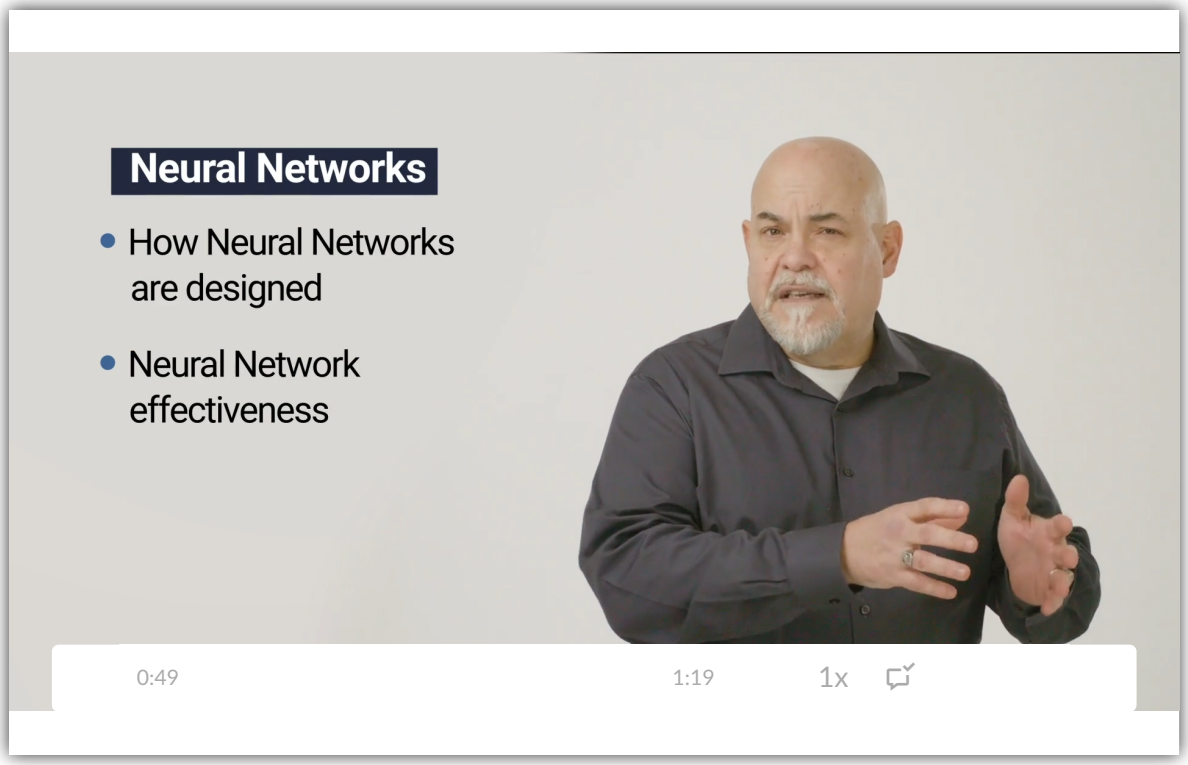
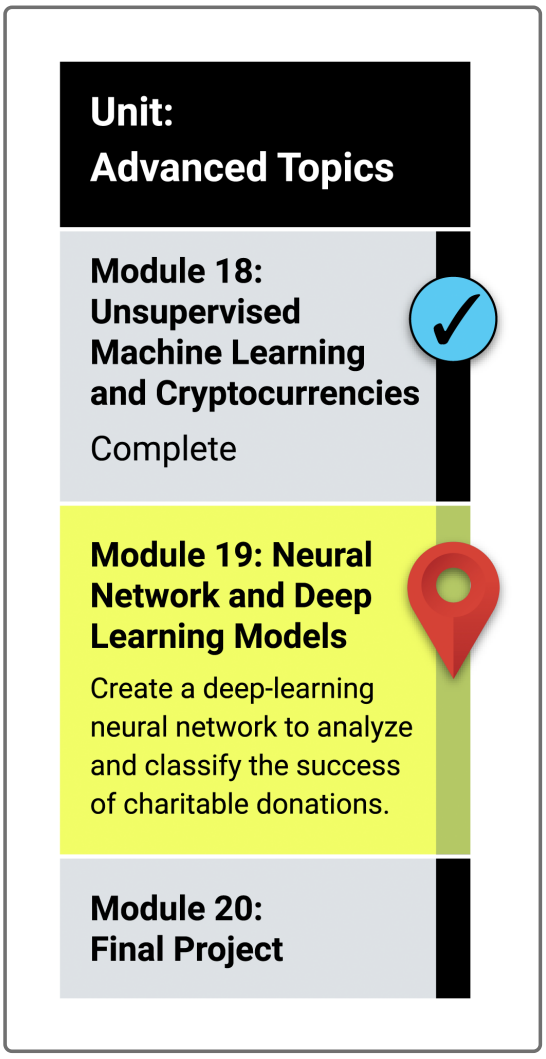
19.0.1 The Rise of Machine Learning



19.0.2 Module 19 Roadmap

The Week Ahead

In this module, we'll explore and implement neural networks using the TensorFlow platform in Python. We'll discuss the background and history of computational neurons as well as current implementations of neural networks as they apply to deep learning. We'll discuss the major costs and benefits of different neural networks and compare these costs to traditional machine learning classification and regression models. Additionally, we'll practice implementing neural networks and deep neural networks across a number of different datasets, including image, natural language, and numerical datasets. Finally, we'll learn how to store and retrieve trained models for more robust uses.



What You Will Learn

By the end of this module, you will be able to:

- Compare the differences between the traditional machine learning classification and regression models and the neural network models.
- Describe the perceptron model and its components.
- Implement neural network models using TensorFlow.
- Explain how different neural network structures change algorithm performance.
- Preprocess and construct datasets for neural network models.
- Compare the differences between neural network models and deep neural networks.
- Implement deep neural network models using TensorFlow.
- Save trained TensorFlow models for later use.

Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Introduction to Module 19 (15 minutes)
- Introduction to Neural Networks (30 minutes)
- Build Your First Neural Network (2 hours)
- Prepare Your Neural Network Datasets (2 hours)
- Dig Deeper Into Neural Networks (2 hours)
- Select the Best Model for Your Dataset (2 hours)
- Export and Import Trained Models (1 hour)
- Application (5 hours)

19.0.3 Nonprofit Networking

19.1.1 What Is a Neural Network?

If there is one thing Beks knows how to do, it is to jump headfirst into new, complicated technical material. After all, she got her start in the data analytic world by taking a 24-week bootcamp. It was no joke, but it taught her that with hard work she can accomplish anything. So, when Andy trusts her to develop a model that will accurately predict sound investments for the foundation, she jumps at the chance to explore neural networks as a potential solution.

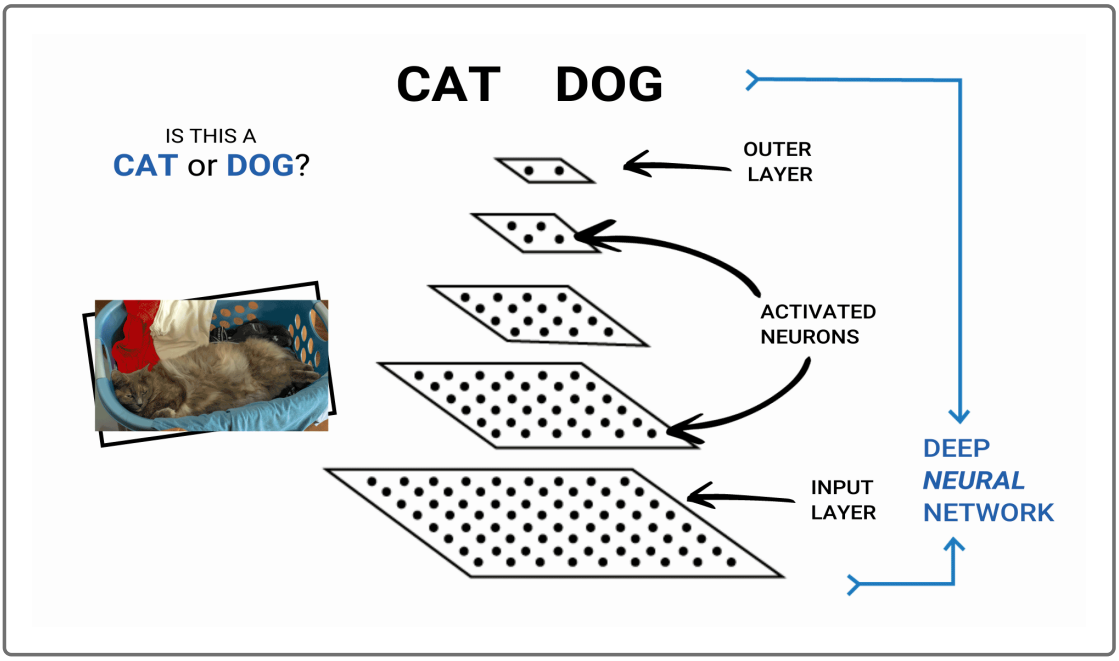
Neural networks (also known as **artificial neural networks**, or **ANN**) are a set of algorithms that are modeled after the human brain. They are an advanced form of machine learning that recognizes patterns and features in input data and provides a clear quantitative output. In its simplest form, a neural network contains layers of **neurons**, which perform individual computations. These computations are connected and weighed against one another until the neurons reach the final layer, which returns a numerical result, or an encoded categorical result.

What is the difference between a numerical and a categorical result?

- A numerical result is a measurement, whereas a categorical result is a label.
- A categorical result is a measurement, whereas a numerical result is a label.

Check Answer

Finish ▸



Neural networks are particularly useful in data science because they serve multiple purposes.

One way to use a neural network model is to create a classification algorithm that determines if an input belongs in one category versus another. Alternatively neural network models can behave like a regression model, where a dependant output variable can be predicted from independent input variables. Therefore, neural network models can be an alternative to many of the models we have learned throughout the course, such as random forest, logistic regression, or multiple linear regression.

There are a number of advantages to using a neural network instead of a traditional statistical or machine learning model. For instance, neural

networks are effective at detecting complex, nonlinear relationships. Additionally, neural networks have greater tolerance for messy data and can learn to ignore noisy characteristics in data. The two biggest disadvantages to using a neural network model are that the layers of neurons are often too complex to dissect and understand (creating a black box problem), and neural networks are prone to overfitting (characterizing the training data so well that it does not generalize to test data effectively). However, both of the disadvantages can be mitigated and accounted for.

REWIND

Overfitting occurs when a model gives undue importance to patterns within a particular dataset that are not found in other, similar datasets.

Neural networks have many practical uses across multiple industries. In the finance industry, neural networks are used to detect fraud as well as trends in the stock market. Retailers like Amazon and Apple are using neural networks to classify their consumers to provide targeted marketing as well as behavior training for robotic control systems. Due to the ease of implementation, neural networks also can be used by small businesses and for personal use to make more cost-effective decisions on investing and purchasing business materials. Neural networks are scalable and effective—it is no wonder why they are so popular.

19.1.2 Perceptron, the Computational Neuron

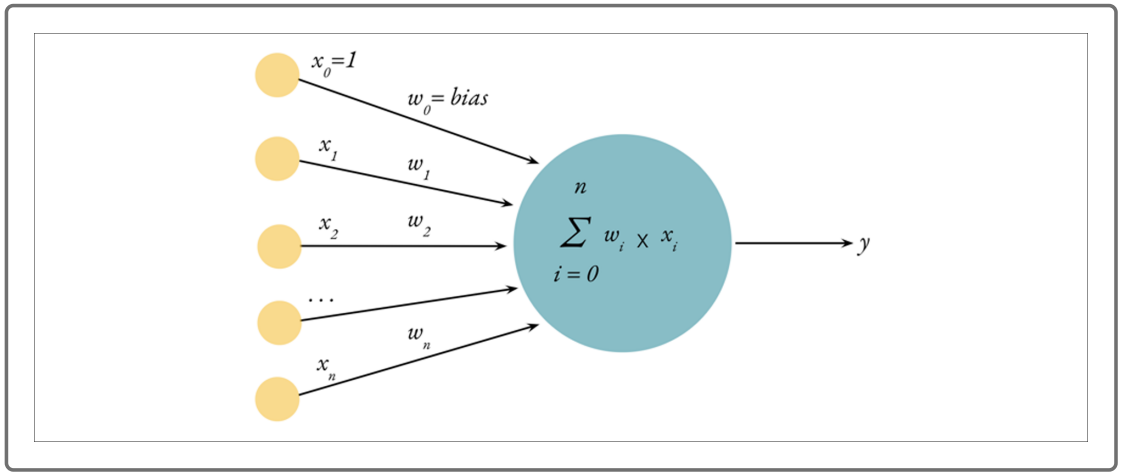
Brain metaphors, easy to use, and cost-effective? Excellent at detecting complex, nonlinear relationships? Neural networks are starting to sound like a great fit for the model. Beks sends Andy a quick Slack message to let him know the research phase of the project is well under way. Her next step will be to dig into the math a bit: What exactly goes into a neural network? To explore this, she'll start with the perceptron model.

Although artificial neural networks have become popular in recent years, the original design for computational neurons (and, subsequently, the neural network) dates as far back as the late 1950s, when Frank Rosenblatt, a pioneer in the field of artificial intelligence, created the perceptron, a machine for training the first neural network. The **perceptron model** is a single neural network unit, and it mimics a biological neuron by receiving input data, weighing the information, and producing a clear output.

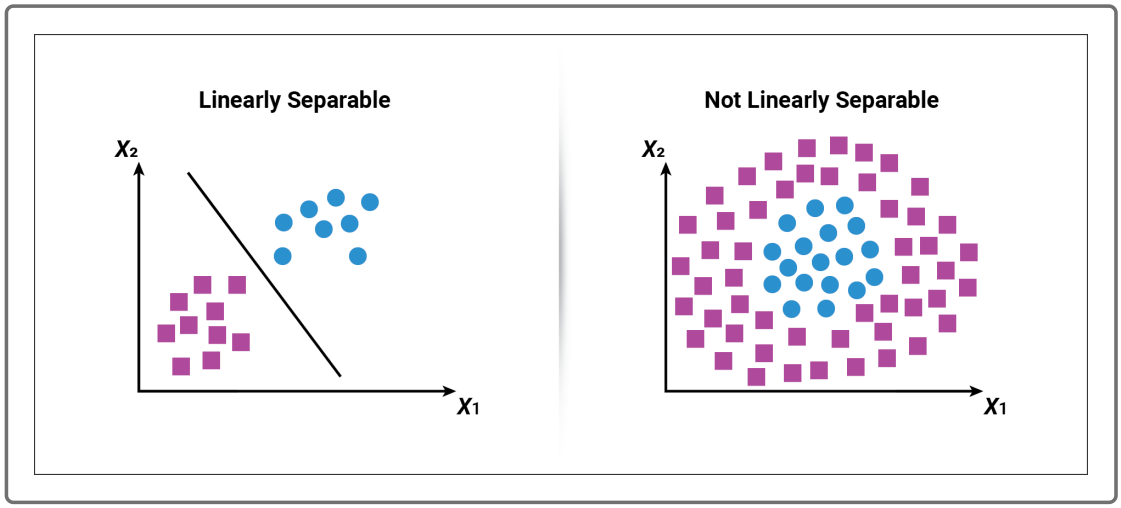
The perceptron model has four major components:

- **Input values**, typically labelled as **x** or χ (chi)

- A **weight coefficient** for each input value, typically labelled as **w** or ω (omega)
- **Bias** is a constant value added to the input to influence the final decision, typically labelled as $\mathbf{w_0}$. In other words, no matter how many inputs we have, there will always be an additional value to "stir the pot."
- A **net summary function** that aggregates all weighted inputs, in this case a weighted summation:



Perceptrons are capable of classifying datasets with many dimensions; however, the perceptron model is most commonly used to separate data into two groups (also known as a **linear binary classifier**). In other words, the perceptron algorithm works to classify two groups that can be separated using a linear equation (also known as **linearly separable**). For example, in the image below, we have a purple group and blue group in both figures. In the left figure, we can draw a line down the middle of the figure to separate the groups entirely. While in the right figure, there is no single (straight) line that can be drawn to separate the two groups entirely. Therefore, the left image is considered linearly separable while the right is not:



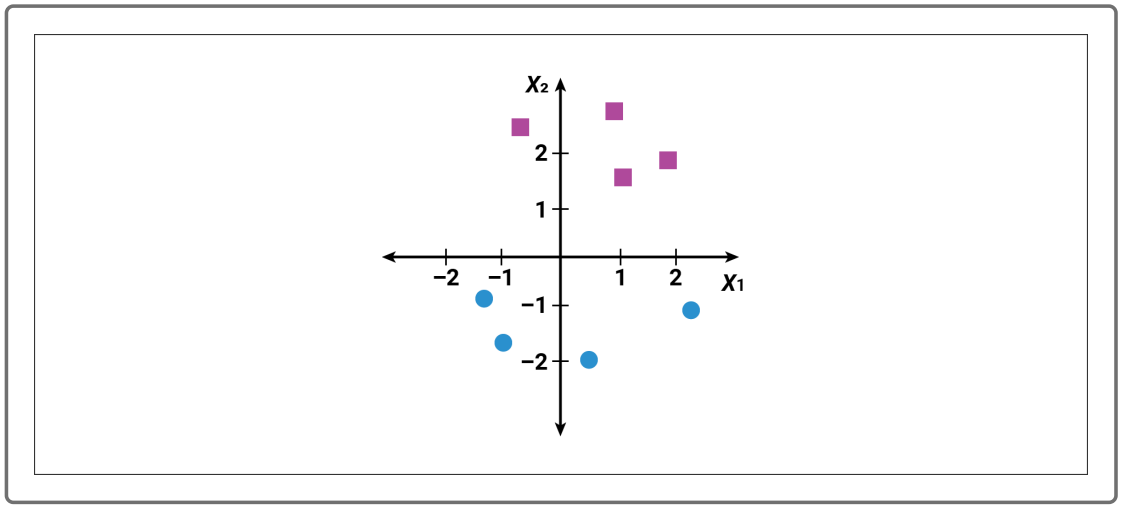
Refer to the perceptron model with labeled input data, above. Does it represent supervised or unsupervised machine learning?

- Supervised machine learning
- Unsupervised machine learning

Check Answer

Finish ▸

The perceptron model is designed to produce a discrete classification model and to learn from the input data to improve classifications as more data is analyzed. To better understand how the perceptron model and algorithm works, let's consider the following dataset:



In this example, we want to generate a perceptron classification model that can distinguish between values that are purple squares versus values that are blue circles. Since this perceptron model will try to classify values in a two-dimensional space, our input values would be:

- χ_2 - the y value
- χ_1 - the x value
- ω_0 - the constant variable (which becomes the bias constant)

IMPORTANT

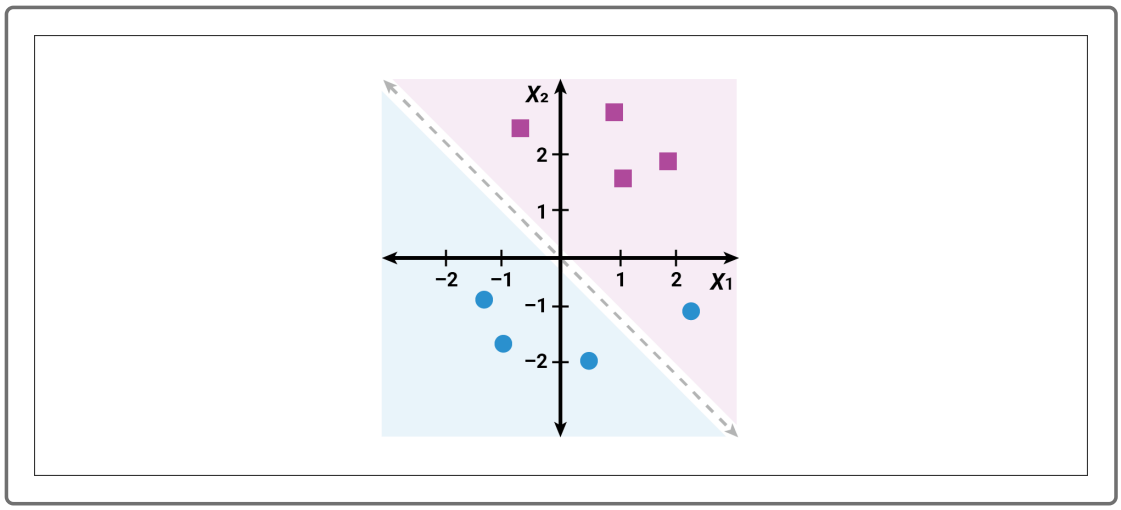
> There will always be one more input variable than the number of dimensions to ensure there is a bias constant within the model.

