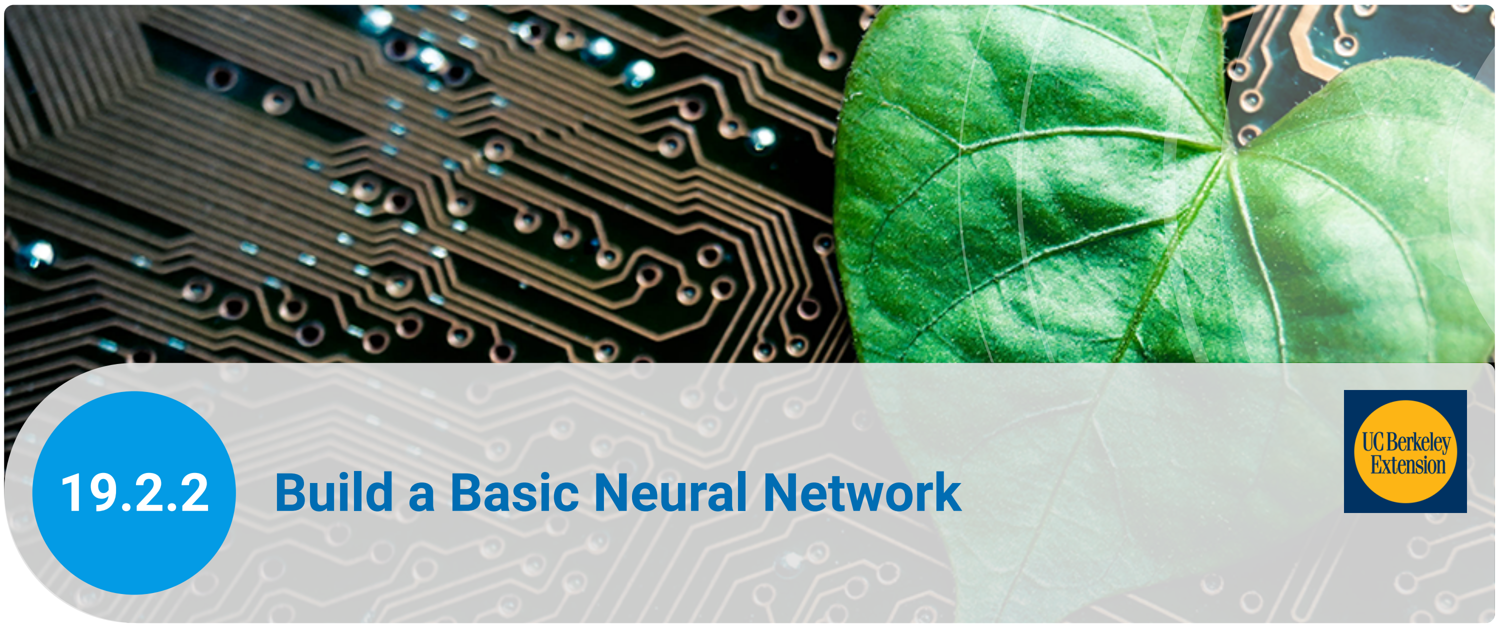


19.2.2 Build a Basic Neural Network

Beks knows she'll want to work with the foundation's datasets once she starts working on her analysis, but while she's still exploring concepts, she wants some dummy data that she can use to train and test.

Now that we have our TensorFlow library installed and understand some classes behind the Keras module, it is time to start implementing. In addition to building and modelling the neural network, we'll need to build a dummy dataset for training and testing. In a new Jupyter Notebook, first we'll import our required libraries by entering the following code:

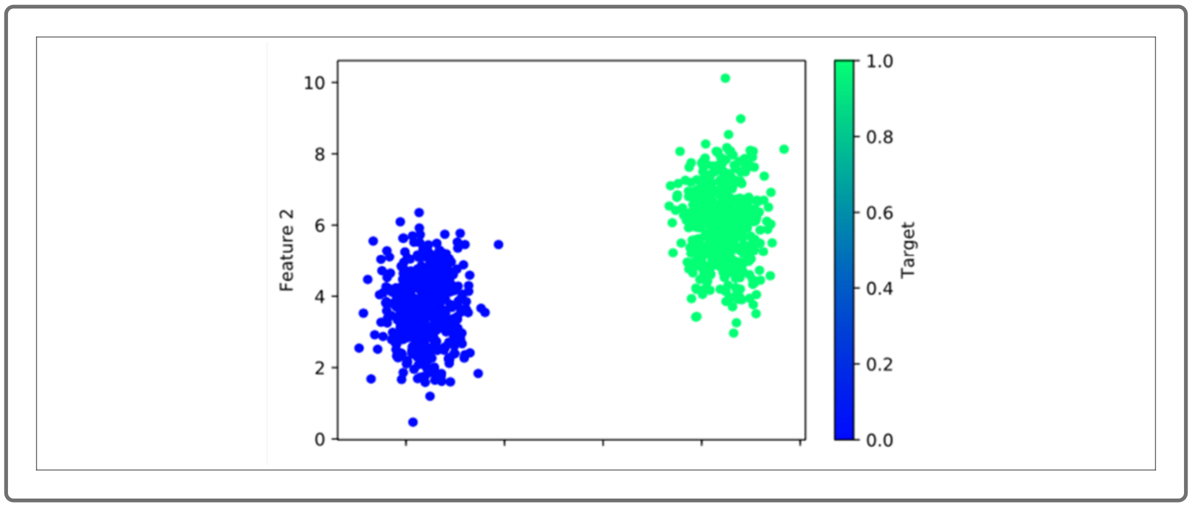
```
# Import our dependencies
import pandas as pd
import matplotlib as plt
from sklearn.datasets import make_blobs
import sklearn as skl
import tensorflow as tf
```

Once we have our required libraries imported into our notebook, we can create the dummy data using Scikit-learn's `make_blobs` method. The `make_blobs` is used to create sample values and contains many parameters that change the shape and values of the sample dataset. For our purposes, we'll use the `make_blobs` method to create 1,000 samples with two features (also known as our x- and y-axis values) that are linearly separable into two groups. In our notebook, we can generate and visualize our dummy data using the following code:

```
nerate dummy dataset
 = make_blobs(n_samples=1000, centers=2, n_features=2, random_state=78)

eating a DataFrame with the dummy data
 pd.DataFrame(X, columns=["Feature 1", "Feature 2"])
Target"] = y

otting the dummy data
lot.scatter(x="Feature 1", y="Feature 2", c="Target", colormap="winter"
```



Once we have our dummy data generated, we'll split our data into training and test datasets using Scikit-learn's `train_test_split` method. In our notebook, enter the following code to generate the training and test datasets:

```
se sklearn to split dataset
m sklearn.model_selection import train_test_split
rain, X_test, y_train, y_test = train_test_split(X, y, random_state=78
```

Now that we have our training data, we need to prepare the dataset for our neural network model. As with any machine learning algorithm, it is crucial to normalize or standardize our numerical variables to ensure that our neural network does not focus on outliers and can apply proper weights to each input. In most cases, the more that input variables are normalized to the same scale, the more stable the neural network model is, and the better the neural network model will generalize. To normalize our dummy data, we'll add and run the following code to the notebook:

```
# Create scaler instance
X_scaler = skl.preprocessing.StandardScaler()

# Fit the scaler
X_scaler.fit(X_train)

# Scale the data
X_train_scaled = X_scaler.transform(X_train)
X_test_scaled = X_scaler.transform(X_test)
```

IMPORTANT

> Don't worry if you do not understand what these normalization functions do—we'll cover preprocessing in greater depth later in this module.

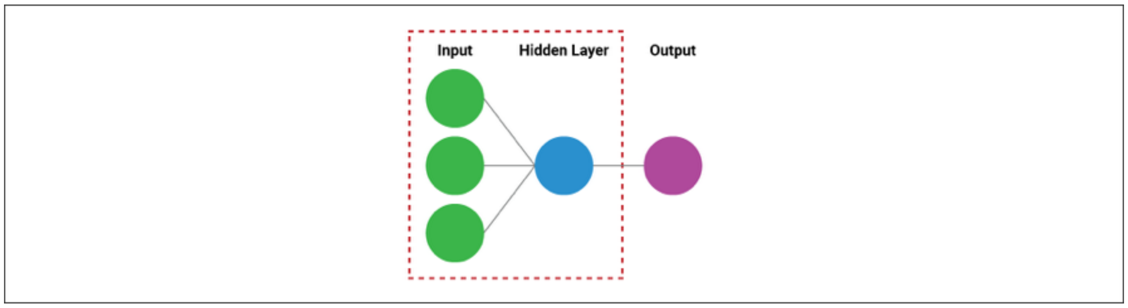
Finally, we have our data ready for our first neural network model! To create the neural network in our notebook, first we must create our Sequential model. To do this, we must add the following code to the notebook:

```
# Create the Keras Sequential model
nn_model = tf.keras.models.Sequential()
```

