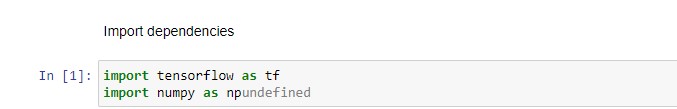
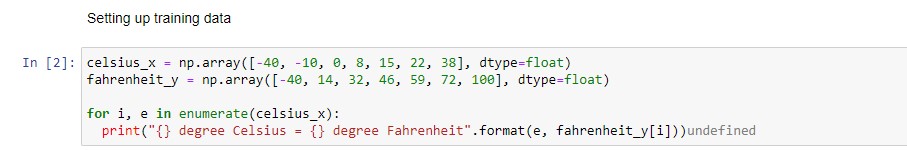
**Import dependencies**

First import TensorFlow. Here we are calling tf for ease of use. We also tell it to only display errors. Next, import NumPy as np that helps us to represent our data as highly performant lists.



**Set up training data**:

Supervised Machine Learning is all about figuring out an algorithm given a set of inputs and outputs. since the task in this colab is to create a model that can give the temperature in Fahrenheit when given the degree in Celsius, we create two lists celsius\_x and fahrenheit\_y that we can use to train our model.



**Some Machine Learning Terminology**

Feature: The input(s) to our model. In this case, a single value - the degree in Celsius. Labels: The output our model predicts. In this case, a single value - the degree in Fahrenheit.

**Example:** A pair of i/o used during training. In our case a pair of values from celsius\_x and farhenheit\_y at a specific index, such as (22, 72).

**Create the model:**

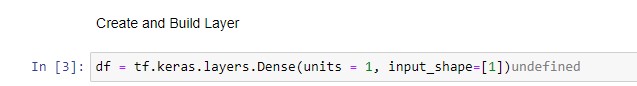
Next create the model. We will use simplest possible model we can, a Dense network. Since the problem is straightforward, this network will require only a single layer, with a single neuron.

Build a layer

We'll call the layer 10 and create it by instantiating tf. keras.layers.Dense with the following configuration:

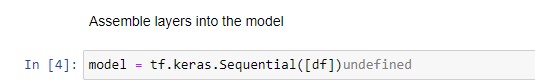
**input\_shape** = [1] - This specifies that the input to this layer is a single value. That is, the shape is one-dimensional array with one member. Since this is the first (and only) layer, that input shape is the input shape of the entire model. The single value is a floating-point number, representing degrees Celsius.

**units** =1 - This specifies the number of neurons in the layer. The number of neurons defines how many internal variables the layer has to try to learn how to solve the problem (more later). Since this is the final layer, it is also the size of the model's output - a single float value representing degrees Fahrenheit. (In a multi-layered network, the size and shape of the later would need to match the input\_shape of the next layer.)



**Assemble layers into the model:**

Once layers are defined, they need to be assembled into a model. The Sequential model definition takes a list of layers as argument, specifying the calculation order from the input to the output. This model has just a single layer.

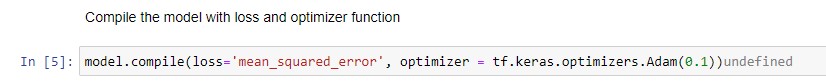


**Compile the model, with loss and optimizer functions**:

before training, the model has to be compiled. When compiled for training, the model is given:

**Loss function**: A way of measuring how far off predictions are from the desired outcome. (The measure difference is called the "loss")

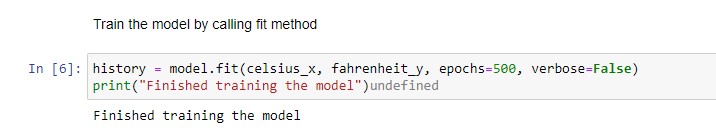
**Optimizer function**: A way of adjusting internal values in order to reduce the loss.