As for the weight and bias coefficients, these values are arbitrary when the perceptron model first looks at the data. As a result, the two-dimensional perceptron's net sum function would be:

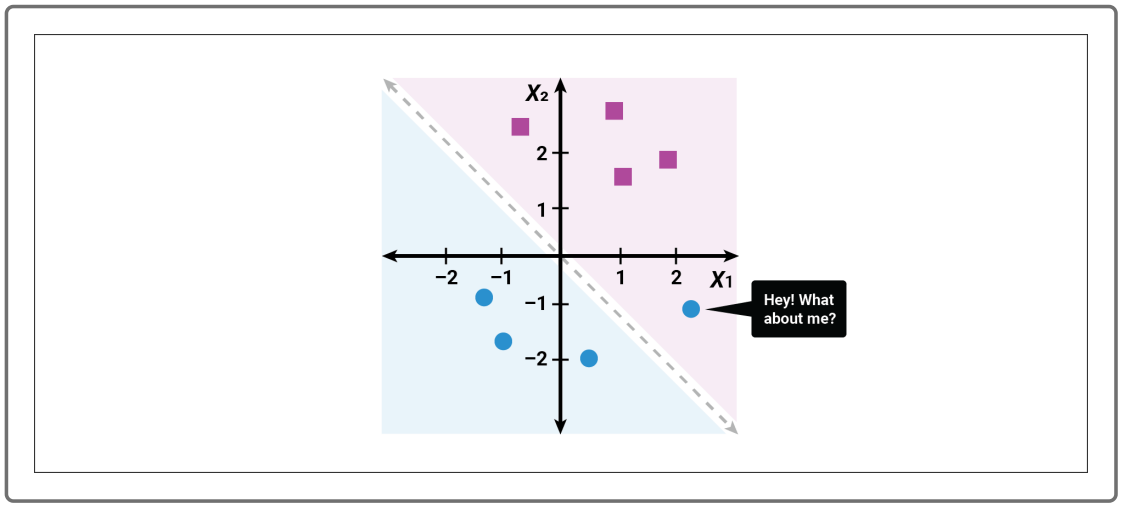
$$\omega_0 + \chi_1\omega_1 + \chi_2\omega_2$$

Where $\chi_0\omega_0$ is the bias term, and $\chi_1\omega_1$ and $\chi_2\omega_2$ are the weighted x and y values for each data point. If the net sum of the data point is greater than zero, it classifies the data point as a purple square, otherwise the data point is classified as a blue circle.

Due to the initial weight coefficients being arbitrary, it is very likely that the first iteration of the perceptron model will classify values incorrectly. Let's say that the first iteration of our perceptron model looks like the following image:

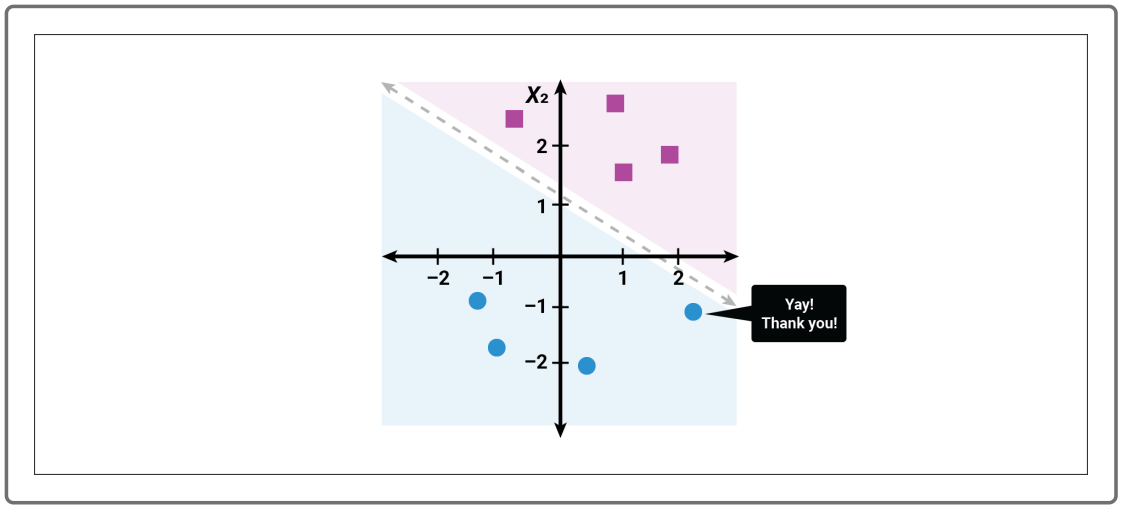


In this image, the perceptron's linear classifier is represented with the dashed line. According to the perceptron model, values above the dashed line would be considered to be purple squares, and values below the dashed line would be considered to be blue circles. Although the perceptron model correctly classified all of the purple square data points, it misclassified one of the blue circle data points:



The next step in the perceptron algorithm is to check each data point and determine if we need to update the weight coefficients to better classify all data points. When the perceptron model evaluates all of the input data, the correctly classified data points will not change the weight coefficients; however, the incorrectly classified data points will adjust the weight coefficients to move toward the data point.

After adjusting the weights, the perceptron algorithm will reevaluate each data point using the new model:



As with other machine learning algorithms, this process of perceptron **model training** continues again and again until one of three conditions are met:

1. The perceptron model exceeds a predetermined performance threshold, determined by the designer before training. In machine learning this is quantified by minimizing the **loss** metric.
2. The perceptron model training performs a set number of iterations, determined by the designer before training.
3. The perceptron model is stopped or encounters an error during training.

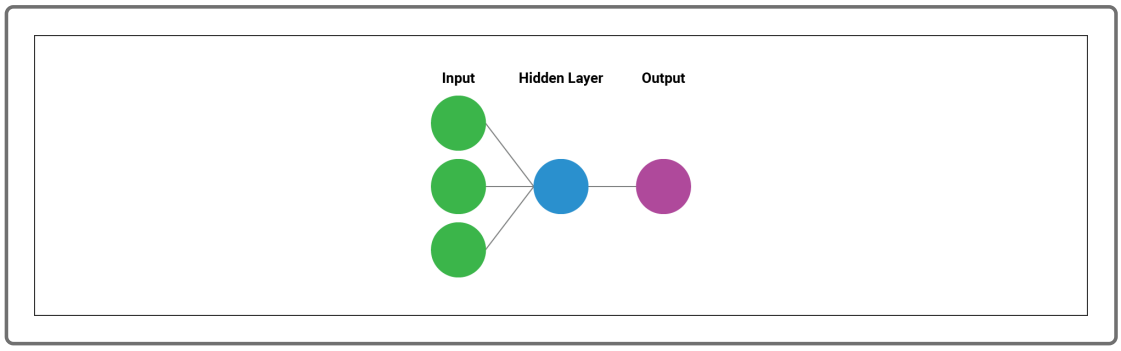
At first glance, the perceptron model is very similar to other classification and regression models; however, the power of the perceptron model

comes from its ability to handle multidimensional data and interactivity with other perceptron models. As more multidimensional perceptrons are meshed together and layered, a new, more powerful classification and regression algorithm emerges—the neural network.

19.1.3 Make the Connections and Explore TensorFlow Playground

Beks has a solid understanding of the structure of a single neuron—now it's time to unpack the structure of the network.

Now that we understand the structure of a single neuron, we can begin to build the structure of a neural network:



A basic neural network has three layers:

- An **input layer** of input values transformed by weight coefficients
- A **single "hidden" layer** of neurons (single neuron or multiple neurons)

- An **output layer** that reports the classification or regression model value

As mentioned previously, neural networks work by linking together neurons and producing a clear quantitative output. But if each neuron has its own output, how does the neural network combine each output into a single classifier or regression model? The answer is an **activation function**.

The **activation function** is a mathematical function applied to the end of each “neuron” (or each individual perceptron model) that transforms the output to a quantitative value. This quantitative output is used as an input value for other layers in the neural network model. There are a wide variety of activation functions that can be used for many specific purposes; however, most neural networks will use one of the following activation functions:

1. The **linear function** transforms the output into the coefficients of a linear model (the equation of a line).
2. The **sigmoid function** is identified by a characteristic S curve. It transforms the output to a range between 0 and 1.
3. The **tanh function** is also identified by a characteristic S curve; however, it transforms the output to a range between -1 and 1.
4. The **Rectified Linear Unit (ReLU) function** returns a value from 0 to infinity, so any negative input through the activation function is 0. It is the most used activation function in neural networks due to its simplifying output, but it might not be appropriate for simpler models.
5. The **Leaky ReLU function** is a “leaky” alternative to the ReLU function, whereby negative input values will return very small negative values.

Match the activation function names to the correct image using the descriptions in this section for guidance.

Image A

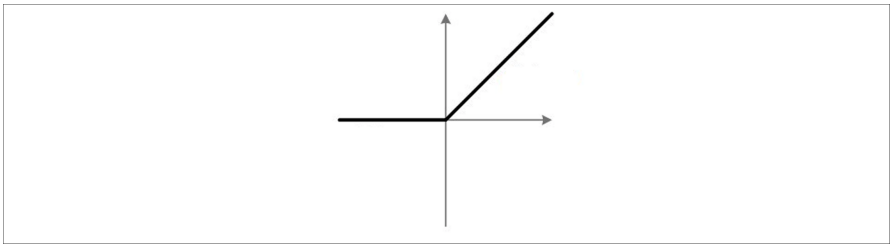


Image B

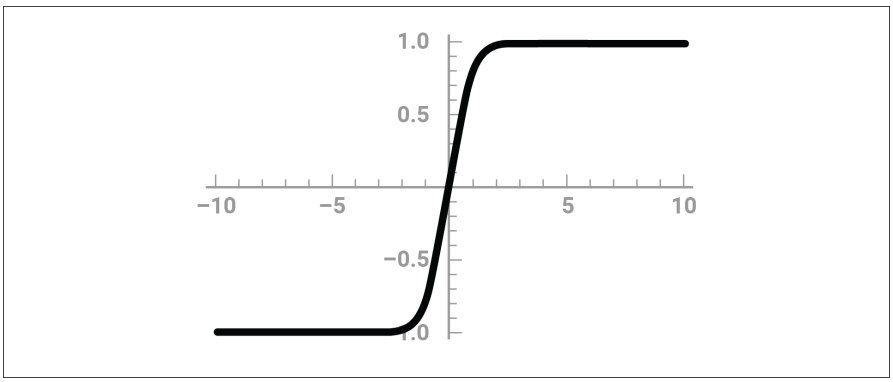


Image C

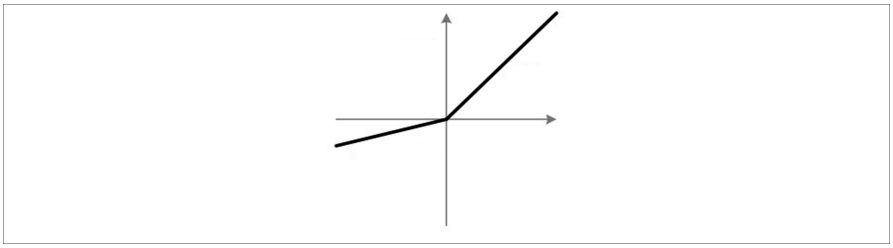


Image D

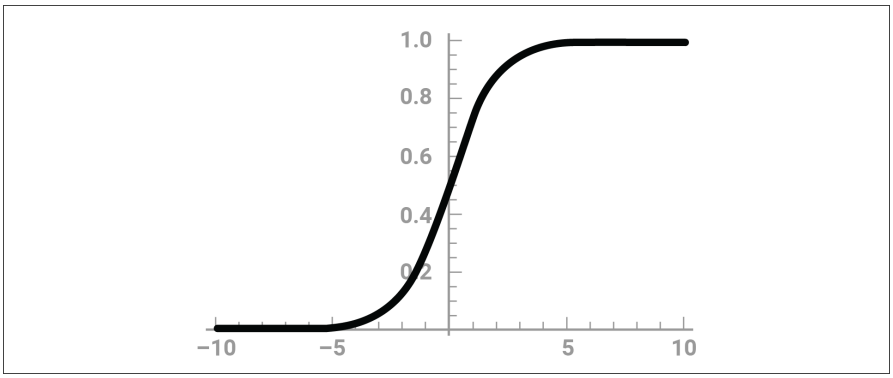
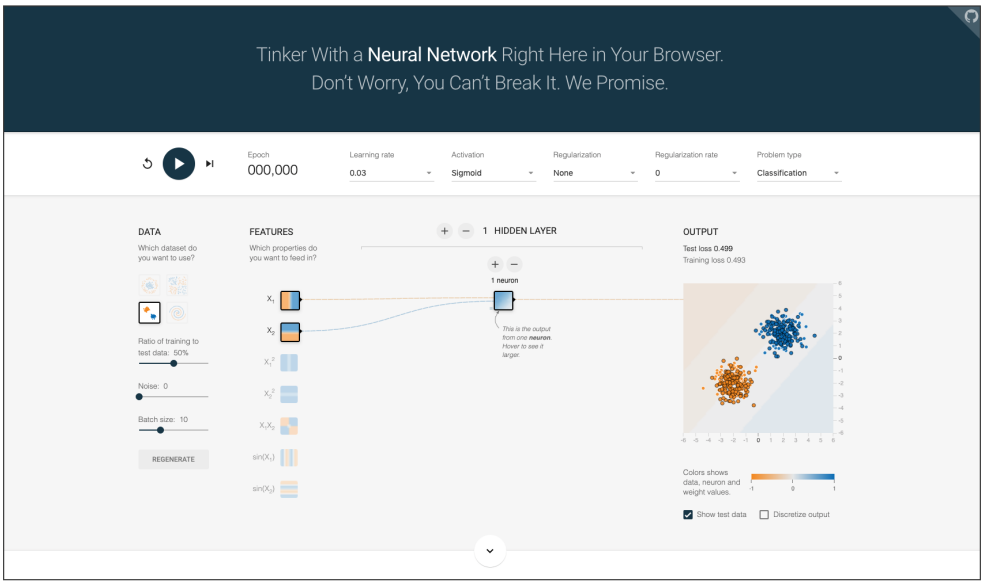


Image A:

To better understand how multiple neurons connect together with activation functions to make a robust neural network, we'll explore a teaching application known as the **TensorFlow Playground (https://playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=1&seed=0.10587&showTestData=false&discretize=true&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false&discretize_hide=true&regularization_hide=true&learningRate_hide=true&regularizationRate_hide=true&percTrainData_hide=true&showTestData_hide=true&noise_hide=true&batchSize_hide=true)** .

TensorFlow is a neural network and machine learning library for Python that has become an industry standard for developing robust neural network models. TensorFlow developed its playground application as a teaching tool to demystify the black box of neural networks and provide a working simulation of a neural network as it trains on a variety of different datasets and conditions.

Does your screen match the following?

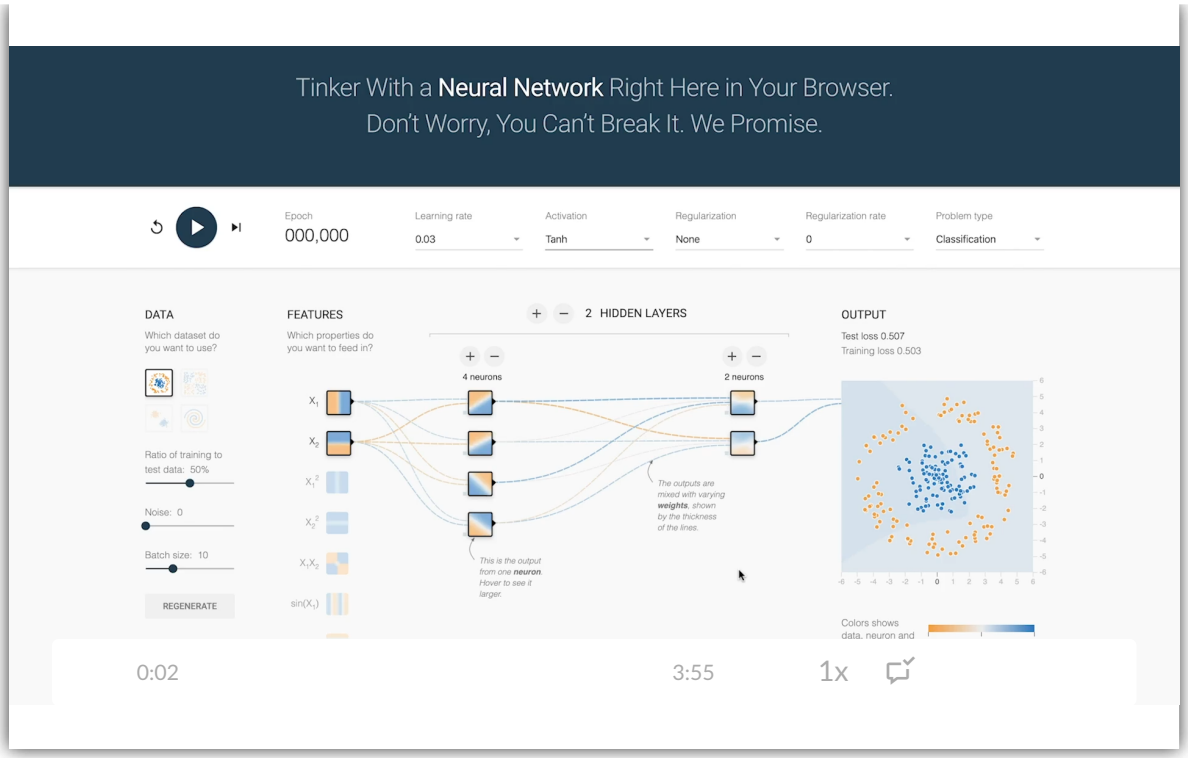


- Yes.
- No.

Check Answer

Finish ▸

In this video, you'll use the simulations in the TensorFlow Playground to better understand how altering the neurons and activation functions of a neural network can change its performance.



Now that we have spent some time understanding the structure of a basic neural network and how each component impacts the final model, it is time to learn how to build our own functioning models. Don't worry if you feel like there are too many components, options, and parameters to keep track of—neural networks start off simple and grow to match the complexity of the input data.

19.2.1 Set Up the TensorFlow Pipeline

Beks knows from her boot camp experience that there's nothing quite like hands-on experience when it comes to coding. So, now that she feels comfortable playing with TensorFlow, she's ready to start the REAL fun: building her first neural network!

To build our first neural networks in Python, we must first get acquainted with the TensorFlow 2.0 programming library. TensorFlow 2.0 provides users an easy-to-use application programming interface (API) that can design, train, export, and import neural network models without extensive machine learning knowledge. To install TensorFlow 2.0 into our Anaconda environment, we can use pip. Simply type the following command in your terminal (for Mac OS X users) or your Git Bash window (for Windows users):

```
# Installs latest version of TensorFlow 2.X
conda install tensorflow
```

Run this code to check if your TensorFlow version is installed correctly. Did you get an error?

```
python -c "import tensorflow; print(tensorflow.__version__)"
```

- ○ Yes.
- ○ No.