The `nn_model` object will store the entire architecture of our neural network model. Our next step is to add our first layer, which will contain our inputs and a hidden layer of neurons.

IMPORTANT

> The Keras module does not have specific classes for input, hidden, and output layers. All layers are built using the Dense class, and the input and first hidden layer are always built in the same instance.



As we learned earlier, we can add layers to our Sequential model using Keras' `Dense` class. For our first layer, we need to define a few parameters:

- The `input_dim` parameter indicates how many inputs will be in the model (in this case two).
- The `units` parameter indicates how many neurons we want in the hidden layer (in this case one).
- The `activation` parameter indicates which activation function to use. We'll use the ReLU activation function to allow our hidden layer to identify and train on nonlinear relationships in the dataset.

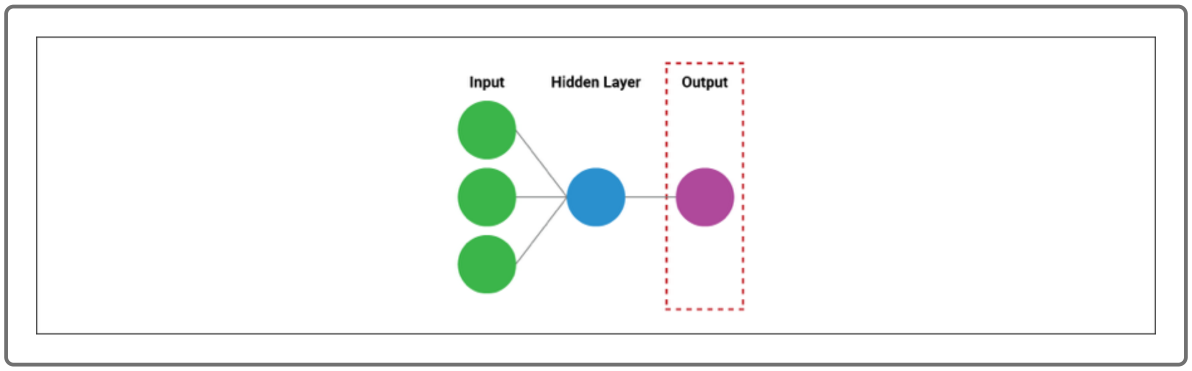
Putting it all together, our first Dense layer should have the following Python code:

```
d our first Dense layer, including the input layer
odel.add(tf.keras.layers.Dense(units=1, activation="relu", input_dim=2)
```

NOTE

> Defining an activation function as part of the first layer is suggested but not required. By default, a Dense layer will look for linear relationships.

Now that we have our input and hidden layers built, we need to add an output layer:

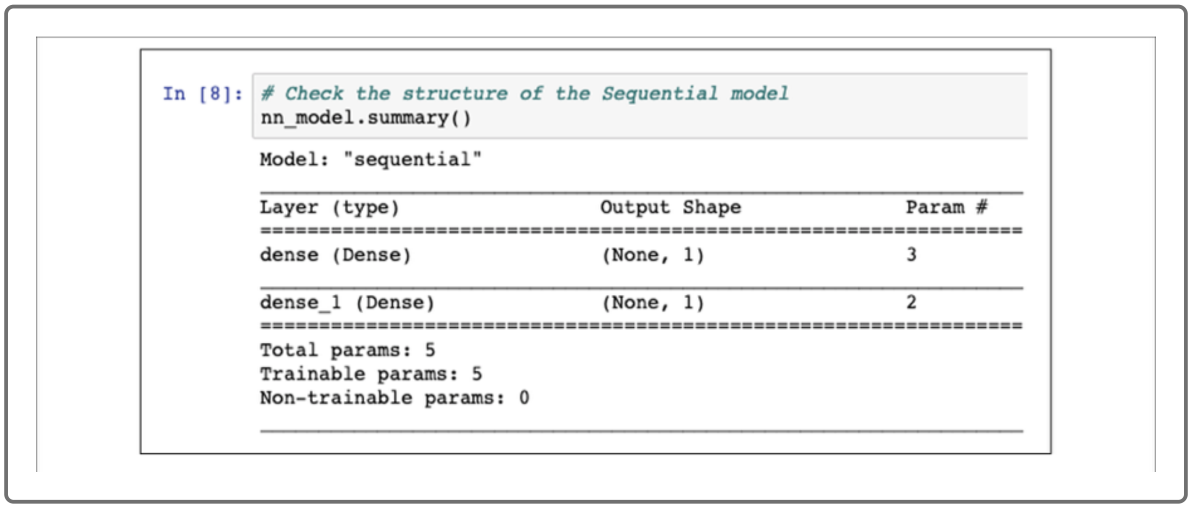


Once again, we'll use the `Dense` class to tell our Sequential model what to do with the data. This time, we only need to supply the number of output neurons. For a classification model, we only want a yes or no binary decision; therefore, we only need one output neuron. In our previous layer, we used a ReLU activation function to enable nonlinear relationships; however, for our classification output, we want to use a sigmoid activation function to produce a probability output. Let's add the following code to our notebooks:

```
# Add the output layer that uses a probability activation function
nn_model.add(tf.keras.layers.Dense(units=1, activation="sigmoid"))
```

Now that we have added our layers to the Sequential model, we can double-check our model structure using the `summary` method. Try running the following code in your notebook:

```
# Check the structure of the Sequential model
nn_model.summary()
```



```
In [8]: # Check the structure of the Sequential model
        nn_model.summary()

        Model: "sequential"
        _________________________________________________________________
        Layer (type)                 Output Shape              Param #
        =================================================================
        dense (Dense)                (None, 1)                 3
        _________________________________________________________________
        dense_1 (Dense)              (None, 1)                 2
        =================================================================
        Total params: 5
        Trainable params: 5
        Non-trainable params: 0
        _________________________________________________________________
```

Error An error occured while saving the activity. Please checkout your network to ensure you are

NOTE

> Note that the number of parameters in each layer does not equal the number of neurons we defined in the notebook. Remember, every layer has one additional input known as our bias term (or weighted constant).

Now that we have our layers defined, we have to inform the model how it should train using the input data. The process of informing the model how it should learn and train is called **compiling** the model.

Depending on the function of the neural network, we'll have to compile the neural network using a specific optimization function and loss metric. The **optimization function** shapes and molds a neural network model while it is being trained to ensure that it performs to the best of its ability. The **loss metric** is used by machine learning algorithms to score the performance of the model through each iteration and epoch by evaluating the inaccuracy of a single input. To enhance the performance of our classification neural network, we'll use the `adam` optimizer, which uses a gradient descent approach to ensure that the algorithm will not get stuck on weaker classifying variables and features. As for the loss function, we'll use `binary_crossentropy`, which is specifically designed to evaluate a binary classification model.

IMPORTANT

> There are many types of **optimization functions (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers)** and **loss metrics (https://www.tensorflow.org/api_docs/python/tf/keras/losses)** you may use in neural networks. We'll discuss and use a few in this module, but feel free to check out the **Keras documentation (https://www.tensorflow.org/guide/keras)** for the full list of options.

In addition to the optimization function and loss metric, we'll also add a more reader-friendly **evaluation metric**, which measures the quality of the machine learning model. There are two main types of evaluation metrics—the model predictive accuracy and model mean squared error (MSE). We use `accuracy` for classification models and `mse` for regression models. For model predictive accuracy, the higher the number the better, whereas for regression models, MSE should reduce to zero.

! Error An error occured while saving the activity. Please checkout your network to ensure you are

Putting all of these metrics together, we'll add and run the following code to our notebooks:

```
# Compile the Sequential model together and customize metrics
nn_model.compile(loss="binary_crossentropy", optimizer="adam", metrics
```