There are used during training (model.fit(), below) to first calculate the loss at each point, and then improve it. In fact, the act of calculating the current loss of a model and then improving it is precisely what training is. During training, the optimizer function is used to calculate adjustments to the model's internal variables. The goal is to adjust the internal variables until the model (which really a math function) mirrors the actual equation for converting Celsius to Fahrenheit. TensorFlow uses numerical analyses to perform this tuning and all this complexity is hidden from you so we will not go into the details here. what is useful to know about these parameters are: The loss function (mean squared error) and the optimizer (Adam) used here are standard for simple models like this one, but many others are available. One part of the Optimizer we may need to think about when building our own models is the learning rate (0.1 in the above code). This is the step size taken when adjusting values in the model. Fif the value is too small; it will take too many iterations to train the model. Too large, and accuracy goes down. Finding a good value often involves some trial and error, but the range is usually within 0.001 (default), and 0.1

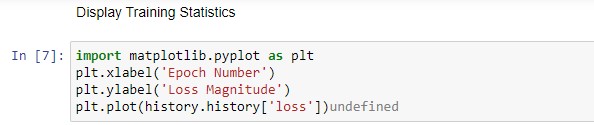
**Train the model trains the model by calling the fit method**.

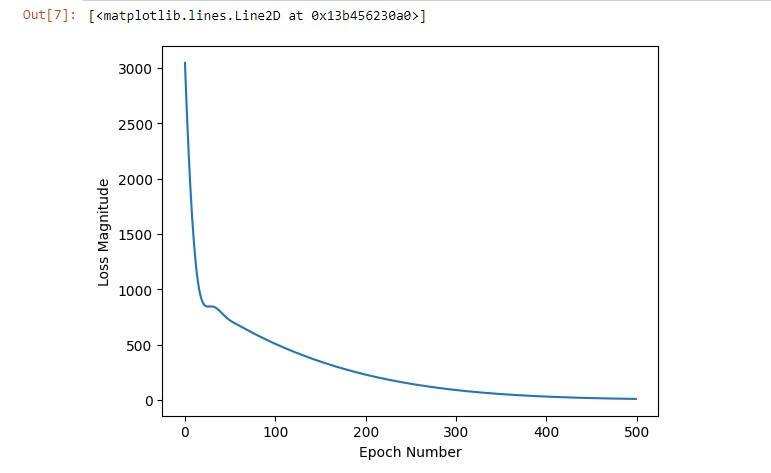
During training the model takes in Celsius values, performs a calculation using the current internal variables (called "weight") and outputs values which are meant to be the Fahrenheit equivalent. Since the weights are initially set randomly, the output will not be close to the correct value. The difference between the actual input and the desired output is calculated using the loss function and the optimizer function directs how the weights should be adjusted. This cycle of calculate, compare, adjust is controlled by the fit method. The first arguments are the inputs, the second argument is the desired outputs. The epochs argument specifies how many times this cycle should be run and the verbose argument controls how much output the method produces.



**Display Training Statistics:**

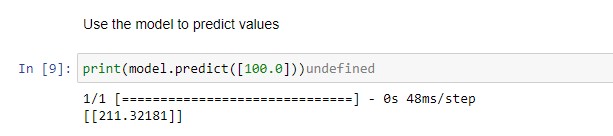
The fit method returns a history object. We can use this object to plot how the loss of our model goes down after each training epoch. A high less means that the Fahrenheit degrees the model predicts is far from the corresponding value in fahrenheit\_y. We'll use Matplotlib to visualize this. As we can see, our model improves very quickly at first and then has a steady, slow improvement until it is very near perfect towards the end.





**Use the model to predict values**:

Now we have a model that has been trained to lean the relationship between celsius\_x and fahrenheit\_y. We can use the predict method to have it calculate the Fahrenheit degrees for a previously unknown Celsius degrees. So, for instance, if the Celsius value is 200, what do you think the Fahrenheit will be?



The correct answer is 100 \* 1.8 + 32 = 212, so our **model is doing really well**.

