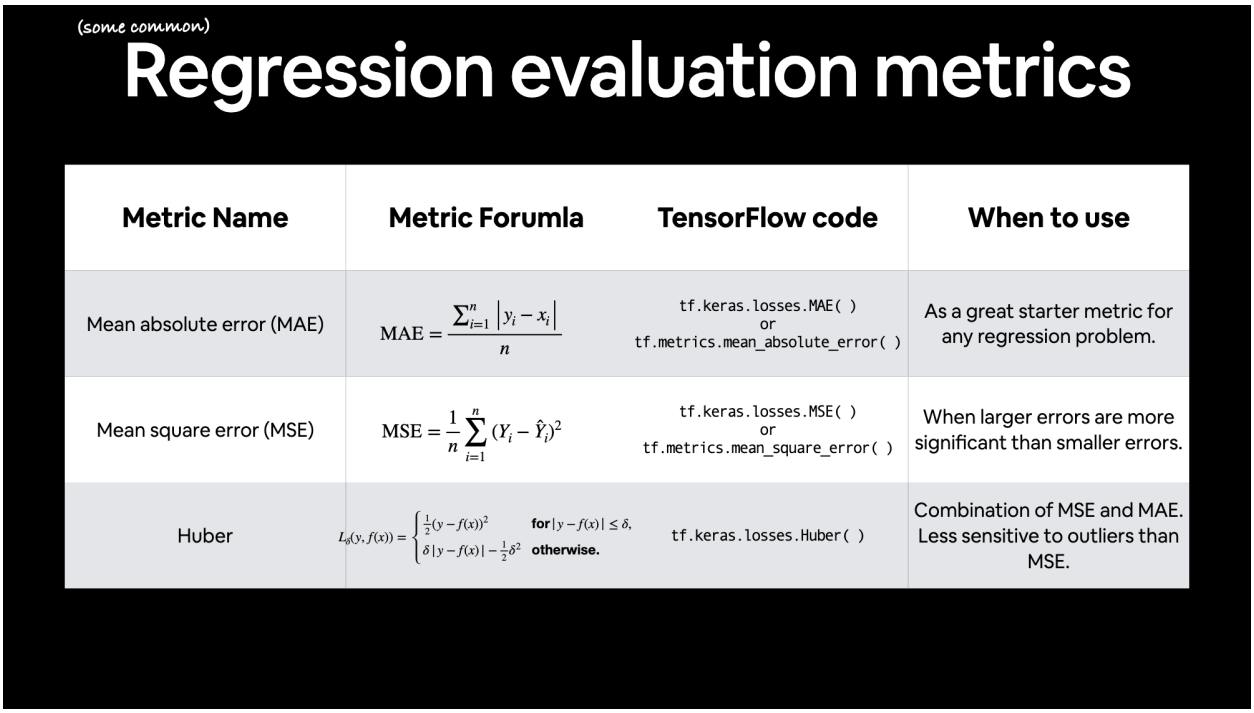


Evaluating our model's predictions with Regression Evaluation Metrics

Depending on the problem we're working on, there will be different evaluation metrics to evaluate our model's performance.

Since we're working on a regression, two of the main metrics:

- ***MAE**** - Mean Absolute Error, "on average, how wrong is each of my model's predictions"
- ***MSE**** - Mean Squared Error, "square the average errors".



Quick way to evaluate the model

```
# Evaluate the model on the test - Quick way
model.evaluate(X_test,y_test)
```