Check Answer

Finish ▸

There are a number of smaller modules within the TensorFlow library that make it even easier to build machine learning models. For our purposes, we'll use the Keras module to help build our basic neural networks. Keras contains multiple classes and objects that can be combined to design a variety of neural network types. These classes and objects are order-dependant, which means that depending on what Keras objects are used (and in what order), the behavior of the neural network model will change accordingly. For our basic neural network, we'll use two Keras classes:

- The **Sequential** class is a linear stack of neural network layers, where data flows from one layer to the next. This model is what we simulated in the TensorFlow Playground.
- The generalized **Dense** class allows us to add layers within the neural network.

With the Sequential model, we'll add multiple Dense layers that can act as our input, hidden, and output layers. For each Dense layer, we'll define the number of neurons, as well as the activation function. Once we have completed our Sequential model design, we can apply the same Scikit-learn **model -> fit -> predict/transform** workflow as we used for other machine learning algorithms.

REWIND

The process of **model -> fit -> predict/transform** follows the same general steps across all of data science:

1. Decide on a model, and create a model instance.
2. Split into training and testing sets, and preprocess the data.
3. Train/fit the training data to the model. (Note that “train” and “fit” are used interchangeably in Python libraries as well as the data field.)
4. Use the model for predictions and transformations.

19.2.2 Build a Basic Neural Network

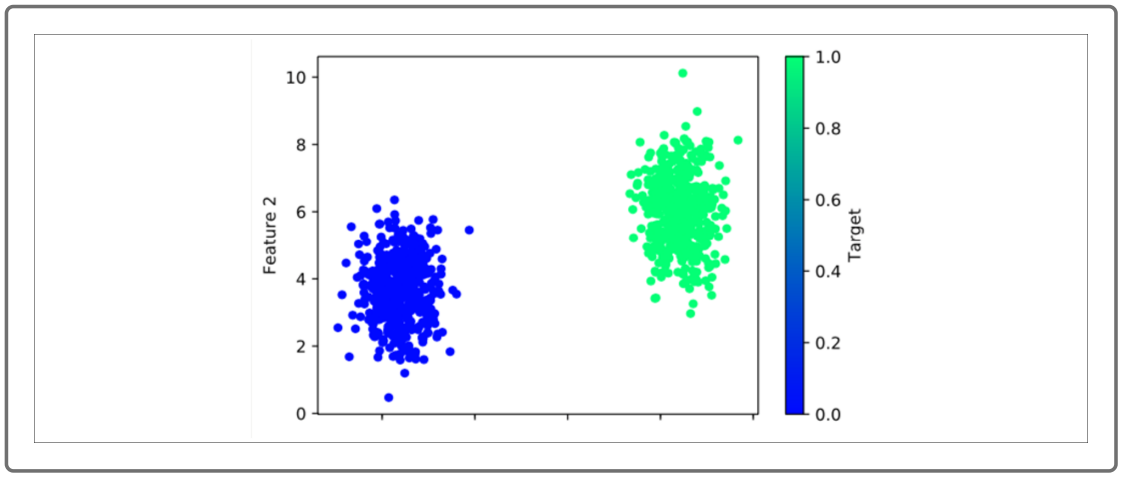
Beks knows she'll want to work with the foundation's datasets once she starts working on her analysis, but while she's still exploring concepts, she wants some dummy data that she can use to train and test.

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:

```
# Generate dummy dataset
X, y = make_blobs(n_samples=1000, centers=2, n_features=2, random_state=78)

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

# Plotting the dummy data
df.plot.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:

```
# Use sklearn to split dataset
from sklearn.model_selection import train_test_split
X_train, 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.

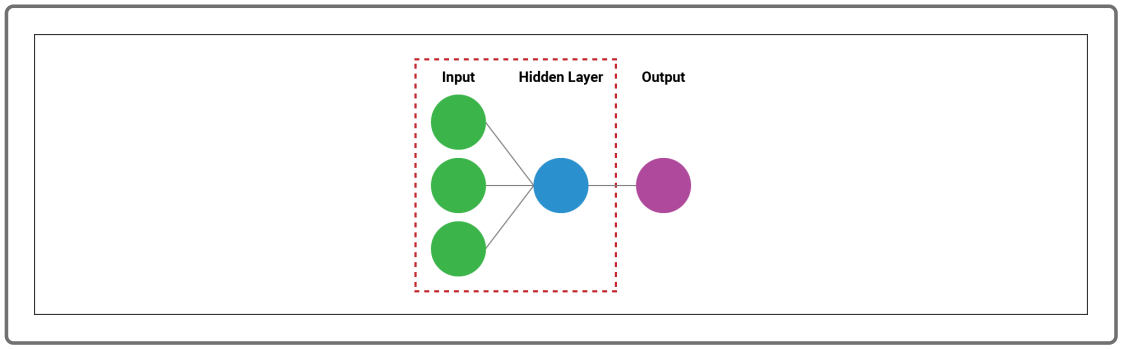
Finally, we have our data ready for our first neural network model! To create the neural network in our notebook, first we must create our Sequential model. To do this, we must add the following code to the notebook:

```
# 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.

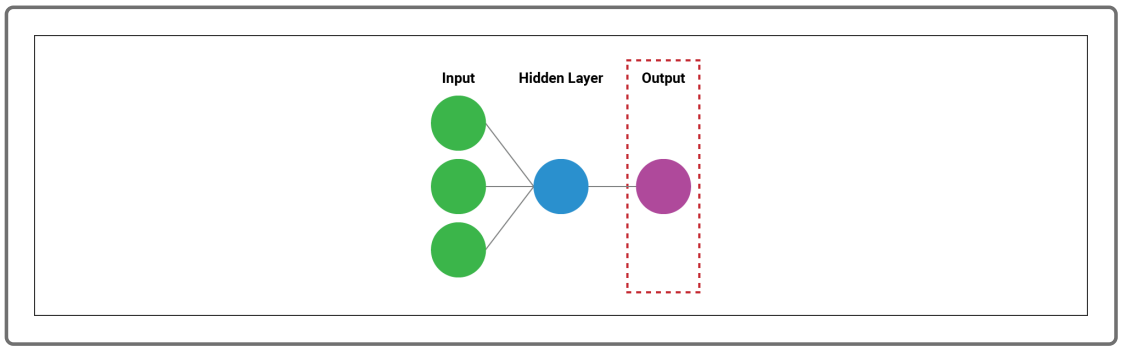
Putting it all together, our first Dense layer should have the following Python code:

```
# Add our first Dense layer, including the input layer
nn_model.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:

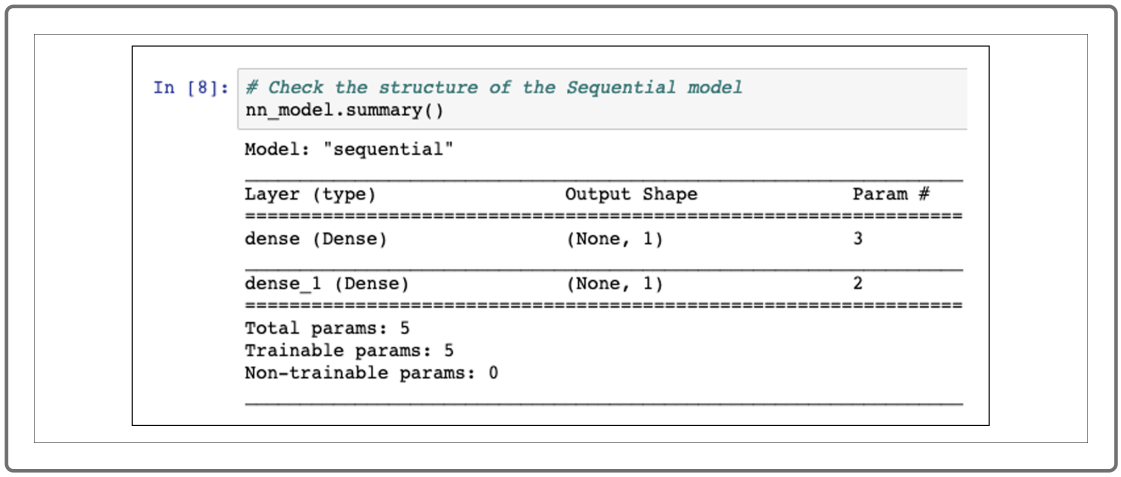


Once again, we'll use the `Dense` class to tell our Sequential model what to do with the data. This time, we only need to supply the number of output neurons. For a classification model, we only want a yes or no binary decision; therefore, we only need one output neuron. In our previous layer, we used 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()
```



After running the previous code block, how many total parameters does the model have?

- Six
- Five
- Four

Check Answer

Finish ▸

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.

Match the key terms with their definitions.

		loss metric	evaluation metric	optimization function	activation function
A	Measures the quality of a machine learning model.	○	○	○	○
B	Measures how poorly a model characterizes the data after each iteration.	○	○	○	○
C	Improves the performance of a machine learning algorithm.	○	○	○	○
D	Adds an additional step at each layer.	○	○	○	○

Check Answer

Finish ▸

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=["acc
```

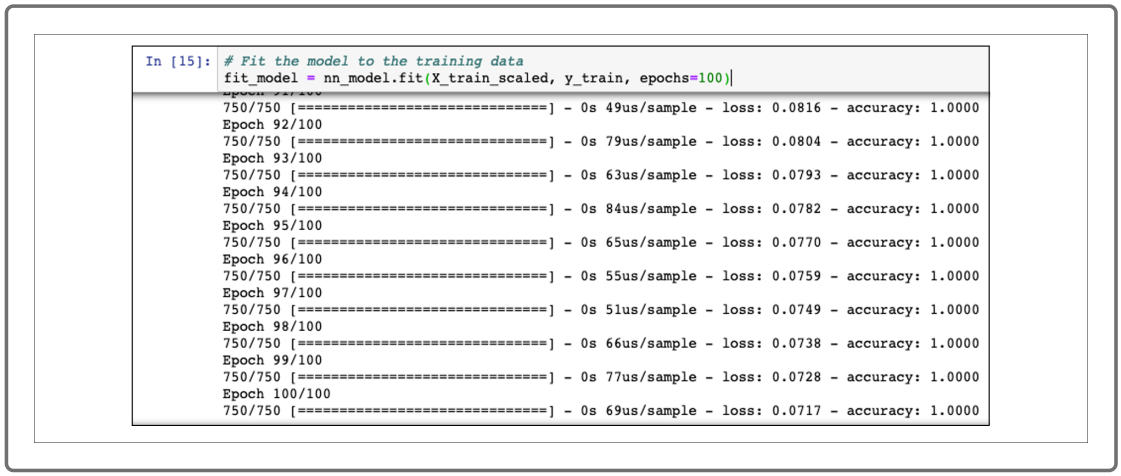
19.2.3 Train and Test a Basic Neural Network

Beks has her first neural network modeled and compiled. It's time to put that dummy data to use.

Beks knows that neural networks can be prone to overfitting, so she wants to be extra careful with her training and testing sets.

At last, our Sequential neural network is modeled and compiled, so now we can move onto training/fitting our model to the dummy data. To train/fit our Keras model, we'll use the `fit` method and provide the x training values and y training values, as well as the number of epochs. As we tested earlier in the TensorFlow Playground, the neural network binary classification model reached optimal performance at around 100 epochs. Since we designed our Sequential model to match the TensorFlow Playground simulation, we should expect similar performance. To our notebooks, we'll add and run the following code:

```
# Fit the model to the training data
fit_model = nn_model.fit(X_train_scaled, y_train, epochs=100)
```

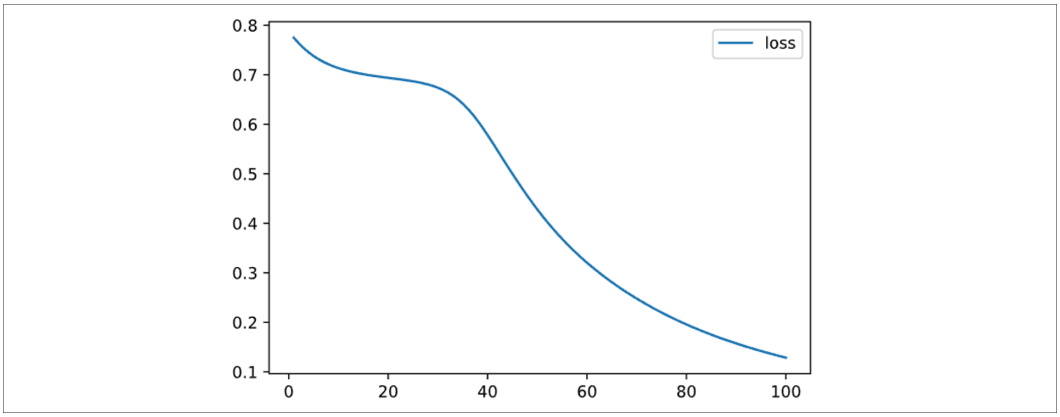


Looking at the model training output, we know that the loss metric was 0.07 and the predictive accuracy is 1.0. This means that although our model performance had more loss than the simulation data, the model correctly classifies all of our training data, which is sufficient for our needs.

Under the hood, the neural network will select random weights to start training the model, so that each and every time we create a neural network, the model will be different. However, due to the `adam` optimizer, our model's end performance should be very similar, regardless of the loss function.

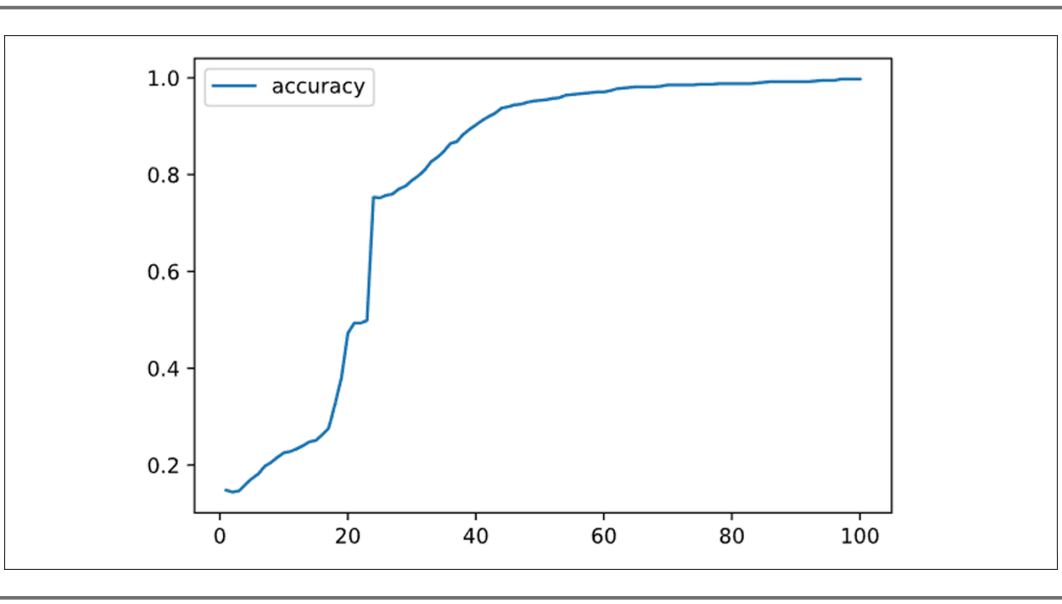
When training completes, the model object stores the loss and accuracy metrics across all epochs, which we can use to visualize the training progress. For example, if we wanted to visualize our model's loss over the full 100 epochs, we can run the following code in our notebooks:

```
# Create a DataFrame containing training history
history_df = pd.DataFrame(fit_model.history, index=range(1,len(fit_model.his

# Plot the loss
history_df.plot(y="loss")
```



Similarly, we can plot the accuracy over time using the same DataFrame:

```
# Plot the accuracy
history_df.plot(y="accuracy")
```



Looking at the loss and accuracy plots in the images above, how many epochs did it take before the model started to fit the training data with high success?

- ○ 0–20 epochs
- ○ 10–20 epochs
- ○ 20–30 epochs

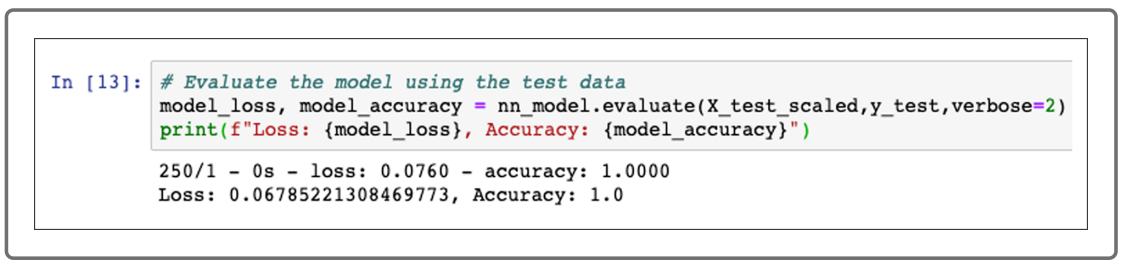
Check Answer

Finish ▸

Remember, our neural network consisted of a single hidden layer with one neuron—if we were to increase the number of neurons in the hidden layer, the neural network would have been able to fit the training data even faster!