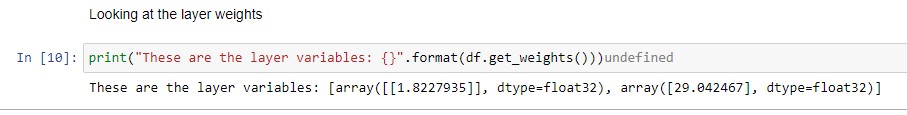
**To review**:

* we created a model with a Dense layer
* we trained it with 3500 examples (7 pairs, over 500 epochs).

Our model tuned the variables (weights) in the Dense layer until it was able to return the correct Fahrenheit value for any Celsius value.

**Looking at the layer weights:**

Finally, let's print the internal variables of the Dense layer.

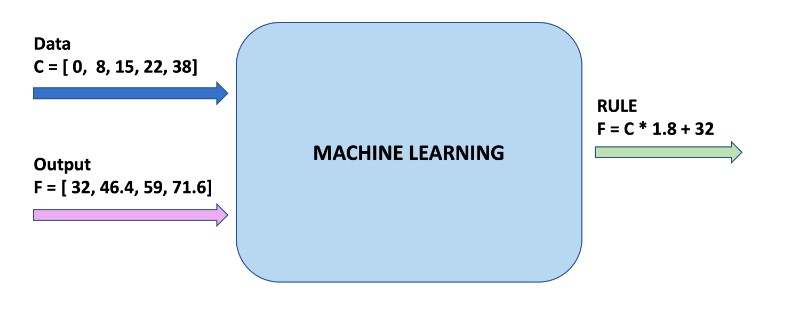


The first variable is close to ~1.8 and the second to ~32. These values are the actual variables in the real conversion formula. This is really close to the values in the conversion formula. We show how dense layer works, but for a single neuron with a single input and a single output, the internal math looks the same as the equation for a line, y = mx + b, which has the same form as the conversion equation:

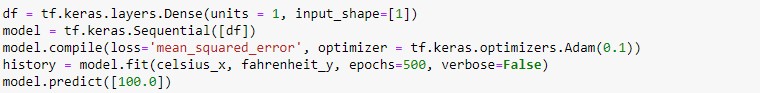
**f = 1.8c + 32**.

**Recap:**

We just trained our Machine Learning model. We saw that by training the model with input data and the corresponding output, the model learned to multiply the input by 1.8 and then add 32 to get the correct result.



This was really impressive considering that we only needed a few lines code:



This example is the general plan for of any Machine Learning program. We will use the same structure to create and train our neural network and use it to make predictions.

**The Training Process:**

The training process (happening in model.fit(...)) is really about tuning the internal variables of the networks to the best possible values, so that they can map the input to the output. This is achieved through an optimization process called **Gradient Descent**, which uses Numeric Analysis to find the best possible values to the internal variables of the model.

To do Machine Learning, we don't really need to understand these details. But for the curious: gradient descent **iteratively adjusts parameters**, **nudging them in the correct direction a bit at a time until they reach the best values**. In this case “best values” means that nudging them any more would make the model perform worse. The function that measures how good or bad the model is during each iteration is called the “**loss function**”, and the goal of each nudge is to “**minimize the loss function**.”

**The training process starts with a forward pass, where the input data is fed to the neural network (see Fig.1). Then the model applies its internal math on the input and internal variables to predict an answer** ("Model Predicts a Value" in Fig. 1).

In our example, the input was the degrees in Celsius, and the model predicted the corresponding degrees in Fahrenheit.