Calculate the mean absolute error

```
# Calculate the mean absolute error
tf.metrics.mean_absolute_error(y_true=y_test, y_pred=tf.constant(y_pred)) # We have got the test metrics for each prediction

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

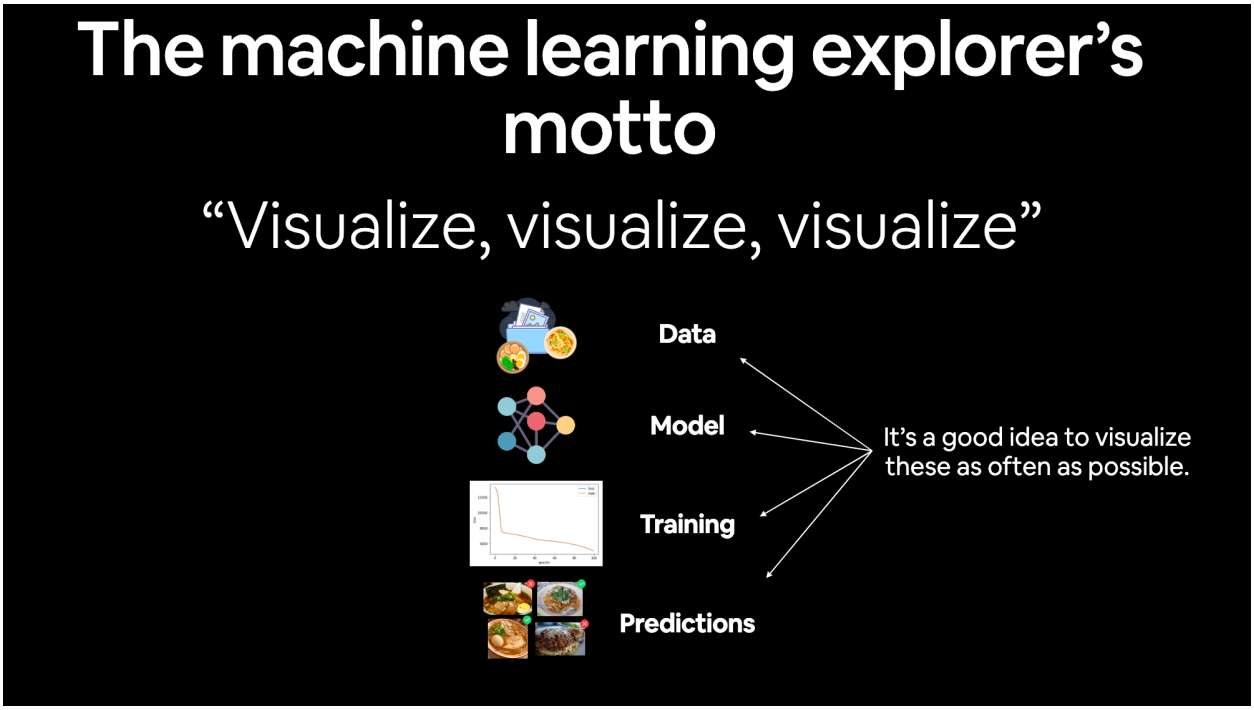
Calculate the mean squared error

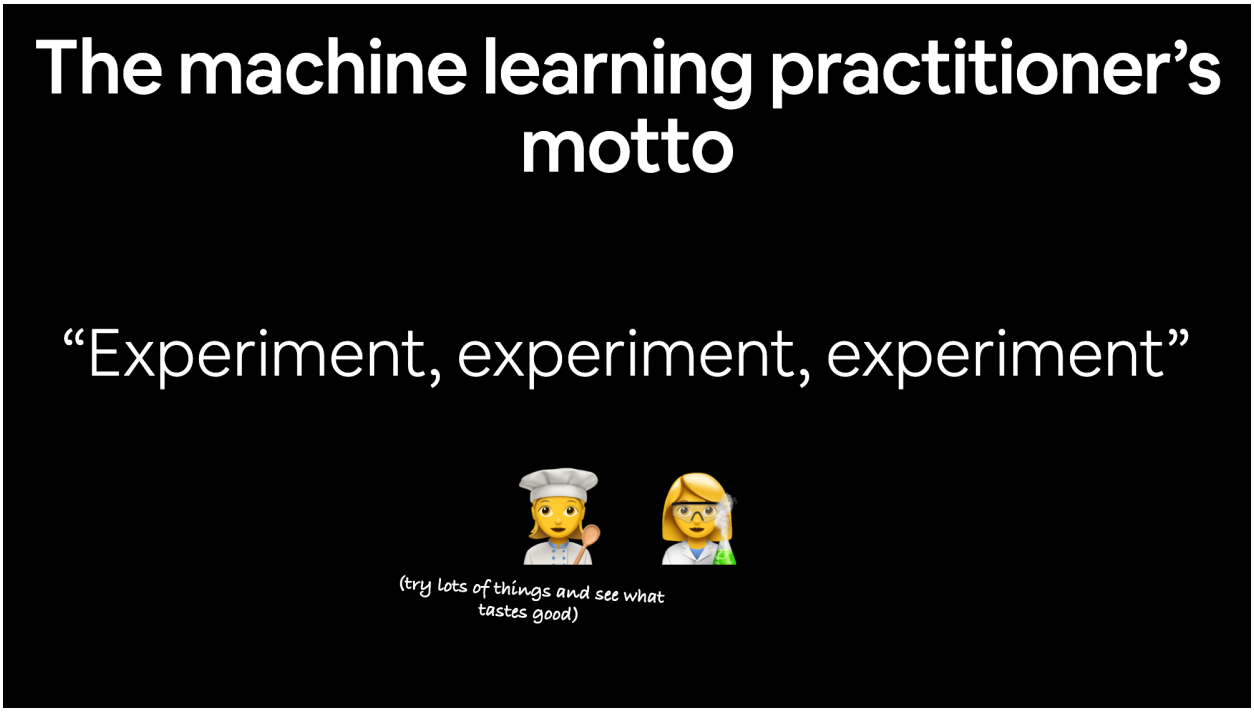
```
# Calculate the mean squared error
mse = tf.metrics.mean_squared_error(y_true=y_test, y_pred=tf.squeeze(y_pred))
mse
```

Reusability:

```
# Make some functions to reuse MAE and MSE

def mae(y_true,y_pred):
  return tf.metrics.mean_absolute_error(y_true=y_true,
                                        y_pred=y_pred)

def mse(y_true,y_pred):
  return tf.metrics.mean_squared_error(y_true=y_true,
                                       y_pred=y_pred)
```

Running Experiments to improve our Model






```
Build a model -> fit it -> evaluate it -> tweak a model -> fit it -> evaluate it -> tweak a model -> fit it -> evaluate it -> .....
```

Ways to improve our model:

1. Get more data - get more examples for your model to train on (more opportunities to learn patterns or relationships between featrues and labels).
2. Make your model larger (using 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 patterns in the data.

📝 **Note**:
When you are running modelling experiments, start with a baseline model, and then change one of the parameters for the next experiment, do the same for the next experiment and so on.

You want to start with small experiments (small models) and make sure they work and then increase their scale when necessary.

Let's do 3 modelling experiments:

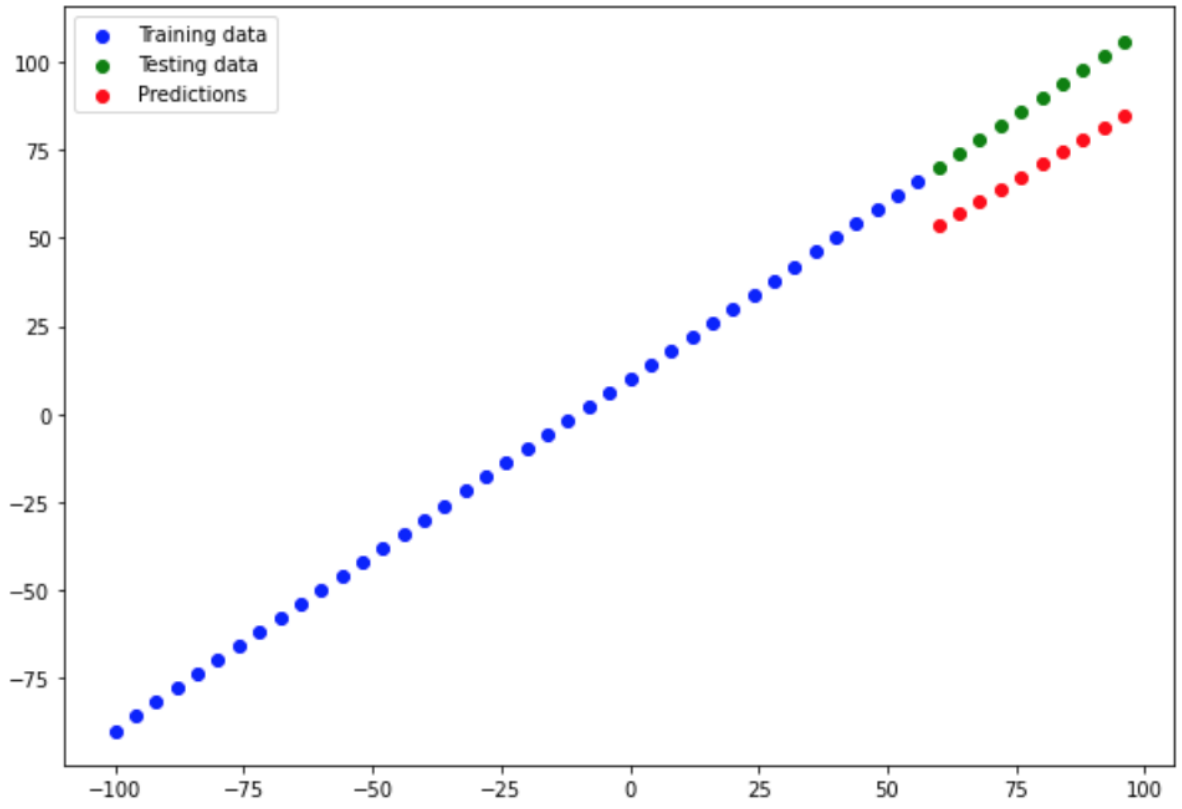
1. `model_1` - same as the original model: 1 layers, trained for 100 epochs.
2. `model_2` - 2 layers, trained for 100 epochs.
3. `model_3` - 2 layers, trained for 500 epochs.

Build `model_1`

- Sames as the original model: 1 layer, trained for 100 epochs

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

# 1. Create the model
model_1 = tf.keras.Sequential([
    tf.keras.layers.Dense(1)
])

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

# 3. Fit the model
model_1.fit(tf.expand_dims(X_train,axis=-1), y_train, epochs=100)
```