Now that our model has been properly trained, we must evaluate model performance using the test data. Testing a neural network model in TensorFlow is very similar to testing a machine learning model in Scikit-learn. For our purposes, we'll use the `evaluate` method and print the testing loss and accuracy values. In our notebooks, we'll add and run the following code:

```
# Evaluate the model using the test data
model_loss, model_accuracy = nn_model.evaluate(X_test_scaled,y_test,verbose=
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```



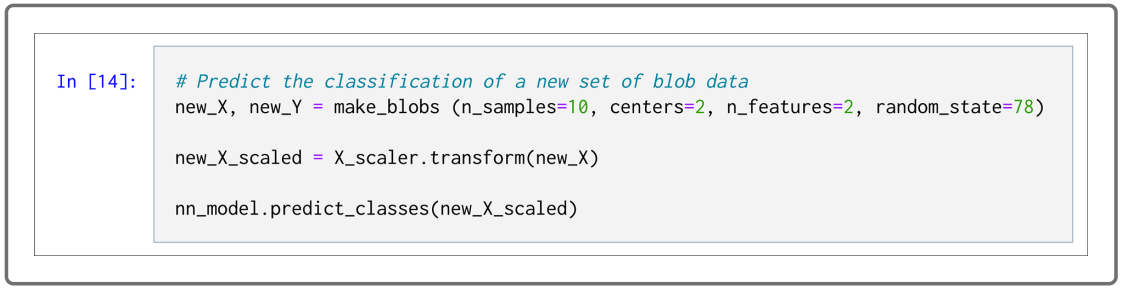
Looking at the performance metrics from the model, the neural network was able to correctly classify each of the points in the test data. In other words, the model was able to correctly classify data it was not trained on 100% of the time. Although perfect model performance is ideal, more complex datasets and models may not be able to achieve 100% accuracy. Therefore, it is important to establish model performance thresholds before designing any machine learning model. Depending on the type of data and the use case, we may have to recreate and retrain a model using different parameters, different training/test data, or even look to use a different model entirely.

NOTE

> Later in this module, we'll discuss strategies on improving a model's performance.

Now that we have a trained neural network model and we have verified its performance using a test dataset, we can apply this model to novel datasets and predict the classification of a data point. In our Sequential model, we can use the appropriately named `predict_classes` method to generate predictions on new data. For example, if we wanted to predict the class of a new blob dataset, we can add and run the following code:

```
# Predict the classification of a new set of blob data
new_X, new_Y = make_blobs(n_samples=10, centers=2, n_features=2, random_stat
new_X_scaled = X_scaler.transform(new_X)
nn_model.predict_classes(new_X_scaled)
```



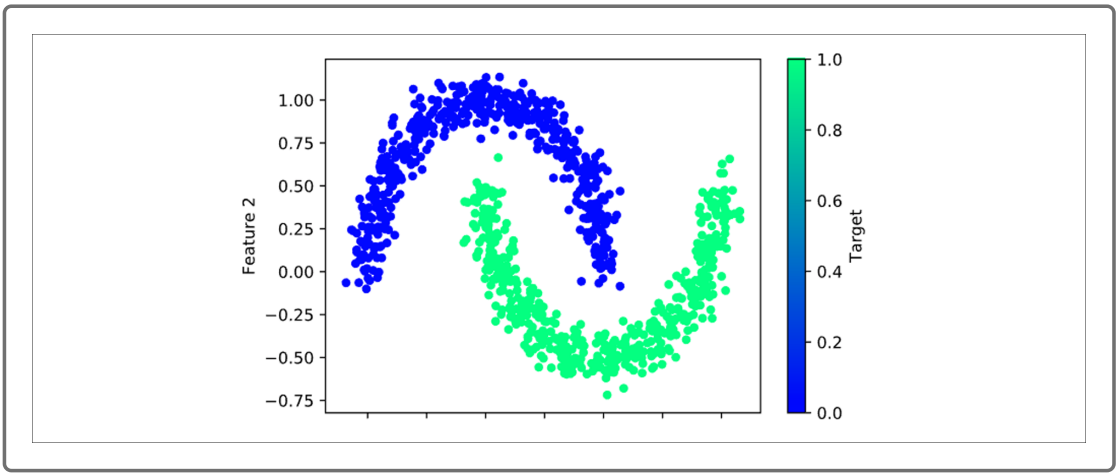
19.2.4 Nuances of Neural Networks on Nonlinear Numbers

Beks is really excited about her first single-neuron, single-layer model. She knows this model will work really well on linear data. But she *also* knows the foundation's work, and knows that some of the relationships that Andy will want to see likely won't be linear. How can neural networks help Beks predict investments when the data is more complicated? Lucky for Beks, she has the work ethic to match *any* dataset. Time to dive back in!

Now that we have designed a basic single-neuron, single-layer model, trained the model, and evaluated its performance using a test dataset, we are ready to start testing the limits of the basic neural network. As previously mentioned, simple linear regression and singular perceptron models work *really* well as a binary classifier when the data is linearly separable. But what about nonlinear data? How does our basic neural network model behave when data becomes more complicated?

To test this behavior, let's generate some new dummy data. This time we'll generate some nonlinear moon-shaped data using Scikit-learn's `make_moons` method and visualize it using Pandas and Matplotlib. Using the same notebook, we'll add and run the following code:

```
from sklearn.datasets import make_moons

# Creating dummy nonlinear data
X_moons, y_moons = make_moons(n_samples=1000, noise=0.08, random_state=78)

# Transforming y_moons to a vertical vector
y_moons = y_moons.reshape(-1, 1)

# Creating a DataFrame to plot the nonlinear dummy data
df_moons = pd.DataFrame(X_moons, columns=["Feature 1", "Feature 2"])
df_moons["Target"] = y_moons

# Plot the nonlinear dummy data
df_moons.plot.scatter(x="Feature 1",y="Feature 2", c="Target",colormap="wint
```



Since we are not changing the structure of our neural network, nor are we changing its function, we can use the same Sequential model object. The only difference from our previous workflow is this time we will retrain and evaluate on the nonlinear moon-shaped data.

NOTE

> If you want to preview how this model behaves, try simulating the single-neuron, single-layer model in **TensorFlow Playground (https://playground.tensorflow.org/#activation=sigmoid&batchSize=10&dataset=spiral&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=1&s**

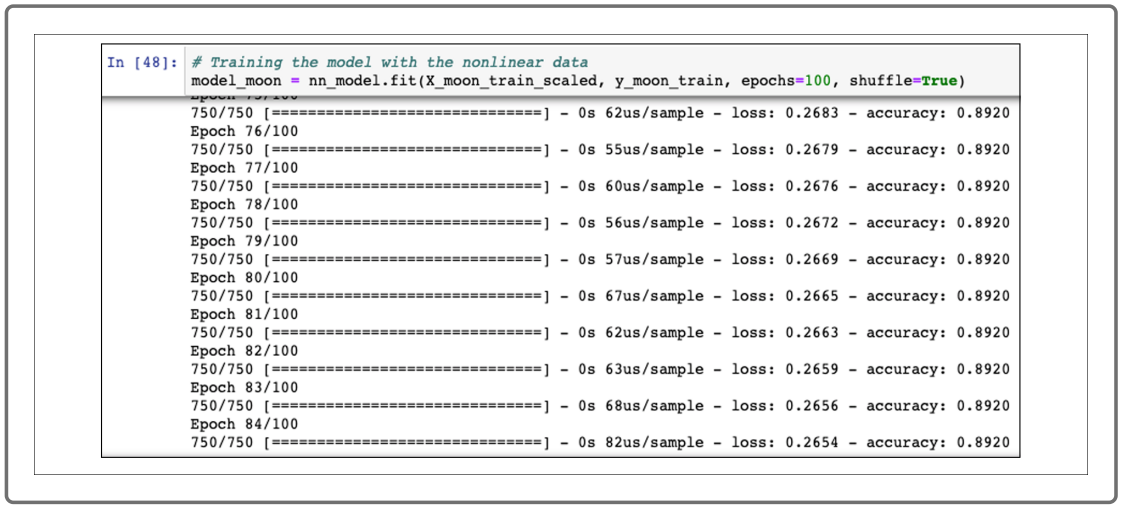
> eed=0.81307&showTestData=false&discretize=true&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false&discretize_hide=true&regularization_hide=true&learningRate_hide=true&regularizationRate_hide=true&percTrainData_hide=true&showTestData_hide=true&noise_hide=true&batchSize_hide=true) , using their spiral dataset!

As with our previous dummy data example, we need to split our nonlinear data into training and testing datasets and normalize our datasets. In the same notebook, we'll add and run the following code:

```
# Create training and testing sets
X_moon_train, X_moon_test, y_moon_train, y_moon_test = train_test_split(
    X_moons, y_moons, random_state=78
)

# Create the scaler instance
X_moon_scaler = skl.preprocessing.StandardScaler()

# Fit the scaler
X_moon_scaler.fit(X_moon_train)

# Scale the data
X_moon_train_scaled = X_moon_scaler.transform(X_moon_train)
X_moon_test_scaled = X_moon_scaler.transform(X_moon_test)
```

Just as we did with the linear data, we'll train our neural network model using the `fit` method on the nonlinear training data. Let's add and run the following code to our notebooks:

```
# Training the model with the nonlinear data
model_moon = nn_model.fit(X_moon_train_scaled, y_moon_train, epochs=100, shu
```



```
In [48]: # Training the model with the nonlinear data
         model_moon = nn_model.fit(X_moon_train_scaled, y_moon_train, epochs=100, shuffle=True)
         750/750 [==============================] - 0s 62us/sample - loss: 0.2683 - accuracy: 0.8920
         Epoch 76/100
         750/750 [==============================] - 0s 55us/sample - loss: 0.2679 - accuracy: 0.8920
         Epoch 77/100
         750/750 [==============================] - 0s 60us/sample - loss: 0.2676 - accuracy: 0.8920
         Epoch 78/100
         750/750 [==============================] - 0s 56us/sample - loss: 0.2672 - accuracy: 0.8920
         Epoch 79/100
         750/750 [==============================] - 0s 57us/sample - loss: 0.2669 - accuracy: 0.8920
         Epoch 80/100
         750/750 [==============================] - 0s 67us/sample - loss: 0.2665 - accuracy: 0.8920
         Epoch 81/100
         750/750 [==============================] - 0s 62us/sample - loss: 0.2663 - accuracy: 0.8920
         Epoch 82/100
         750/750 [==============================] - 0s 63us/sample - loss: 0.2659 - accuracy: 0.8920
         Epoch 83/100
         750/750 [==============================] - 0s 68us/sample - loss: 0.2656 - accuracy: 0.8920
         Epoch 84/100
         750/750 [==============================] - 0s 82us/sample - loss: 0.2654 - accuracy: 0.8920
```

Notice that this time our single-neuron, single-layer neural network was unable to accurately classify all of our training data.

How many epochs did it take for TensorFlow to stop training the model?

- 84 epochs
- 85 epochs
- 86 epochs

Check Answer

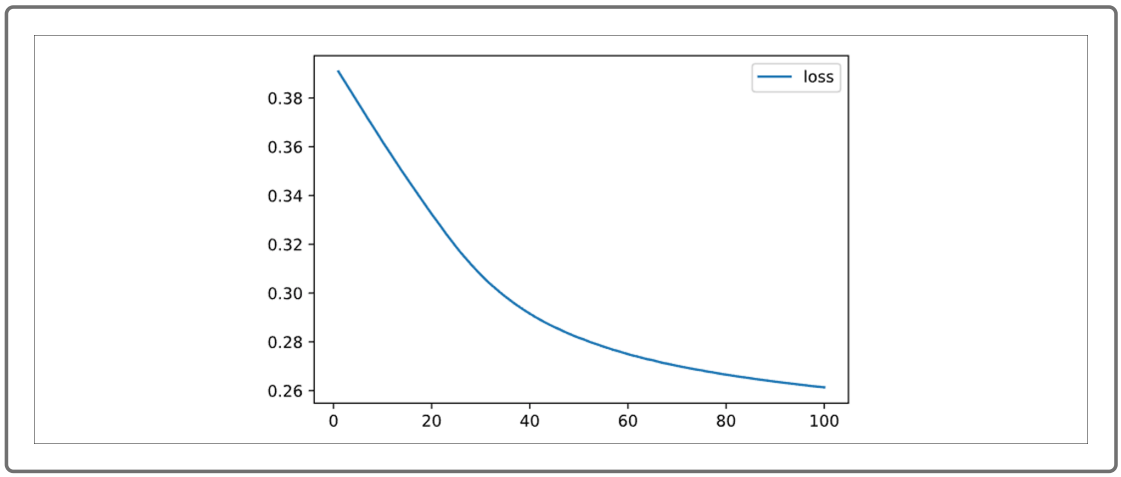
Finish ▸

In our example, TensorFlow stopped training the model after 84 epochs because the loss metric was no longer decreasing at a substantial rate, and the accuracy was not improving.

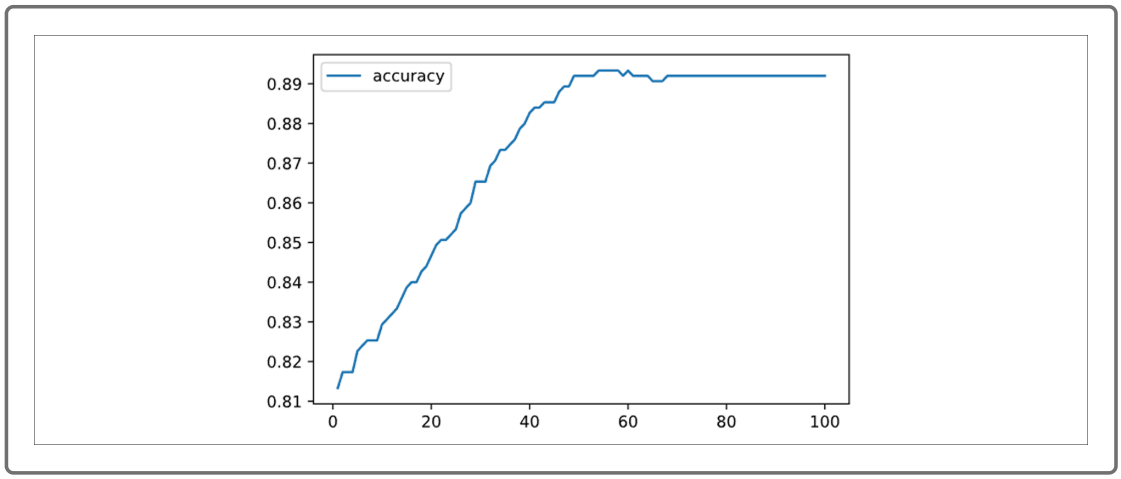
Let's plot out the loss and accuracy curves over our training iterations. Again, we'll add and run the following code to the notebook:

```
# Create a DataFrame containing training history
history_df = pd.DataFrame(model_moon.history, index=range(1,len(model_moon.h
```

```
# Plot the loss
history_df.plot(y="loss")
```




```
# Plot the loss
history_df.plot(y="accuracy")
```



According to the accuracy metric, the basic single-neuron, single-layer neural network model was only able to correctly classify 89% of all data points in the nonlinear training data. Depending on a person's use case, 89% accuracy could be sufficient for a first-pass model. For example, if we were trying to use a neural network model to separate left-handed people from right-handed people, a model that is correct 89% of the time is very accurate, and guessing incorrectly does not have a huge negative impact.

However, in many industrial and medical use cases, a model's classification accuracy must exceed 95% or even 99%. In these cases, we wouldn't be satisfied with the basic single-neuron, single-layer neural network model, and we would have to design a more robust neural network. In summary, the more complicated and nonlinear the dataset, the more components we'd need to add to a neural network to achieve our desired performance.

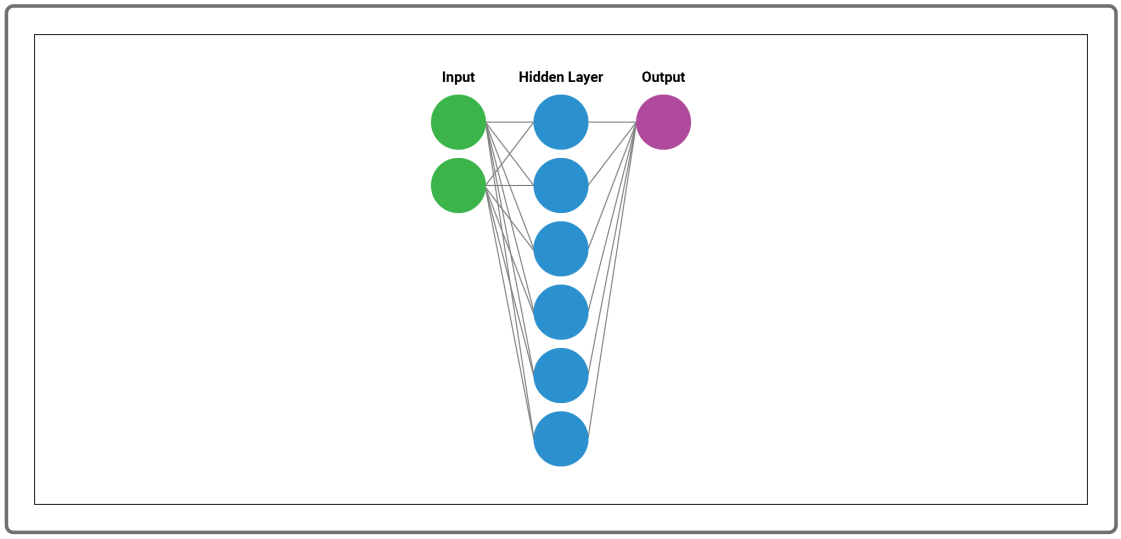
19.2.5 Create the Connective Tissue, the Multiple-Neuron Neural Network

Beks and Andy have a great relationship—Andy trusts Beks to develop the best possible models, and Beks puts in the work to make sure Andy has the information he needs to make decisions. So, even though Andy might not ask questions about a model that is correct 89% of the time, Beks isn't quite satisfied that it will actually serve the foundations—or the grantees—as well as it could. How might she make a model that performs a bit better? She decides to give multiple-neuron neural networks a try.

Earlier in the module, we learned that adding neurons to a neural network can help to generate a well-performing model faster than using a single-neuron, single-layer neural network. In fact, there are multiple advantages to adding neurons to our hidden layer in a neural network. Some of the advantages follow:

- There is a distributed effort to find optimal weights—faster.
- Each neuron can focus on different features to identify nonlinear effects—smarter.
- It is less likely to fixate on complex variables—more robust.

So, if adding more neurons to our neural network model increases the performance, why wouldn't we always use the maximum number of neurons? There are two main reasons to limit the number of neurons in a neural network model: overfitting and computation resources. Similar to other machine learning algorithms, neural networks are susceptible to overfitting where the model fits the training data too well. As a result of overfitting, the neural network will not generalize well and won't be able to classify new data correctly. Additionally, a neural network model with a large number of neurons requires equally large training dataset—training a large neural network requires more data, more epochs, and more time. Therefore, it is important that a neural network model has an appropriate number of neurons to match the size of the data, the complexity of the problem, and the amount of input neurons:



IMPORTANT

> A good rule of thumb for a basic neural network is to have two to three times the amount of neurons in the hidden layer as the number of inputs.

Since our blob and moon-shaped dummy datasets are created using two variables, our neural network model uses two input values.

If we apply the rule of thumb, how many neurons should our neural network have in the hidden layer?

- Two neurons
- Eight neurons
- Six neurons

Check Answer

Finish ▸

Applying the neuron rule of thumb, we should use a neural network model with six neurons in the hidden layer to properly model our linear and nonlinear datasets. Thankfully, designing a neural network with more layers is the exact same workflow as before, except we tweak a few parameters.

Since we want to change the structure of our neural network model, we must first create a new Sequential model by adding and running the following code to our notebook:

```
# Generate our new Sequential model
new_model = tf.keras.models.Sequential()
```

Using our new Sequential model, we'll add our input, hidden, and output layers using the `Dense` class. However, this time we'll create a hidden layer with six neurons instead of one by changing the `units` parameter while keeping all other parameters the same. To our notebooks, we'll add and run the following code:

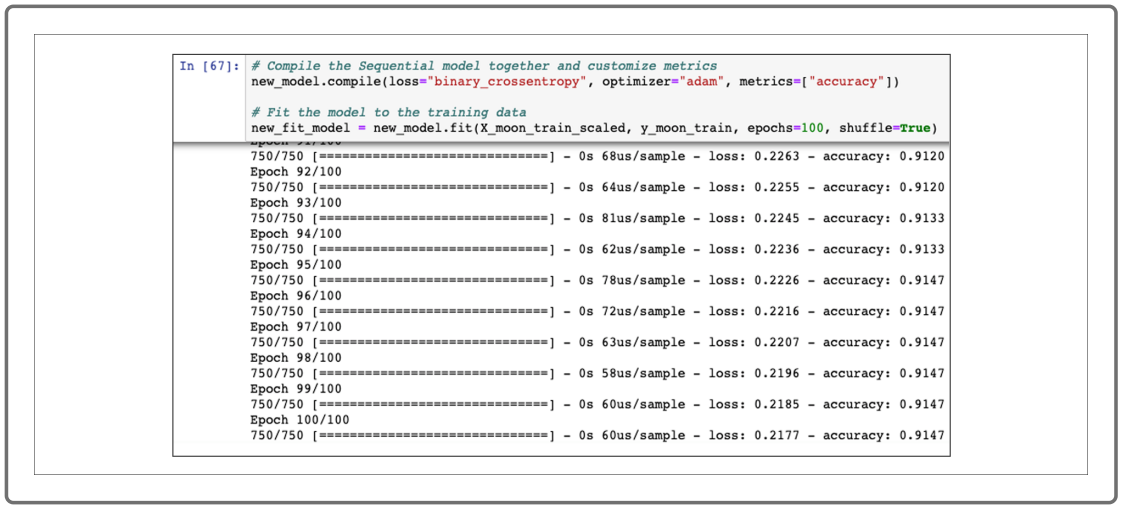
```
# Add the input and hidden layer
number_inputs = 2
number_hidden_nodes = 6
```

```
new_model.add(tf.keras.layers.Dense(units=number_hidden_nodes, activation="r

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

Now that our new Sequential model is defined, we need to compile and train the model using our nonlinear moon-shaped dummy data. For us to make direct comparisons with our single-neuron model, we'll use the same training parameters, including loss metric, optimizer function, and number of epochs.