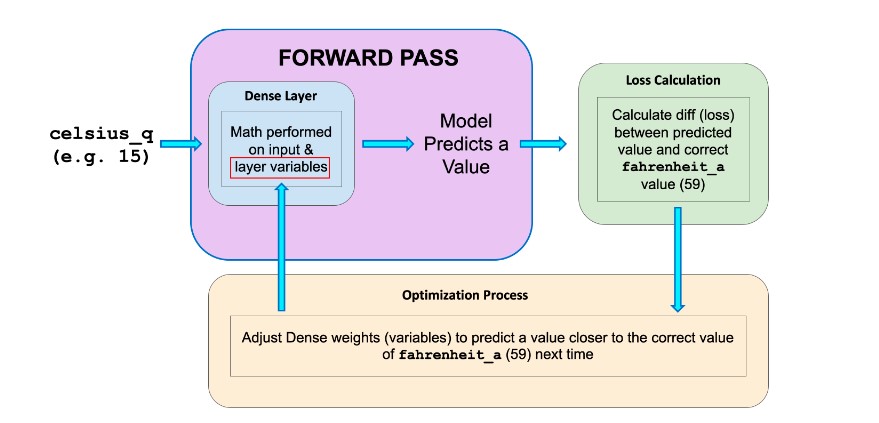


Fig: 1 Forward Pass

Once a value is predicted, the difference between that predicted value and the correct value is calculated. This difference is called the **loss**, and it's a measure of how well the model performed the mapping task. The value of the loss is calculated using a loss function, which we specified with the loss parameter when calling model.compile().

After the loss is calculated, the internal variables (weights and biases) of all the layers of the neural network are adjusted, so as to minimize this loss — i.e., to make the output value closer to the correct value (see Fig. 2).

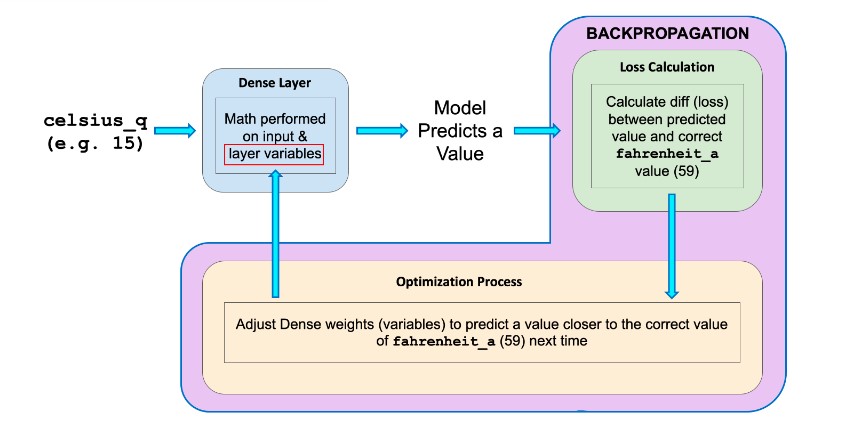


Fig: 2 Backpropagation

This optimization process is called Gradient Descent. The specific algorithm used to calculate the new value of each internal variable is specified by the optimizer parameter when calling model.compile(...). In this example we used the **Adam** optimizer.

By now we should know what the following terms are:

**Feature**: The input(s) to our model

**Examples**: An input/output pair used for training

**Labels**: The output of the model

**Layer**: A collection of nodes connected together within a neural network.

**Model**: The representation of your neural network

**Dense and Fully Connected (FC)**: Each node in one layer is connected to each node in the previous layer.

**Weights and biases**: The internal variables of model

**Loss**: The discrepancy between the desired output and the actual output

**MSE**: Mean squared error, a type of loss function that counts a small number of large discrepancies as worse than a large number of small ones.

**Gradient Descent**: An algorithm that changes the internal variables a bit at a time to gradually reduce the loss function.

**Optimizer**: A specific implementation of the gradient descent algorithm. (There are many algorithms for this. We will only use the “**Adam**” Optimizer, which stands for ADAptive with Momentum. It is considered the best-practice optimizer.)

**Learning rate**: The “step size” for loss improvement during gradient descent.

**Batch**: The set of examples used during training of the neural network

**Epoch**: A full pass over the entire training dataset

**Forward pass**: The computation of output values from input

**Backward pass (backpropagation)**: The calculation of internal variable adjustments according to the optimizer algorithm, starting from the output layer and working back through each layer to the input.