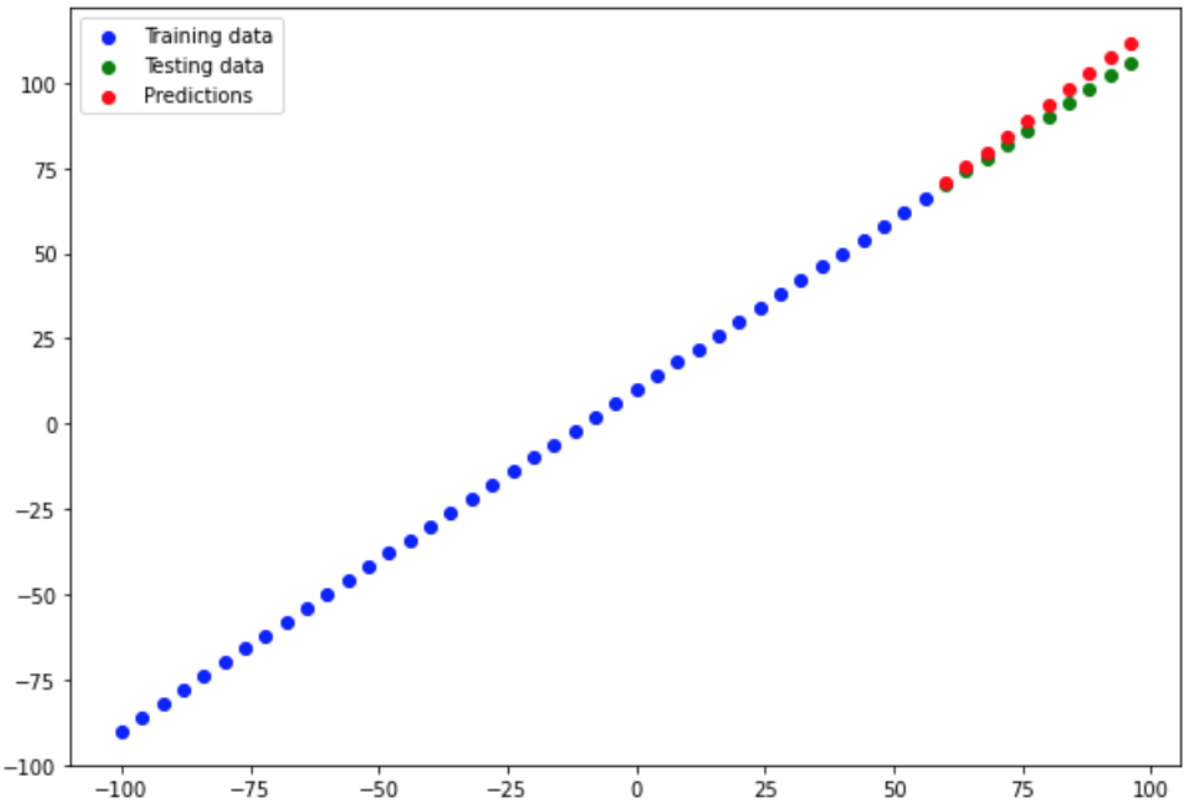
```
# Make predictions
y_pred_1 = model_1.predict(X_test)
# Plot predictions
plot_predictions(predictions=y_pred_1)

# Calculate model_1 evaluation metrics
mae_1 = mae(y_test,y_pred_1)
mse_1 = mse(y_test,y_pred_1)
mae_1, mse_1
```



Build `model_2`

- 2 dense layers, trained for 100 epochs.