Again, we'll add and run the following code to our notebooks:

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

# Fit the model to the training data
new_fit_model = new_model.fit(X_moon_train_scaled, y_moon_train, epochs=100,
```



```
In [67]: # Compile the Sequential model together and customize metrics
         new_model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])

         # Fit the model to the training data
         new_fit_model = new_model.fit(X_moon_train_scaled, y_moon_train, epochs=100, shuffle=True)
         750/750 [==============================] - 0s 68us/sample - loss: 0.2263 - accuracy: 0.9120
         Epoch 92/100
         750/750 [==============================] - 0s 64us/sample - loss: 0.2255 - accuracy: 0.9120
         Epoch 93/100
         750/750 [==============================] - 0s 81us/sample - loss: 0.2245 - accuracy: 0.9133
         Epoch 94/100
         750/750 [==============================] - 0s 62us/sample - loss: 0.2236 - accuracy: 0.9133
         Epoch 95/100
         750/750 [==============================] - 0s 78us/sample - loss: 0.2226 - accuracy: 0.9147
         Epoch 96/100
         750/750 [==============================] - 0s 72us/sample - loss: 0.2216 - accuracy: 0.9147
         Epoch 97/100
         750/750 [==============================] - 0s 63us/sample - loss: 0.2207 - accuracy: 0.9147
         Epoch 98/100
         750/750 [==============================] - 0s 58us/sample - loss: 0.2196 - accuracy: 0.9147
         Epoch 99/100
         750/750 [==============================] - 0s 60us/sample - loss: 0.2185 - accuracy: 0.9147
         Epoch 100/100
         750/750 [==============================] - 0s 60us/sample - loss: 0.2177 - accuracy: 0.9147
```

Looking at the training metrics of our new model, as we increase the number of neurons within the hidden layer, the classification accuracy improves. You may have noticed that adding multiple neurons to our neural network did not yield a perfect classification model. As input data

becomes more complex, neural networks will require more and more optimization tweaks to achieve their desired accuracy.

19.2.6 Give Your Model a Synaptic Boost

Beks is almost ready for a break—she's come a long way in just one day of work! Before she signs off, she decides to explore if there are any quick-and-dirty methods to give her model a little boost.

As with all machine learning algorithms, neural networks are not perfect and will often underperform using a basic implementation. When a neural network model does not meet performance expectations, it is usually due to one of two causes: inadequate or inappropriate model design for a given dataset, or insufficient or ineffective training data. Although collecting more training/test data is almost always beneficial, it may be impossible due to budget or logistical limitations. Therefore, the most straightforward means of improving neural network performance is tweaking the model design and parameters.

When it comes to tweaking a neural network model, a little can go a long way. If we tweak too many design aspects and parameters at once, we can cause a model to become less effective without a means of understanding why. To avoid trapping ourselves in endless optimization

iterations, we can use characteristics of our input data to determine what parameters should be changed.

There are a few means of optimizing a neural network:

- Check out your input dataset.
- Add more neurons to a hidden layer.
- Add additional hidden layers.
- Use a different activation function for the hidden layers.
- Add additional epochs to the training regimen.

Check Out Your Input Dataset

Before you start down the endless journey of model optimization, it is always a good idea to check the input data and ensure that there are no variables or set of outliers that are causing the model to be confused. Although neural networks are tolerant of noisy characteristics in a dataset, neural networks can learn bad habits (like the brain does). Even if we standardize and scale our numerical variables, too many outliers in a single variable can lead to performance issues.

NOTE

> We will discuss standardizing and scaling numerical features in great detail later in the module.

SKILL DRILL

Try plotting a variable using Pandas' Series.plot method to look for outliers that can help identify if a particular numerical variable is causing confusion in a model. Try leaving out a noisy variable from the rest of the training features and see if the model performs better.

Add More Neurons and Hidden Layers

REWIND

Previously, we explored how to optimize a neural network by adding neurons to the hidden layer. Adding neurons to a hidden layer has diminishing returns—more neurons means more data as well as a risk to overfitting the model.

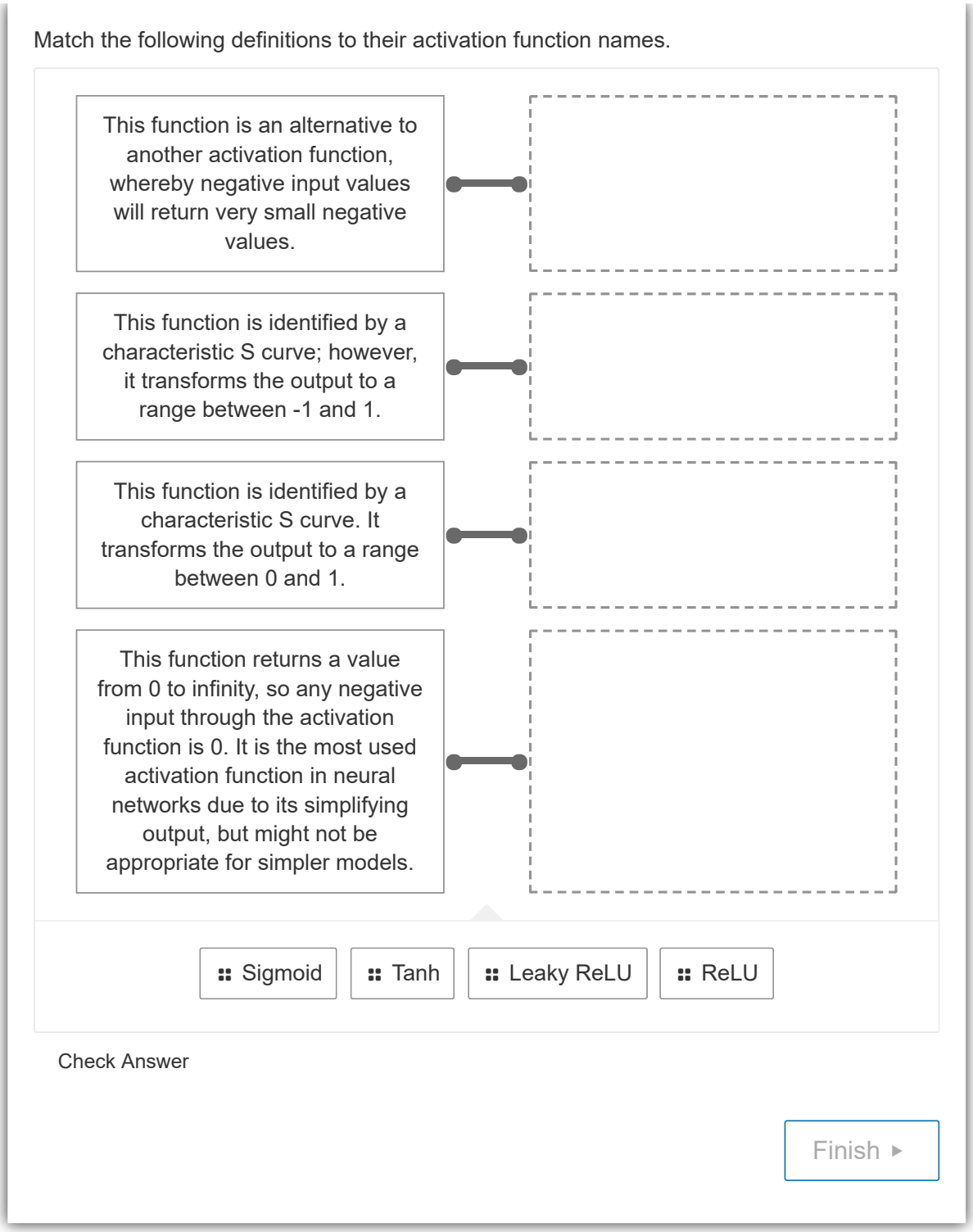
Instead of adding more neurons, we could change the structure of the model by adding additional hidden layers, which allows neurons to train on activated input values, instead of looking at new training data. Therefore, a neural network with multiple layers can identify nonlinear characteristics of the input data without requiring more input data.

IMPORTANT

> This concept of a multiple-layered neural network is known as a **deep learning neural network**. We'll be exploring deep learning neural networks in greater detail later in the module.

Use a Different Activation Function

Another strategy to increase performance of a neural network is to change the activation function used across hidden layers. Depending on the shape and dimensionality of the input data, one activation function may focus on specific characteristics of the input values, while another activation function may focus on others.



It is important to use an activation function that matches the complexity of the input data. If we wanted to rank the four most-used activation functions by data complexity and ideal use case, the order would be as follows:

1. The **sigmoid function** values are normalized to a probability between 0 and 1, which is ideal for binary classification.

2. The **tanh function** can be used for classification or regression, and it expands the range between -1 and 1.
3. The **ReLU function** is ideal for looking at positive nonlinear input data for classification or regression.
4. The **Leaky ReLU function** is a good alternative for nonlinear input data with many negative inputs.

NOTE

> By default, the Keras Dense layer will implement the **linear** activation function, which means that the net sum value is not transformed. In other words:
>
> $$\alpha(x) = x$$
>
> The linear activation function limits the neural network model to only perform a linear regression. Therefore, the linear activation function is only appropriate for an output layer.

To experiment and optimize using an activation function, try selecting from activation functions that are slightly more complex than your current activation function. For example, if you were trying to build a regression neural network model using a wide input dataset, you might start with a tanh activation function. To optimize the regression model, try training with the ReLU activation function, or even the Leaky ReLU activation function. In most cases, it is better to try optimizing using a higher complexity activation function rather than a lower complexity activation function. Using a higher complexity activation function will assess the input data differently without any risk of censoring or ignoring lower complexity features.

Add Additional Epochs to Training Regimen

If your model still requires optimizations and tweaking to meet desired performance, you can increase the number of epochs, or training iterations. As the number of epochs increases, so does the amount of information provided to each neuron. By providing each neuron more information from the input data, the neurons are more likely to apply more effective weight coefficients. Adding more epochs to the training parameters is not a perfect solution—if the model produces weight coefficients that are too effective, there is an increased risk of model overfitting. Therefore, models should be tested and evaluated each time the number of epochs are increased to reduce the risk of overfitting.

As with all machine learning models, creating an ideal classification or regression model is part mathematics and part art. As we design more and more models, optimizing and fine-tuning becomes less trial and error and more pattern recognition.

SKILL DRILL

Go ahead and use the TensorFlow Playground and dummy datasets to practice building basic neural networks. In addition, try saving multiple neural network models in the same script and compare the performance of each model, asking the following questions:

- What is the accuracy of my model? Is it acceptable or does it need to be higher?
- How long did it take to train my model? How many minutes or hours? How many epochs?
- Does it look like my model is as complex as my input data? These reflective questions will help you identify what steps are needed to make your neural network (and other machine learning and statistical) models even better!

19.3.1 Measure Twice, Model Once

Beks has been at the foundation for five years, so by now she knows that writing the checks is the easy part. The hard part is figuring out how much they should be and to whom they should go! In the same way, most of the work in building complicated models doesn't go into the actual coding. Instead, it's figuring out what is going on with the data, cleaning it, and getting it organized. Luckily for Alphabet Soup, Beks always does her due diligence.

When building a computational model, most of the design effort is not writing code to build the complex model. Rather, most of the effort in computational model building is preprocessing and cleaning up the input data. Neural networks are no exception to this rule. In fact, neural networks tend to require the most preprocessing of input data compared to all other statistical and machine learning models. This is because neural networks are really good at identifying patterns and trends in data; therefore, they are susceptible to getting stuck when looking at abstract or raw data. When data has many categorical values, or large gaps between numerical values, a neural network might think that these variables are less important (or more important) than they really are. As a result, the

neural network may ignore other variables that should provide more meaningful information to the model.

For example, if a bank wanted to build a neural network model to identify if a company was eligible for a loan, it might look at factors such as a company's net worth. If the bank's input dataset contained information from large fortune 500 companies, such as Google and Facebook, as well as small mom-and-pop stores, the variability in net worth would be outrageous. Without normalizing the input data, a neural network could look at net worth as being a strong indicator of loan eligibility, and as a result, could ignore all other factors, such as debt-to-income ratio, credit status, or requested loan amount. Instead, if the net worth was normalized on a factor such as number of employees, the neural network would be more likely to weigh other factors more evenly to net worth. This would result in a neural network model that assesses loan eligibility more fairly, without introducing any additional risk.

In the next few pages, we'll look at different preprocessing steps that can prepare input data for training neural network models. By the end of this section, we'll no longer need to use dummy input data—we'll be able to apply neural network models to any dataset!

19.3.2 The Headache of Categorical Variables

Beks knows all about categorical and numerical variables. After all, she did graduate from a 24-week data analytics boot camp! However, she isn't quite sure about how neural networks handle various variables. What type of preprocessing might she need to perform for categorical variables? Time to find out!

Unlike the statistical models such as multiple linear regression, or machine learning models such as random forest, neural networks cannot handle categorical variables in their raw form. Specifically, the perceptron neuron has no way to segment and keep track of all possible values in a categorical variable. So, what happens if our datasets contain categorical variables that are essential for identifying samples or groups of samples within the population data? Thankfully, there are straightforward solutions to grouping and encoding categorical variables without losing any information across neurons.

For a neural network to understand and evaluate a categorical variable, we must preprocess the values using a technique called **one-hot encoding**. One-hot encoding identifies all unique column values and splits the single categorical column into a series of columns, each containing information

about a single unique categorical value. For example, let's imagine Beks received a dataset from a medical device company that contains clinical trial data. One of the variables in this dataset is a categorical variable that identifies the person's eye color:

Eye_Color
Blue
Brown
Brown
Brown
Hazel
Green
Hazel
Brown
Blue
Brown

If Beks were to apply the one-hot encoding to this categorical variable, it would break into four separate columns—one for each unique eye color (blue, brown, hazel, and green). Therefore, the newly encoded data columns would look as shown:

Eye_Color:Blue	**Eye_Color:Brown**	**Eye_Color:Hazel**	**Eye_Color:Green**
1	0	0	0
0	1	0	0

0	1	0	0
0	1	0	0
0	0	1	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0
0	1	0	0

For each unique column value, every individual data point is evaluated, and if the categorical value matches, it is given the value of 1, otherwise it is 0. This binary encoding ensures that each neuron receives the same amount of information from the categorical variable. As a result, the neural network will interpret each value individually and provide each categorical value with an independent weight.

Although one-hot encoding is a very robust solution, it can be very memory-intensive. Therefore, categorical variables with a large number of unique values (or very large variables with only a few unique values) might become difficult to navigate or filter once encoded. To address this issue, we must reduce the number of unique values in the categorical variables. The process of reducing the number of unique categorical values in a dataset is known as **bucketing** or **binning**. Bucketing data typically follows one of two approaches:

1. Collapse all of the infrequent and rare categorical values into a single “other” category.
2. Create generalized categorical values and reassign all data points to the new corresponding values.

The first bucketing approach takes advantage of the fact that uncommon categories and “edge cases” are rarely statistically significant. Therefore,

regression and classification models are unlikely to be able to use rare categorical values to produce robust models, and instead will ignore the rare events altogether and focus on more informative values.

The second bucketing approach collapses the number of unique categorical values and maintains relative order and magnitude so that the machine learning model can train on the categorical variable with minimal impact to performance. This approach is particularly useful when dealing with a categorical variable whose distribution of unique values is relatively even. Once we have bucketed our categorical variables, we can proceed to transform the categorical variable using one-hot encoding.

Say we have a list of categorical values, such as the following:

Green
Red
White
Red
Black
Red
Red
White
Green
White
Black
Black
Green
White
Black
Red
Green
Green
Black
Red

What would be an appropriate bucketing strategy?

- Collapse all infrequent and rare values into a single “other” category.
- Create more generalized categorical variables, such as “color” and “no color.”

Check Answer

Finish ▸

19.3.3 Practice Encoding Categorical Variables

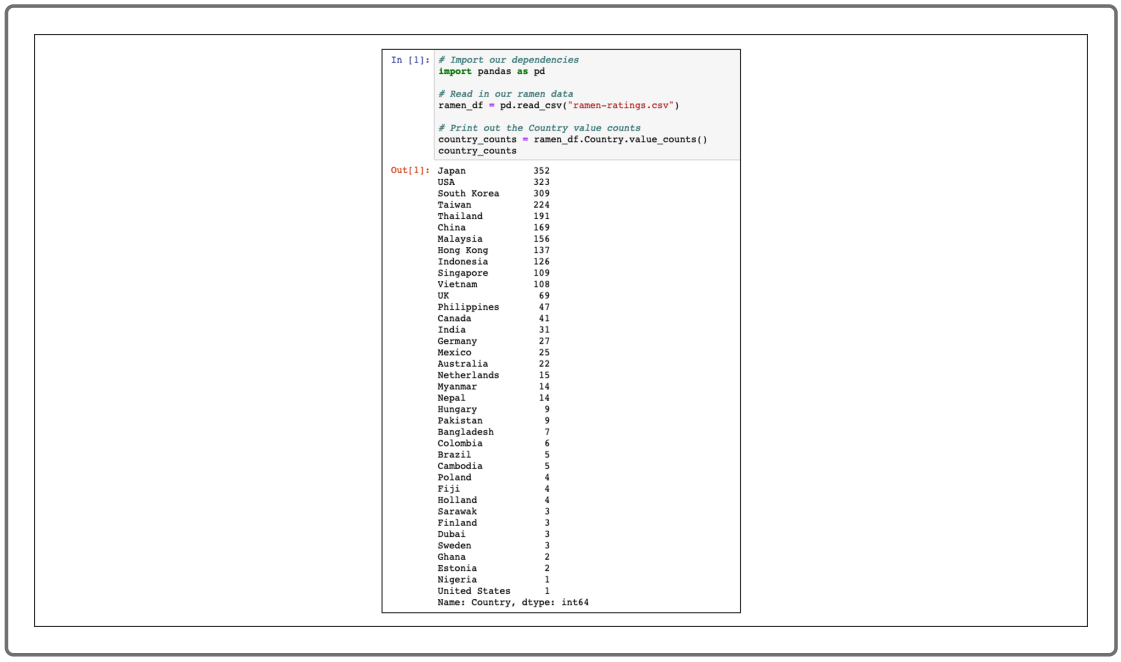
Whew! Turns out it's a good thing Beks dug into how neural networks handle categorical data of various types—she'll need to make sure she uses the one-hot encoding method with any categorical variables in the dataset.

However Beks knows reading about one-hot encoding is one thing - implementing one-hot encoding is another.

To better understand the process of transforming and encoding categorical variables, let's practice implementing a one-hot encoder through Scikit-learn. First, download this **categorical variable CSV file (ramen-ratings.csv) (https://courses.bootcampspot.com/courses/138/files/28482/download?wrap=1)**  **(https://courses.bootcampspot.com/courses/138/files/28482/download?wrap=1)** and place it in the same folder as your Jupyter Notebook. This dataset from **The Ramen Rater (https://www.theramenrater.com/)** contains more than 2,500 ratings of different ramen dishes around the world. In a hypothetical scenario, we need to use the "country" variable as a categorical variable in a larger dataset that will predict restaurant

satisfaction. Before we can convert “country” into a one-hot encoding, we need to make sure that there are not too many unique values that would cause our dataset to become too wide. The easiest way to check for unique values is to use the Pandas DataFrame’s `value_counts` method. Let’s start by adding the following code to the notebook:

```
# Import our dependencies
import pandas as pd
import sklearn as skl

# Read in our ramen data
ramen_df = pd.read_csv("ramen-ratings.csv")

# Print out the Country value counts
country_counts = ramen_df.Country.value_counts()
country_counts
```



Looking at the country’s unique value counts, there are a number of countries that appear frequently in the dataset such as Japan, USA, and South Korea. However, many countries appear semi-frequently and even more rarely appear in the dataset.

Why would we want to bucket the rare values into the “other” category?

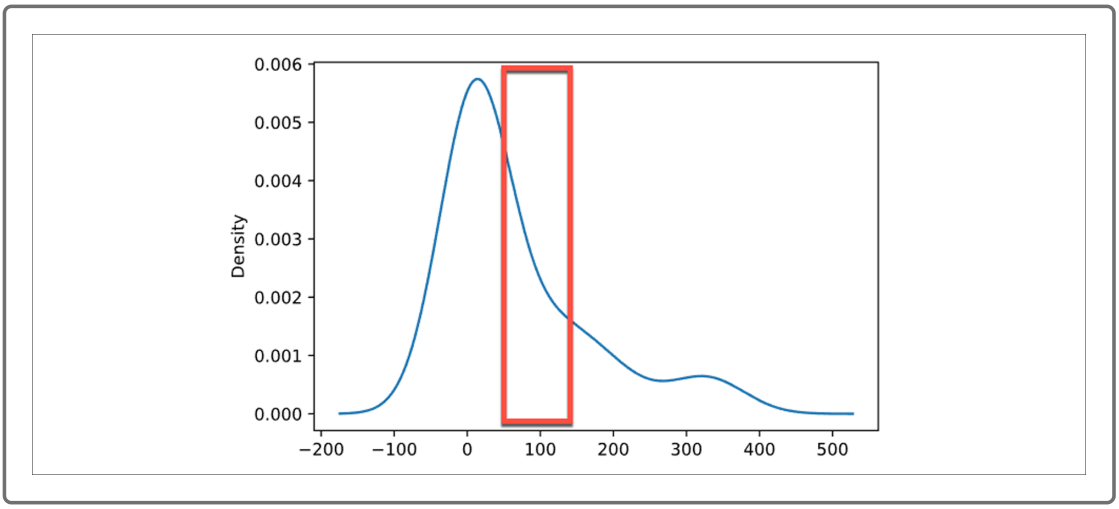
- There are many unique values that can be grouped together logically.
- There are many unique values, and there is an uneven distribution.
- The neural network model cannot handle some of the unique values.

Check Answer

Finish ▸

How do we determine which countries are uncommon enough to bucket into the “other” category? The most straightforward method is to use a density plot to identify where the value counts “fall off” and set the threshold within this region. To produce a density plot in Pandas, add and run the following code:

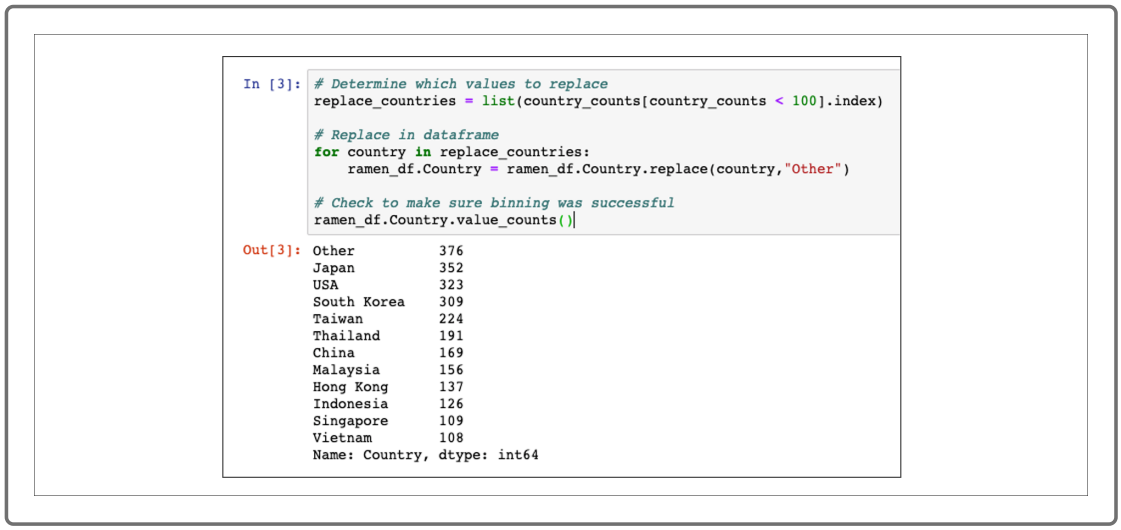
```
# Visualize the value counts
country_counts.plot.density()
```



According to the density plot, the most common unique values have more than 100 instances within the dataset. Therefore, we can bucket any country that appears fewer than 100 times in the dataset as “other.” To do

this, we'll use a Python `for` loop and Pandas' `replace` method. To our notebooks, we need to add and run the following code:

```
# Determine which values to replace
replace_countries = list(country_counts[country_counts < 100].index)

# Replace in DataFrame
for country in replace_countries:
    ramen_df.Country = ramen_df.Country.replace(country,"Other")


# Check to make sure binning was successful
ramen_df.Country.value_counts()
```



Now that we have reduced the number of unique values in the country variable, we're ready to transpose the variable using one-hot encoding. The easiest way to perform one-hot encoding in Python is to use Scikit-learn's `OneHotEncoder` module on the country variable. To build the encoded columns, we must create an instance of `OneHotEncoder` and "fit" the encoder with our values. Again, we must add and run the following code:

```
# Create the OneHotEncoder instance
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder(sparse=False)
```

```
# Fit the encoder and produce encoded DataFrame
encode_df = pd.DataFrame(enc.fit_transform(ramen_df.Country.values.reshape(-

# Rename encoded columns
encode_df.columns = enc.get_feature_names(['Country'])
encode_df.head()
```

	Country_China	Country_Hong Kong	Country_Indonesia	Country_Japan	Country_Malaysia
0	0.0	0.0	0.0	1.0	0.0
1	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

We must join the encoded DataFrame with the original and drop the original “Country” column. The process of joining the two DataFrames together is handled by the Pandas `merge` method and can be performed within one line. Finally, we’ll add and run the final code block:

```
# Merge the two DataFrames together and drop the Country column
ramen_df.merge(encode_df,left_index=True,right_index=True).drop("Country",1)
```

In [5]:
```
# Merge the two dataframes together and drop the Country column
ramen_df.merge(encode_df,left_index=True,right_index=True).drop("Country",1)
```

Out[5]:

	Review #	Brand	Variety	Style	Stars	Top Ten	Country_China	Country_Hong Kong	Country_Indonesia	Country_Japan
0	2580	New Touch	T's Restaurant Tantanmen	Cup	3.75	NaN	0.0	0.0	0.0	1.0
1	2579	Just Way	Noodles Spicy Hot Sesame Spicy Hot Sesame Guan...	Pack	1	NaN	0.0	0.0	0.0	0.0
2	2578	Nissin	Cup Noodles Chicken Vegetable	Cup	2.25	NaN	0.0	0.0	0.0	0.0
3	2577	Wei Lih	GGE Ramen Snack Tomato Flavor	Pack	2.75	NaN	0.0	0.0	0.0	0.0
4	2576	Ching's Secret	Singapore Curry	Pack	3.75	NaN	0.0	0.0	0.0	0.0

Although this ramen dataset is just an example, the process of bucketing and encoding are two of the most common preprocessing steps required to set up your neural network training datasets. As with many other data science techniques, it might seem complicated at first, but it will become easier with practice.

19.3.4 Span the Gap Using Standardization

Nice! Now that Beks has finished implementing the one-hot encoding on some practice data, she can encode any categorical data she finds.

Now, what other issues could potentially come up in her dataset?

Unlike categorical data, neural network models can interpret and evaluate all forms of numeric data. In other words, if our input data has no categorical data types, it can be provided to a neural network model in its raw form. Even though a neural network *can* train on raw numerical data, it does not mean that it *should* train on raw data. There are many reasons why a raw numeric variable is insufficient for use when training a neural network model, such as:

- Raw data often has outliers or extreme values that can artificially inflate a variable's importance.
- Numerical data can be measured using different units across a dataset—such as time versus temperature, or length versus volume.
- The distribution of a variable can be skewed, leading to misinterpretation of the central tendency.

If we use raw numeric data to train a neural network model, there is a chance that the neural network model will perform adequately. However, there is a far greater probability that the neural network model will interpret the raw numerical data inappropriately, which will yield an inadequate model. Thankfully, we can minimize this risk by standardizing (also commonly referred to as normalization) the numerical data prior to training.

REWIND

Scikit-learn's `StandardScaler` module standardizes numerical data such that a variable is rescaled to a mean of 0 and standard deviation of 1.

If we use the `StandardScaler` module to standardize our numerical variables, we reduce the overall likelihood that outliers, variables of different units, or skewed distributions will have a negative impact on a model's performance.

Standardization Practice

To reinforce and practice standardizing data, let's implement Scikit-learn's `StandardScaler` module. First, **download the hr_dataset.csv file (https://courses.bootcampspot.com/courses/138/files/28478/download?wrap=1)**  **(https://courses.bootcampspot.com/courses/138/files/28478/download?wrap=1)** and place it in the same folder as your Jupyter Notebook.

This dataset is from **Kaggle (https://www.kaggle.com/inugami/dataset-on-company-clients-satisfaction)** and contains employee metrics such as satisfaction level, number of projects completed, time spent at the

company, and number of promotions. In a hypothetical scenario, we could use this data to determine whether or not an employee is eligible for a bonus. This dataset has four variables, each using different numerical scales and units. If we want to use this dataset for a neural network model, we need to standardize these variables, otherwise it is likely that "Satisfaction_Level" (which ranges between 0 and 1) would be undervalued compared to "Time_Spent" (which ranges from approximately 50 to 4,800 hours). Let's start by adding the following code to the notebook:

```
# Import our dependencies
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Read in our dataset
hr_df = pd.read_csv("hr_dataset.csv")
hr_df.head()
```

In [1]:
```
# Import our dependencies
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Read in our dataset
hr_df = pd.read_csv("hr_dataset.csv")
hr_df.head()
```

Out[1]:

	Satisfaction_Level	Num_Projects	Time_Spent	Num_Promotions
0	0.30	1	253	2
1	0.25	1	200	0
2	0.90	4	2880	5
3	0.65	3	1450	3
4	0.50	2	785	2

If this dataset contained categorical data, what would we need to do with the data?

- Slice out the categorical data prior to scaling.
- Slice out the categorical data after scaling.

Check Answer

Finish ▸

To apply our standardization, we need to create a `StandardScaler` instance by adding and running the following code:

```
# Create the StandardScaler instance
scaler = StandardScaler()
```

Once we have our `StandardScaler` instance, we need to fit the input data by adding and running the next line of code:

```
# Fit the StandardScaler
scaler.fit(hr_df)
```

After our `StandardScaler` instance is fitted with the numerical data, we can transform and standardize the dataset using the following code:

```
# Scale the data
scaled_data = scaler.transform(hr_df)
```

Lastly, once we have our transformed data within the `StandardScaler` instance, we must export the transformed data into a Pandas DataFrame. Again, we must add and run the following code to our notebooks:

```
# Create a DataFrame with the scaled data
transformed_scaled_data = pd.DataFrame(scaled_data, columns=hr_df.columns)
transformed_scaled_data.head()
```

In [6]:
```
# Create a DataFrame with the scaled data
transformed_scaled_data = pd.DataFrame(scaled_data, columns=hr_df.columns)
transformed_scaled_data.head()
```

Out[6]:

	Satisfaction_Level	Num_Projects	Time_Spent	Num_Promotions
0	-1.303615	-1.162476	-1.049481	-0.558656
1	-1.512945	-1.162476	-1.094603	-1.804887
2	1.208335	0.860233	1.187080	1.310692
3	0.161689	0.185996	-0.030385	0.064460
4	-0.466299	-0.488240	-0.596549	-0.558656

Looking at the results of our transformation, all of the variables have now been standardized, with a mean value of 0 and a standard deviation of 1. Now the data is ready to be passed along to our neural network model.

19.4.1 Unleash the Hidden Potential of Neural Networks

Beks has come a long way in a short time—having started the project with almost no knowledge of neural networks, she can now build a neural network by herself!

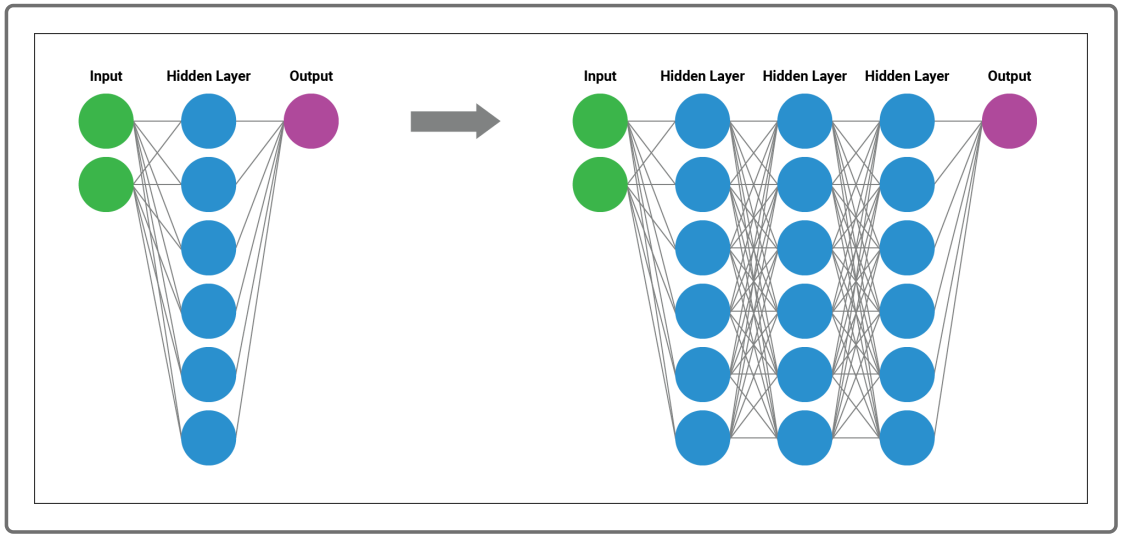
But if being in that boot camp taught her anything, it's that sometimes the most powerful models are just around the corner.

What might she be able to do with a deep learning neural network? She can't wait to find out.

As we have seen, neural networks are designed to calculate weights for various input data and pass along an output value based on an activation function. With our basic neural networks, input data is parsed using an input layer, evaluated in a single hidden layer, then calculated in the output layer. In other words, a basic neural network is designed such that the input values are evaluated *only once* before they are used in a classification or regression equation. Although basic neural networks are relatively easy to conceptualize and understand, there are limitations to using a basic neural network, such as:

- A basic neural network with many neurons will require more training data than other comparable statistics and machine learning models to produce an adequate model.
- Basic neural networks struggle to interpret complex nonlinear numerical data, or data with many confounding factors that have hidden effects on more than one variable.
- Basic neural networks are incapable of analyzing image datasets without severe data preprocessing.

To address the limitations of the basic neural network, we can implement a more robust neural network model by adding additional hidden layers. A neural network with more than one hidden layer is known as a **deep neural network**:



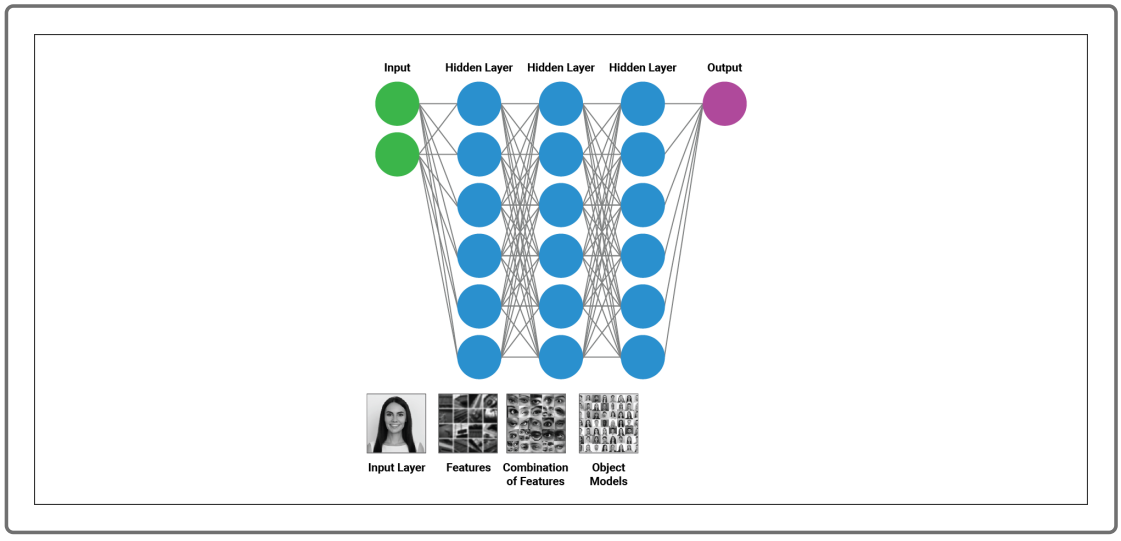
Deep neural networks function similarly to the basic neural network, with one major exception. The outputs of one hidden layer of neurons (that have been evaluated and transformed using an activation function) become the inputs to additional hidden layers of neurons. As a result, the next layer of neurons can evaluate higher order interactions between weighted variables and identify complex, nonlinear relationships across the entire dataset. These additional layers can observe and weight interactions between clusters of neurons across the entire dataset, which means they can identify and account for more information than any number of neurons in a single hidden layer.

Deep neural network models also are commonly referred to as **deep learning models** due to their ability to learn from example data, regardless of the complexity or data input type. Just like humans, deep learning models can identify patterns, determine severity, and adapt to changing input data from a wide variety of data sources. Compared to basic neural network models, which require a large number of neurons to identify nonlinear characteristics, deep learning models only need a few neurons across a few hidden layers to identify the same nonlinear characteristics.

NOTE

> Although the numbers are constantly debated, many data engineers believe that even the most complex interactions can be characterized by as few as three hidden layers.

In addition, deep learning models can train on images, natural language data, soundwaves, and traditional tabular data (data that fits in a table or DataFrame), all with minimal preprocessing and direction:

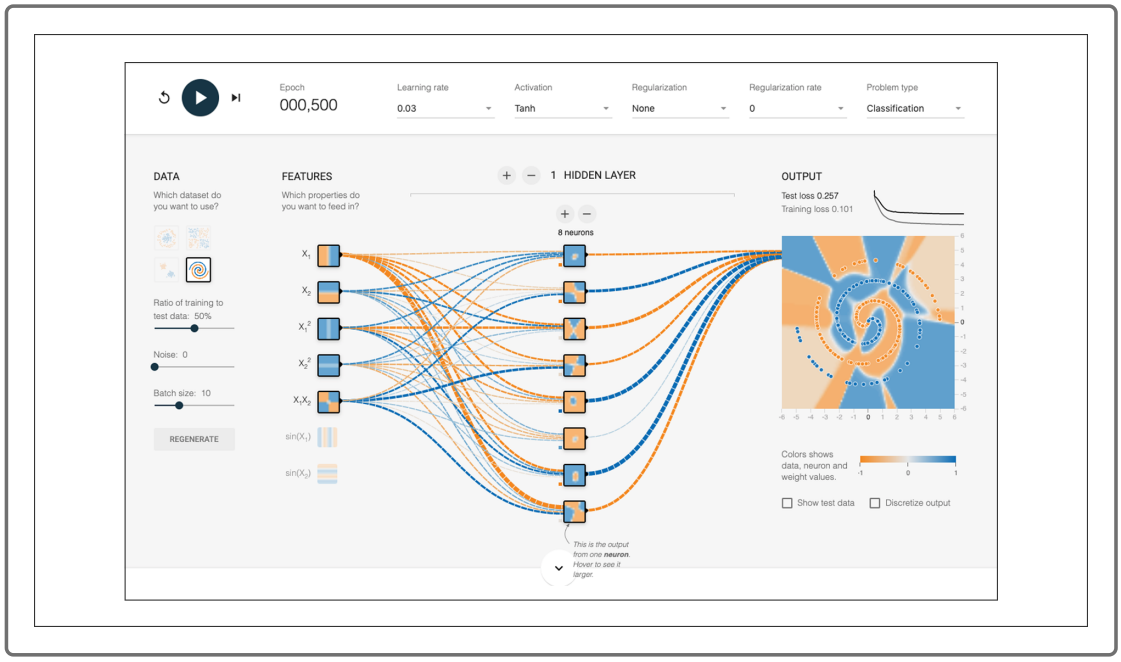


The best feature of deep learning models is its capacity to systematically process multivariate and abstract data while simultaneously achieving performance results that can mirror or even exceed human-level performance. It is no wonder why huge tech, pharmaceutical, and finance

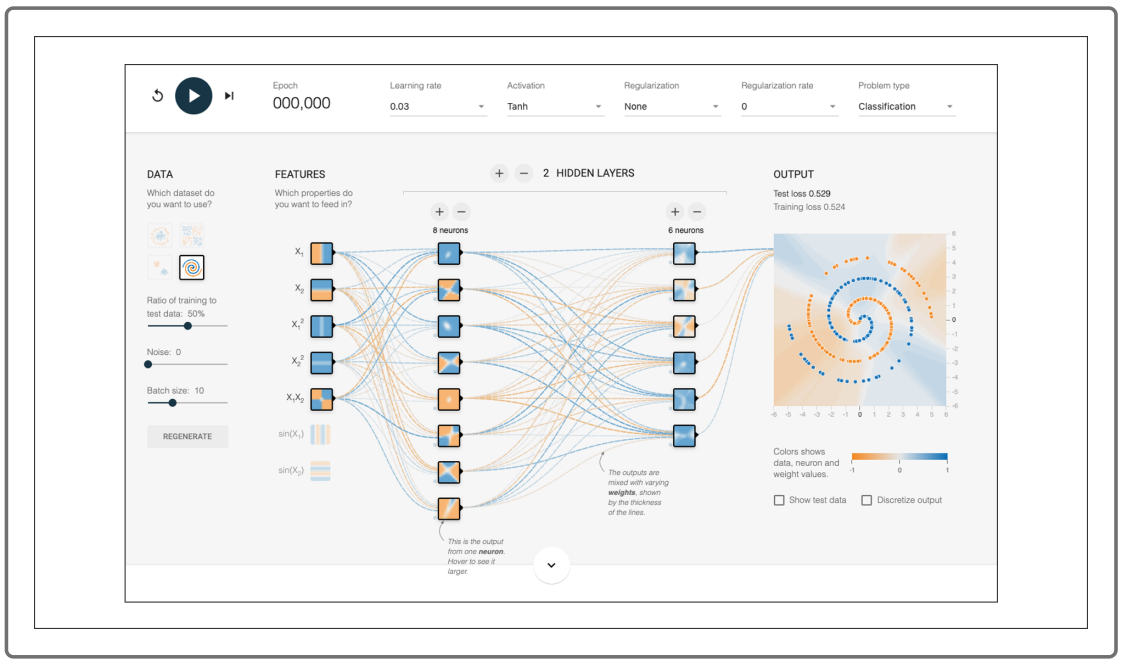
companies turn to deep learning models to evaluate and interpret datasets that were once unusable by traditional modelling techniques.

As with basic neural network models, deep learning models are not a new concept. However, deep learning models were not a feasible option for data scientists until implementation became easier with libraries like TensorFlow, and computing power became more affordable. Deep learning models require significantly longer training iterations and memory resources than their basic neural network counterparts, which allow for the deep learning models to achieve higher degrees of accuracy and precision. In other words, deep learning models may have more upfront costs, but they also have higher performance potential.

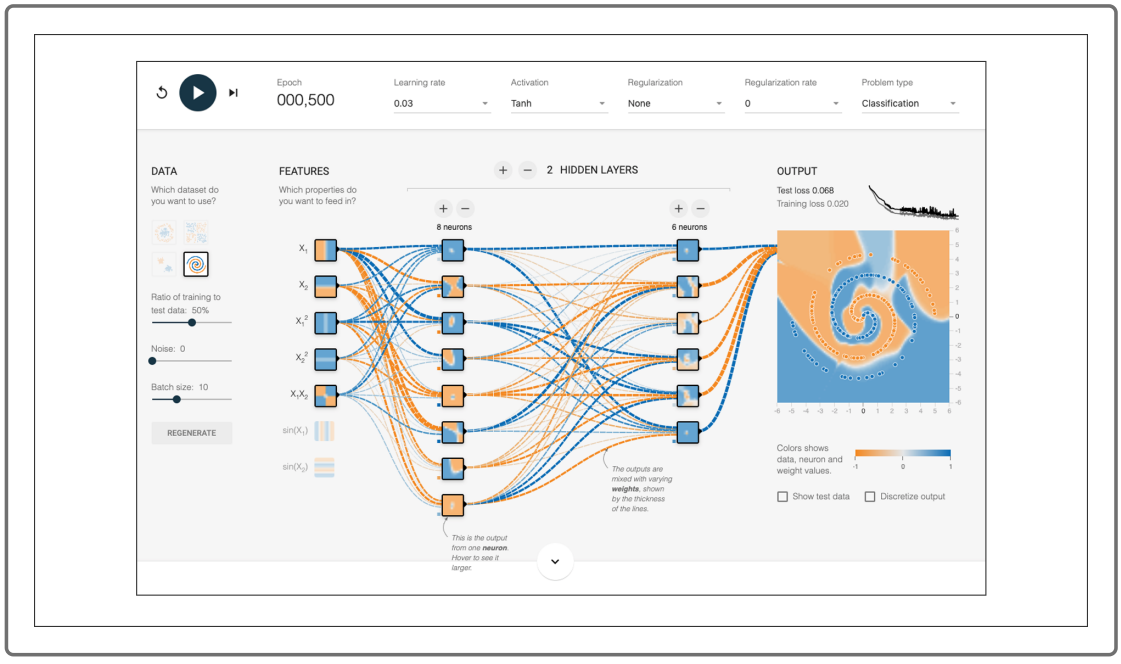
To conceptualize how performance differs between the basic neural network model versus a deep learning model, we'll return to the **TensorFlow Playground (https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=spiral&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=8&seed=0.14370&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=true&xSquared=true&ySquared=true&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false)** , where our TensorFlow neural network model will try to classify a far more complex dataset such as a spiral. Since we're trying to build a binary classifier on nonlinear data, we'll use the tahn activation function rather than the sigmoid activation function. Additionally, we'll look at a couple of other mathematical inputs such as X1^2, X2^2, and X1X2, which will allow for our neural network model to train and identify patterns using nonlinear inputs. To start, let's allow our model to train over roughly 500 epochs:



Notice that by using a single hidden layer over 500 epochs, the model does a decent job building a classification model with loss around 0.2. In some cases this model would be sufficient, but when it comes to industry leaders such as Google and Apple, model performance must be near perfection. Therefore, we must try and build a more robust model by designing a deep learning model with two layers. In your TensorFlow Playground, add an additional hidden layer with six neurons—these will analyze the outputs of our eight neurons in the first layer to try and boost performance:



Now that we have our updated deep learning model, let's try to train the neural network over the same 500 epochs:



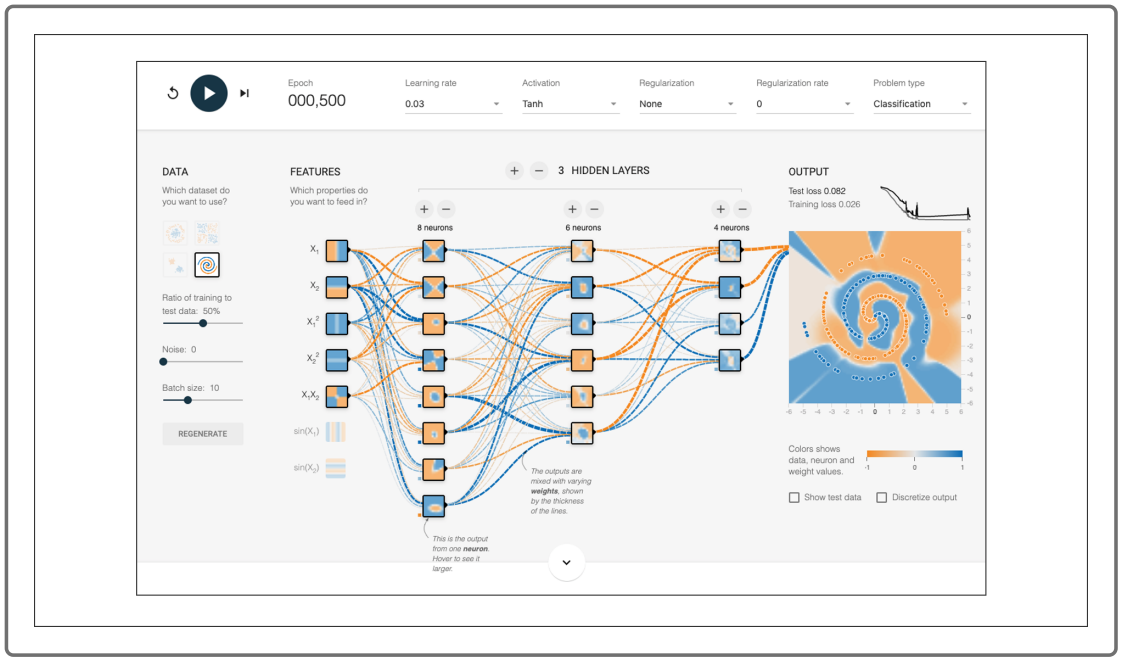
Notice that our deep learning model was able to reduce the loss from roughly 0.2 to 0.07, which could mean the difference of 80% classification accuracy to 95% and above! Let's try adding an additional layer with four more neurons and train the new model over 500 epochs.

Does adding additional layers substantially improve model performance?

- No.
- Yes.

Check Answer

Finish ▸



Looking at the results of our simulated deep learning model, it does not appear that adding more layers increased the overall performance of the model. This is because the additional layer was redundant—the complexity of the dataset was encapsulated within the two hidden layers. Adding layers does not always guarantee better model performance, and depending on the complexity of the input data, adding more hidden layers will only increase the chance of overfitting the training data. Unfortunately, there is no easy solution or rule of thumb to identify how many layers are required to maximize performance. The only way to determine how “deep” the deep learning model should be is through trial and error. You must train and evaluate a model with increasingly deeper and deeper layers until

the model no longer demonstrates noticeable improvements over the same number of epochs.

When would be a good scenario to implement a deep learning model? Select all scenarios that apply:

- [ ] When other machine learning models are unable to meet desired performance
- [ ] When you need a model capable of analyzing a complex dataset with many features and data points
- [ ] When you need to classify nontabular data such as images, voice, and written text

Check Answer

Finish ▸

19.4.2 Real Data, Real Practice—Imports and Setup

Whoa! Beks is certainly glad she looked into deep learning models—they are very promising, especially given the complex dataset she will need to analyze to help the foundation predict good investments. Of course, first she'll want to practice on a different dataset.

Now that we understand the capabilities of deep learning models and their application in data science, it is time to try implementing some deep learning models of our own. As with basic neural network models, the more we implement deep learning models across a wide variety of data, the easier the process becomes. In this section, we'll implement our very first deep learning classification model using tabular input data. Throughout this implementation walkthrough, take some time to critically think about the following:

- What about this dataset makes it complex? Is it a variable? Is it the distribution of values? Is it the size of the dataset?
- Which variables should I investigate prior to implementing my model? What does the distribution look like? Hint: Use Pandas' `Series.plot.density()` method to find out.

- What outcome am I looking for from the model? Which activation function should I use to get my desired outcome?
- What is my accuracy cutoff? In other words, what percent testing accuracy must my model exceed?

For our first deep learning example, we'll look at another human resources (HR) dataset from **IBM's HR department (https://www.kaggle.com/pavansubhasht/ibm-hr-analytics-attrition-dataset)** . This dataset contains employee career profile and attrition (departure from company) information, such as age, department, job role, work/life balance, and so forth. Using this data, we can generate a deep learning model that can help to identify whether or not a person is likely to depart from the company given his or her current employee profile.

This dataset has more than 30 variables, each with different numerical distributions and categorical variables, as well as some dichotomous variables that will need transformation—this dataset truly has it all. By the end of this section, you'll have built a robust deep learning model that is capable of training and classifying on a complex real-world dataset!

First, we'll start by downloading this **HR dataset (HR-Employee-Attrition.csv) (https://courses.bootcampspot.com/courses/138/files/28485/download?wrap=1)**  **(https://courses.bootcampspot.com/courses/138/files/28485/download?wrap=1)** and placing it in a folder with a new Jupyter Notebook. Name your new Jupyter Notebook "DeepLearning_Tabular" (or something similar) that will allow you to easily locate this walkthrough at another time. Once we have created our notebook and placed the dataset into the corresponding folder, we'll start by importing our libraries and reading in the dataset. Copy and run the following code into your notebook:

```
# Import our dependencies
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler,OneHotEncoder
import pandas as pd
import tensorflow as tf
```

```
# Import our input dataset
attrition_df = pd.read_csv('HR-Employee-Attrition.csv')
attrition_df.head()
```

Out[1]:

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	EmployeeCo
0	41	Yes	Travel_Rarely	1102	Sales	1	2	Life Sciences	
1	49	No	Travel_Frequently	279	Research & Development	8	1	Life Sciences	
2	37	Yes	Travel_Rarely	1373	Research & Development	2	2	Other	
3	33	No	Travel_Frequently	1392	Research & Development	3	4	Life Sciences	
4	27	No	Travel_Rarely	591	Research & Development	2	1	Medical	

5 rows × 35 columns

Which preprocessing step should you start with when preparing data for a deep learning model?

- Standardize all numerical data.
- Encode all categorical data.

Check Answer

Finish ▸

Looking at the top of our DataFrame, there are multiple columns with categorical values as well as our numerical values. To make things easier, we should generate a list of categorical variable names. Instead of searching across all 35 columns and keeping track of which variables need categorical preprocessing, we'll let Python do all of the heavy lifting. As an added bonus, we can use our variable list to perform the one-hot encoding once, rather than for each individual variable. To generate our variable list, we'll use Pandas `Dataframe.dtypes` property. Add and run the following code in your notebook:

```
# Generate our categorical variable list
attrition_cat = attrition_df.dtypes[attrition_df.dtypes == "object"].index.t
```

19.4.3 Real Data, Real Practice—Preprocessing

Just as she suspected, Beks will need to start her preprocessing by one-hot encoding her categorical data.

Now that we have our variable names separated, we can start to preprocess our data starting with the one-hot encoding of the categorical data. Looking at our `attrition_cat` variable, there are eight categorical variables that need encoding. However, before we loop through our variables and encode them using Scikit-learn's `OneHotEncoder` module, we need to make sure that none of the categorical variables have more than 10 unique values and require bucketing. To check the three variables, add and run the following code:

```
# Check the number of unique values in each column
attrition_df[attrition_cat].nunique()
```

```
In [3]: # Check the number of unique values in each column
        attrition_df[attrition_cat].nunique()

Out[3]: Attrition          2
        BusinessTravel     3
        Department         3
        EducationField     6
        Gender             2
        JobRole            9
        MaritalStatus      3
        Over18             1
        OverTime           2
        dtype: int64
```

According to the `nunique` method, none of the categorical variables have more than 10 unique values, which means we're ready to encode. Previously, we have encoded a single variable at a time, but Scikit-learn is flexible enough to perform all of the one-hot encodings at the same time. The only difference from our single variable example is we need to pass our `attrition_cat` variable list. Again, add and run the following code to your notebook:

```
# Create a OneHotEncoder instance
enc = OneHotEncoder(sparse=False)

# Fit and transform the OneHotEncoder using the categorical variable list
encode_df = pd.DataFrame(enc.fit_transform(attrition_df[attrition_cat]))

# Add the encoded variable names to the DataFrame
encode_df.columns = enc.get_feature_names(attrition_cat)
encode_df.head()
```

Out[4]:

	Attrition_No	Attrition_Yes	BusinessTravel_Non-Travel	BusinessTravel_Travel_Frequently	BusinessTravel_Travel_Rarely
0	0.0	1.0	0.0	0.0	1.0
1	1.0	0.0	0.0	1.0	0.0
2	0.0	1.0	0.0	0.0	1.0
3	1.0	0.0	0.0	1.0	0.0
4	1.0	0.0	0.0	0.0	1.0

5 rows × 31 columns

Now that our categorical variables have been encoded, they are ready to replace our unencoded categorical variables in our dataset.

What Pandas method(s) should you use to replace the columns in the original DataFrame?

- Merge and drop
- Merge
- Combine

Check Answer

Finish ▸

To replace these columns, we'll use a combination of Pandas' `merge` and `drop` methods. Add and run the following code in the notebook:

```
# Merge one-hot encoded features and drop the originals
attrition_df = attrition_df.merge(encode_df,left_index=True, right_index=Tru
attrition_df = attrition_df.drop(attrition_cat,1)
attrition_df.head()
```

Out[5]:

	Age	DailyRate	DistanceFromHome	Education	EmployeeCount	EmployeeNumber	EnvironmentSatisfaction
0	41	1102	1	2	1	1	2
1	49	279	8	1	1	2	3
2	37	1373	2	2	1	4	4
3	33	1392	3	4	1	5	4
4	27	591	2	1	1	7	1

5 rows × 57 columns

Next we must split our training and testing data, then standardize our numerical variables using Scikit-learn's `StandardScaler` module.

REWIND

We need to split our training and testing data before fitting our `StandardScaler` instance. This prevents testing data from influencing the standardization function.

To build our training and testing datasets, we need to separate two values:

- input values (which are our independent variables commonly referred to as **model features** or "**X**" in **TensorFlow documentation (https://www.tensorflow.org/guide/)**)
- target output (our dependent variable commonly referred to as **target** or "**y**" in TensorFlow documentation)

For our purposes, we want to build a model that will predict whether or not a person is at risk for attrition; therefore, we must separate the "Attrition" columns from the rest of the input data. In fact, because the attrition data is dichotomous (one of two values), we only need to keep the "Attrition_Yes" column—we can ignore the "Attrition_No" column because it is redundant. To separate our features and target as well as perform our training/test split, add and run the following code in your notebook:

```
# Split our preprocessed data into our features and target arrays
y = attrition_df["Attrition_Yes"].values
X = attrition_df.drop(["Attrition_Yes","Attrition_No"],1).values

# Split the preprocessed data into a training and testing dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=78)
```

Now that our training and testing data have been allocated, we're ready to build our `StandardScaler` object and standardize the numerical features. Add and run the following code to the notebook:

```
# Create a StandardScaler instance
scaler = StandardScaler()

# Fit the StandardScaler
X_scaler = scaler.fit(X_train)

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

Now that our data is preprocessed via one-hot encoding and standardization, we should probably perform a gut check to ensure that no data has been lost from our original DataFrame.

19.4.4 Real Data, Real Practice—Deep Learning Model Design

Now that the hardest part—also known as data preprocessing—is over, it's time to make the actual model. Beks takes a quick break for some more coffee and gets back to work.

At last, our data is preprocessed and separated and ready for modelling. For our purposes, we will use the same framework we used for our basic neural network:

- For our **input layer**, we must add the number of input features equal to the number of variables in our feature DataFrame.
- In our **hidden layers**, our deep learning model structure will be slightly different—we'll add two hidden layers with only a few neurons in each layer. To create the second hidden layer, we'll add another Keras `Dense` class while defining our model. All of our hidden layers will use the `relu` activation function to identify nonlinear characteristics from the input values.
- In the **output layer**, we'll use the same parameters from our basic neural network including the `sigmoid` activation function. The `sigmoid`

activation function will help us predict the probability that an employee is at risk for attrition.