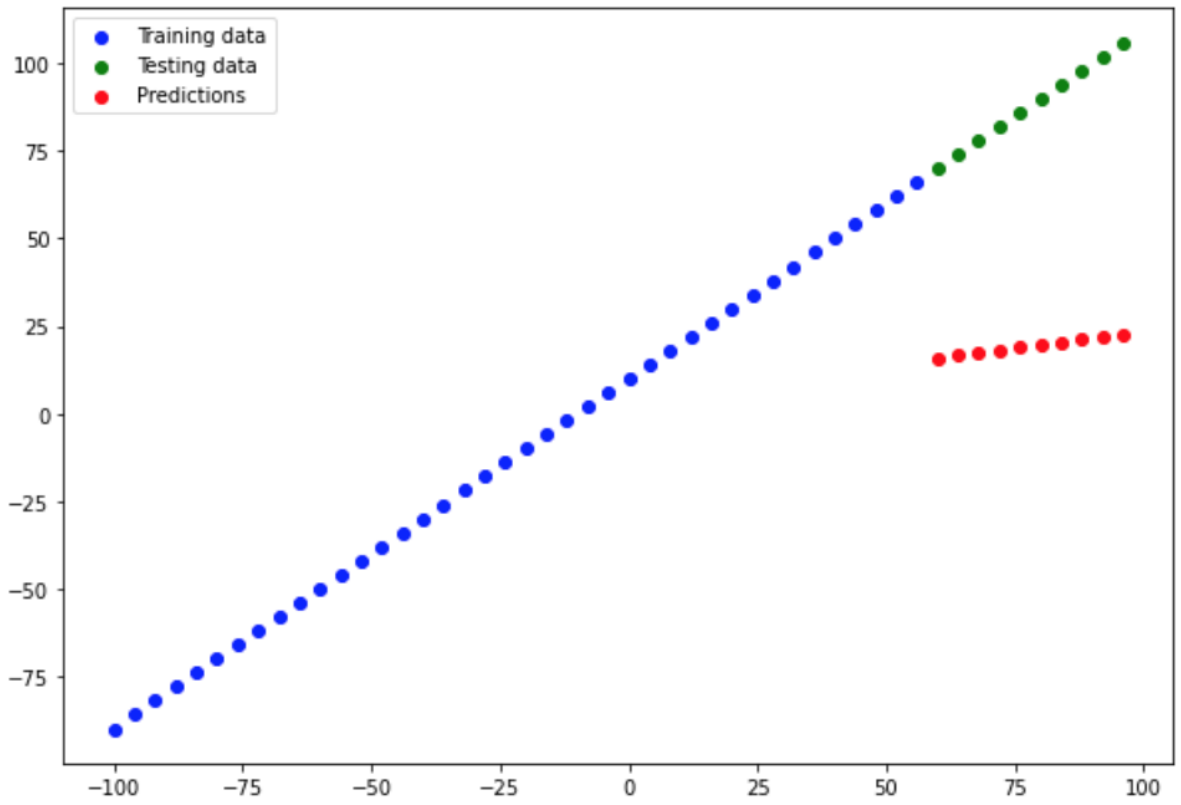
```
# Set random seed
tf.random.set_seed(42)

# 1. Create the model
model_2 = tf.keras.Sequential([
    tf.keras.layers.Dense(10),
    tf.keras.layers.Dense(1)
])

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

# 3. Fit the model
model_2.fit(tf.expand_dims(X_train,axis=-1), y_train, epochs=100)

# Make predictions
y_pred_2 = model_2.predict(X_test)
# Plot predictions
plot_predictions(predictions=y_pred_2)

# Calculate model_2 evaluation metrics
mae_2 = mae(y_test,y_pred_2)
mse_2 = mse(y_test,y_pred_2)
mae_2, mse_2
```



Build `model_3`

- 2 dense layers, trained for 500 epochs.

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

# 1. Create the model
model_3 = tf.keras.Sequential([
    tf.keras.layers.Dense(10),
    tf.keras.layers.Dense(1)
])

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

# 3. Fit the model
model_3.fit(tf.expand_dims(X_train,axis=-1), y_train, epochs=500)

# Make predictions
y_pred_3 = model_3.predict(X_test)
# Plot predictions
plot_predictions(predictions=y_pred_3)

# Calculate model_2 evaluation metrics
mae_3 = mae(y_test,y_pred_3)
mse_3 = mse(y_test,y_pred_3)
mae_3, mse_3
```



Comparing the results of our models

Let's compare our model's results using Pandas DataFrame

```
import pandas as pd

model_results = [["model_1", mae_1.numpy(),mse_1.numpy()],
                 ["model_2",mae_2.numpy(),mse_2.numpy()],
                 ["model_3",mae_3.numpy(),mse_3.numpy()]]
all_results = pd.DataFrame(model_results,columns=["model","mae","mse"])
all_results
```

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 our 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"

Tracking the experiments

One really good habit in Machine Learning Modelling is to track the results of your experiments.

And when doing so, it can be tedious if you're running lots of experiments.

Luckily, there are tools to help us!

Resources: As you build more models, you'll want to look into using:

- TensorBoard - a component of the TensorFlow Library to help track modelling experiments
- Weights & Biases - a tool for tracking all kinds of Machine Learning Experiemnts (plugs straight into TensorFlow)

Saving our model

Saving our models allows us to use them outside of Google Colab (or wherever they were trained) such as in:

- Web Application
- Mobile App

Save and load models | TensorFlow Core

Model progress can be saved during and after training. This means a model can resume where it left off and avoid long training times. Saving also means you can share your model and others can recreate your work.

https://www.tensorflow.org/tutorials/keras/save_and_load

In TensorFlow, there are 2 formats of saved models:

1. SavedModel format: The SavedModel format is another way to serialize models. Models saved in this format can be restored using `tf.keras.models.load_model` and are compatible with TensorFlow Serving.
2. HDF5 format: Keras provides a basic save format using the `HDF5` standard.

Hierarchical Data Format - Wikipedia

Hierarchical Data Format (HDF) is a set of file formats (HDF4, HDF5) designed to store and organize large amounts of data. Originally developed at the National Center for Supercomputing Applications, it is supported by The HDF Group, a non-profit corporation whose mission is to ensure continued development of HDF5 technologies

https://en.wikipedia.org/wiki/Hierarchical_Data_Format

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

# Save model using the HDF5 format
model_2.save('best_model_HDF5_format.h5')
```

Loading in saved model

```
# Load in the SavedModel format model
loaded_SavedModel_format = tf.keras.models.load_model("best_model_SavedModel_format")
loaded_SavedModel_format.summary()

# Compare model_2 predictions with SavedModel format model predictions
model_2_pred = model_2.predict(X_test)
loaded_SavedModel_format_pred = loaded_SavedModel_format.predict(X_test)
model_2_pred == loaded_SavedModel_format_pred
mae(y_true=y_test,y_pred=model_2_pred) == mae(y_true=y_test, y_pred=loaded_SavedModel_format_pred)

# Compare model_2 predictions with SavedModel format model predictions
model_2_pred = model_2.predict(X_test)
loaded_hdf5_format_pred = loaded_hdf5_format.predict(X_test)
model_2_pred == loaded_hdf5_format_pred
mae(y_true=y_test,y_pred=model_2_pred) == mae(y_true=y_test, y_pred=loaded_hdf5_format_pred)
```

How long I should train my model ? 🤔

It depends. However, TensorFlow has a solution - Early Stopping.

tf.keras.callbacks.EarlyStopping | TensorFlow Core v2.8.0

Stop training when a monitored metric has stopped improving.

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

We can add this component to our model to stop training once the models stops improving the specified metric.

Feature Scaling

Feature scaling

Scaling type	What it does	Scikit-Learn Function	When to use
Scale (also referred to as normalisation)	Converts all values to between 0 and 1 whilst preserving the original distribution.	`MinMaxScaler`	Use as default scaler with neural networks.
Standardization	Removes the mean and divides each value by the standard deviation.	`StandardScaler`	Transform a feature to have close to normal distribution (caution: this reduces the effect of outliers).

Source: Adapted from Jeff Hale's Scale, Standardize, or Normalize with Scikit-Learn article.

- **Normalization** is a technique often applied as part of data preparation for machine learning. The goal of normalization is to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values.
- **Standardization** is the process of rescaling the attributes so that they have mean as 0 and variance as 1. The ultimate goal to perform standardization is to bring down all the features to a common scale without distorting the differences in the range of the values.

💡 In terms of scaling values, neural networks tend to prefer normalization. If you are not sure on which to use, you could try both and see which performs better.