To create our deep learning model, we must add and run the following code:

```
# Define the model - deep neural net
number_input_features = len(X_train[0])
hidden_nodes_layer1 =  8
hidden_nodes_layer2 = 5

nn = tf.keras.models.Sequential()

# First hidden layer
nn.add(
    tf.keras.layers.Dense(units=hidden_nodes_layer1, input_dim=number_input_
)

# Second hidden layer
nn.add(tf.keras.layers.Dense(units=hidden_nodes_layer2, activation="relu"))

# Output layer
nn.add(tf.keras.layers.Dense(units=1, activation="sigmoid"))

# Check the structure of the model
nn.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 8)                 448
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 45
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 6
=================================================================
Total params: 499
Trainable params: 499
Non-trainable params: 0
_________________________________________________________________
```

Looking at our model summary, we can see that the number of weight parameters (weight coefficients) for each layer equals the number of input values times the number of neurons plus a bias term for each neuron. Our first layer has 55 input values, and multiplied by the eight neurons (plus eight bias terms for each neuron) gives us a total of 448 weight parameters—plenty of opportunities for our model to find trends in the dataset.

If there are eight neurons in the first layer, and five neurons in the second layer, how many weight parameters exist in the second layer?

- 40 parameters
- 45 parameters

Check Answer

Finish ▸

Now it is time to compile our model and define the loss and accuracy metrics. Since we want to use our model as a binary classifier, we'll use the `binary_crossentropy` loss function, `adam` optimizer, and `accuracy` metrics, which are the same parameters we used for our basic neural network. To compile the model, add and run the following code:

```
# Compile the model
nn.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"
```

19.4.5 Real Data, Real Practice— Train and Evaluate the Model

Now that Beks has her model defined and compiled, it's time to train the model.

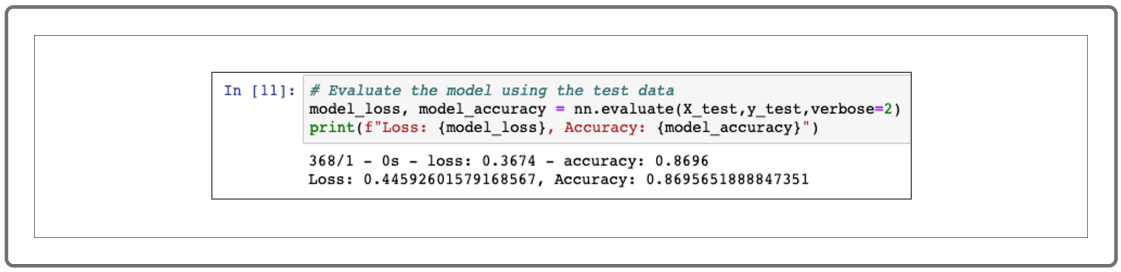
Training and evaluating the deep learning model is no different from a basic neural network. Depending on the complexity of the dataset, we may opt to increase the number of epochs to allow for the deep learning model more opportunities to optimize the weight coefficients. To train our model, we must add and run the following code:

```
# Train the model
fit_model = nn.fit(X_train,y_train,epochs=100)
```

Lastly, now that our deep learning model is properly trained, we can evaluate the model's performance by testing its predictive capabilities on our testing dataset. Add and run the following code to your notebook:

```
# Evaluate the model using the test data
model_loss, model_accuracy = nn.evaluate(X_test,y_test,verbose=2)
```

```
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```



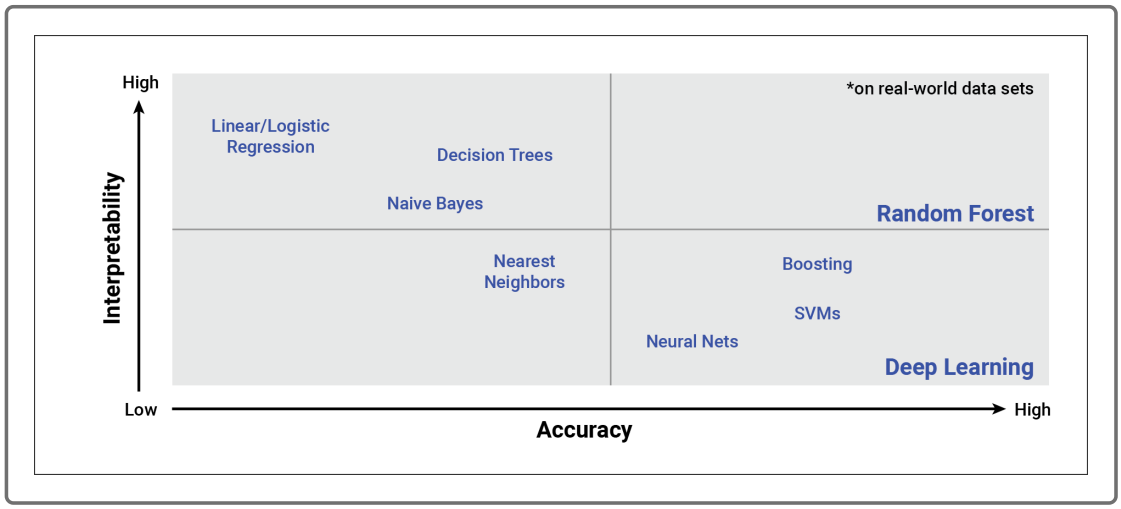
Looking at our deep learning model's performance metrics, the model was able to correctly identify employees who are at risk of attrition approximately 87% of the time. Considering that our input data included more than 30 different variables with more than 1,400 data points, the deep learning model was able to produce a fairly reliable classifier.

SKILL DRILL

Go ahead and test some different neural network designs. You can even test some different machine learning models. How does the performance increase or decrease using different models on the same input data?

19.5.1 Whose Model Is It Anyway?

Andy trusts Beks to manage so much of the foundation's data and decision-making because he knows she is thinking critically not just about how a particular model works, but also about if it's the best model for the dataset. She knows there are pros and cons to every model. So, now that she's comfortable with neural networks, it's time for her to figure out how they compare with other models.



Now that we are familiar with the structure and relative performance of a basic neural network and deep learning models, it is time to learn when and where to use these models. Contrary to what you may believe, neural networks are not the ultimate solution to all data science problems. As shown in the figure above, there are trade-offs to using the new and popular neural network (and deep learning) models over their older, often more lightweight statistics and machine learning counterparts. In this section, we'll discuss three popular regression/classification algorithms and compare them to a neural network alternative.

19.5.2 Logistic Regression Vs. a Basic Neural Network

For her first comparison, Beks will look at logistic regression and the basic neural network.

A logistic regression model is a classification algorithm that can analyze continuous and categorical variables. Using a combination of input variables, logistic regression predicts the probability of the input data belonging to one of two groups. If the probability is above a predetermined cutoff, the sample is assigned to the first group, otherwise it is assigned to the second. For example, using an applicant's personal information (such as age and income), logistic regression could be used by a bank to determine if a person *does* or *does not* qualify for a credit card.

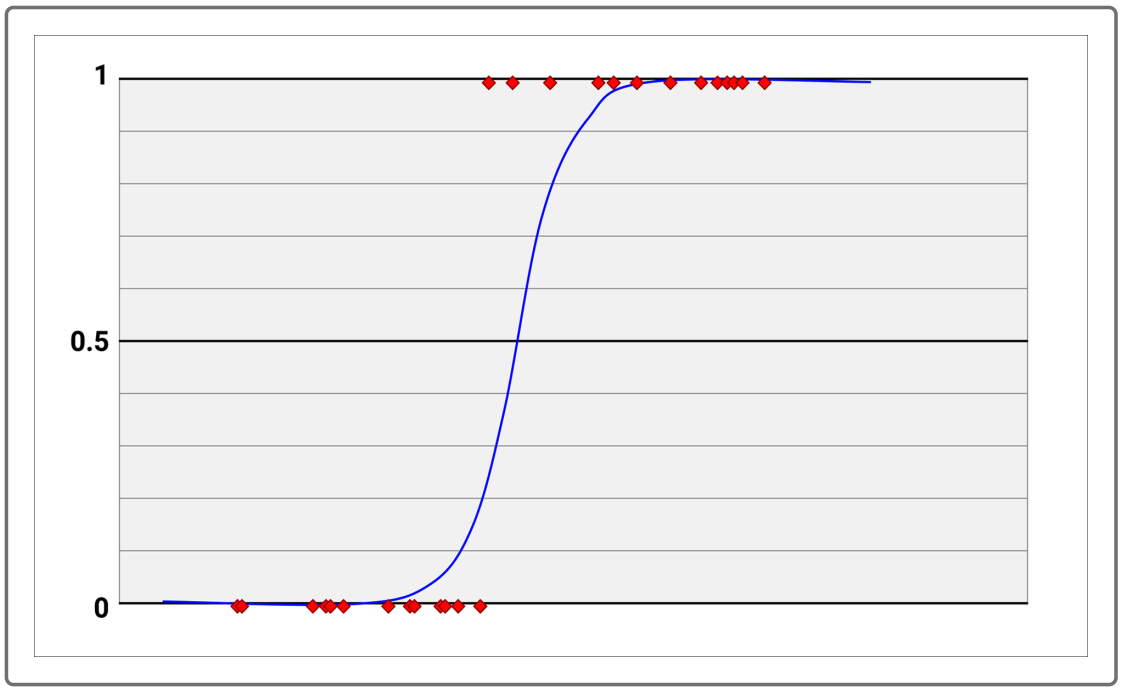
What is the definition of logistic regression?

- A statistical model that predicts the value of a dependent variable using independent variables
- A statistical model that mathematically determines its probability of belonging to one of two groups

Check Answer

Finish ▸

At the heart of the logistic regression model is the sigmoid curve, which is used to produce the probability (between 0 and 1) of the input data belonging to the first group. This sigmoid curve is the *exact* same curve used in the sigmoid activation function of a neural network. In fact, a basic neural network using the sigmoid activation function *is effectively* a logistic regression model:



To demonstrate how similar the logistic regression and basic neural network models are in terms of performance, we'll build and evaluate both models using the same training/testing dataset. First, we'll start by

downloading this **diabetes dataset (diabetes.csv)** (https://courses.bootcampspot.com/courses/138/files/28473/download?wrap=1) and placing it in a folder with a new Jupyter Notebook. Next, we'll make a new Jupyter Notebook and name it "LogisticRegression_NeuralNet" (or something similar) to help us easily locate the comparison example at another time. Once we have created our notebook and placed the dataset into the corresponding folder, we'll start by importing our libraries and reading in the dataset. Copy and run the following code into the notebook:

```
# Import our dependencies
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import pandas as pd
import tensorflow as tf

# Import our input dataset
diabetes_df = pd.read_csv('diabetes.csv')
diabetes_df.head()
```

Out[1]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

This dataset is from the National Institute of Diabetes and Digestive and Kidney Diseases (NIDDK) and contains the patient information of 786 women. It is used as a real-world practice dataset to build a predictive diagnostic model. Since there are only 786 data points across eight

features, this dataset is well suited for a logistic regression model, but still large enough to build a neural network model. Now that we have our dataset loaded into Pandas, we need to prepare the data to train both models. With our logistic regression model, there is no preprocessing or scaling required for the data. However, our basic neural network needs the numerical variables standardized. Therefore, we'll need to keep track of a scaled and unscaled training dataset such that both models have the correct input data in their preferred formats. To split the data, we need to add and run the following code:

```
# Remove diabetes outcome target from features data
y = diabetes_df.Outcome
X = diabetes_df.drop(columns="Outcome")

# Split training/test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, s
```

Next, we need to standardize the numerical variables using Scikit-learn's `StandardScaler` class. To standardize our data, we need to add and run the following code:

```
# Preprocess numerical data for neural network

# Create a StandardScaler instance
scaler = StandardScaler()

# Fit the StandardScaler
X_scaler = scaler.fit(X_train)

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

Now we're ready to train and evaluate our models. We'll first start with our logistic regression model.

REWIND

Logistic regression models can be built using Scikit-learn's `LogisticRegression` class in the `linear_model` module.

For our purposes, we'll use our basic logistic regression parameters, which include:

- The `solver` parameter is set to `'lbfgs'`, which is an algorithm for learning and optimization. The particular solver isn't very important in this example, but note that a number of optimizers exist.
- The `max_iter` parameter will be set to 200 iterations, which will give the model sufficient opportunity to converge on effective weights

Putting all of our arguments together, we'll add and run the following code in the notebook:

```
# Define the logistic regression model
log_classifier = LogisticRegression(solver="lbfgs",max_iter=200)

# Train the model
log_classifier.fit(X_train,y_train)

# Evaluate the model
y_pred = log_classifier.predict(X_test)
print(f" Logistic regression model accuracy: {accuracy_score(y_test,y_pred):
```

Next, we need to build, compile, and evaluate our basic neural network model. Again, we'll use our typical binary classifier parameters:

- Our single hidden layer will have an `input_dim` equal to 8, 16 neuron `units`, and will use the `relu` activation function.

- The loss function should be `binary_crossentropy`, using the `adam` optimizer.
- Our model should provide the additional `accuracy` scoring metric and train over a maximum of 50 epochs.

NOTE

> Compared to the 200 training iterations for our logistic regression model, we'll only train our neural network model through 50 epochs—this will limit the risk of overfitting our model.

Again, we need to add and run the following code in our notebooks:

```
# Define the basic neural network model
nn_model = tf.keras.models.Sequential()
nn_model.add(tf.keras.layers.Dense(units=16, activation="relu", input_dim=8)
nn_model.add(tf.keras.layers.Dense(units=1, activation="sigmoid"))

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

# Train the model
fit_model = nn_model.fit(X_train_scaled, y_train, epochs=100)

# Evaluate the model using the test data
model_loss, model_accuracy = nn_model.evaluate(X_test_scaled,y_test,verbose=
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

If we compare both model's predictive accuracy, their output is very similar. Either model was able to predict whether or not a patient is diagnosed with diabetes more than 70% of the time.

What are possible reasons that both models could not achieve 80% predictive accuracy? Select all that apply.

- [ ] The input data was insufficient—there were not enough data points and too few features.
- [ ] Both models are lacking optimization (parameters, structure, and weights).
- [ ] There are features in the input data that confuse the model.

Check Answer

Finish ▸

Beyond the performance of both models, there are a few other factors to consider when selecting a model for your data. First, neural networks are prone to overfitting and can be more difficult to train than a straightforward logistic regression model. Therefore, if you are trying to build a classifier with limited data points (typically fewer than a thousand data points), or if your dataset has only a few features, neural networks may be overcomplicated. Additionally, logistic regression models are easier to dissect and interpret than their neural network counterparts, which tends to put more traditional data scientists and non-data experts at ease. In contrast, neural networks (and especially deep neural networks) thrive in large datasets. Datasets with thousands of data points, or datasets with complex features, may overwhelm the logistic regression model, while a deep learning model can evaluate every interaction within and across neurons. Therefore, the decision between using a logistic regression model and basic neural network model is nuanced and, in most cases, a matter of preference for the data scientist.

19.5.3 Support Vector Machine Vs. Deep Learning Model

How might the deep learning model compare to support vector machines (SVMs)? Time for Beks to find out.

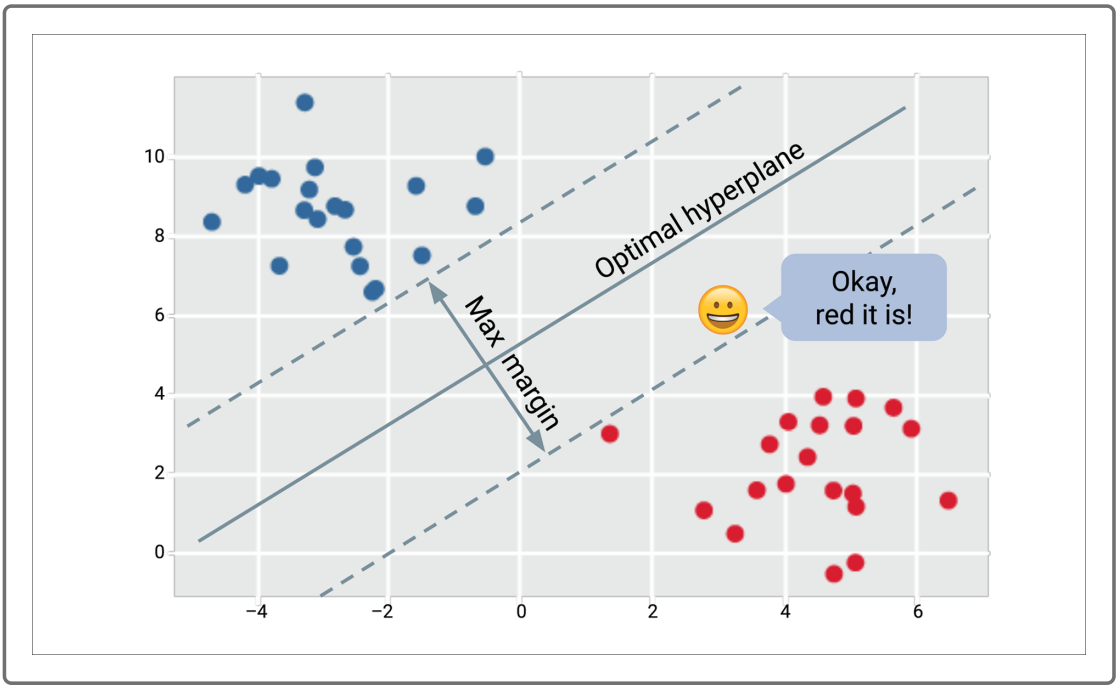
What is the definition of SVMs?

- SVMs are unsupervised learning models that analyze data used for regression and classification.
- SVMs are supervised learning models that analyze data used for regression and classification.

Check Answer

Finish ▸

SVMs are a type of binary classifier that use geometric boundaries to distinguish data points from two separate groups. More specifically, SVMs try to calculate a geometric hyperplane that maximizes the distance between the closest data point of both groups:



Unlike logistic regression, which excels in classifying data that is linearly separable but fails in nonlinear relationships, SVMs can build adequate models with linear or nonlinear data. Due to SVMs' ability to create multidimensional borders, SVMs lose their interpretability and behave more like the black box machine learning models, such as basic neural networks and deep learning models.

SVMs, like neural networks, can analyze and interpret multiple data types, such as images, natural language voice and text, or tabular data. SVMs perform one task and one task very well—they classify and create regression using two groups. In contrast, neural networks and deep learning models are capable of producing many outputs, which means neural network models can be used to classify multiple groups within the same model. Over the years, techniques have been developed to create multiple SVM models side-by-side for multiple classification problems, such as creating multiple SVM kernels. However, a single SVM is not capable of the same outputs as a single neural network.

If we only compare binary classification problems, SVMs have an advantage over neural network and deep learning models:

- Neural networks and deep learning models will often converge on a local minima. In other words, these models will often focus on a specific trend in the data and could miss the "bigger picture."
- SVMs are less prone to overfitting because they are trying to maximize the distance, rather than encompass all data within a boundary.

Despite these advantages, SVMs are limited in their potential and can still miss critical features and high-dimensionality relationships that a well-trained deep learning model could find. However, in many straightforward binary classification problems, SVMs will outperform the basic neural network, and even deep learning models with ease.

To compare and contrast the performance of an SVM versus deep learning model, we'll try to build a binary classifier using the same input data. This adapted **real-world dataset (https://www.kaggle.com/raosuny/success-of-bank-telemarketing-data)** contains bank telemarketing metrics that can be used to predict whether or not a customer is likely to subscribe to a banking service after being targeted by telemarketing advertisements. From this dataset, we want to build a binary classifier using an SVM and deep learning model and compare the predictive accuracy of either model.

First, we'll download the **bank telemarketing dataset (bank_telemarketing.csv) (https://courses.bootcampspot.com/courses/138/files/28468/download?wrap=1)**  **(https://courses.bootcampspot.com/courses/138/files/28468/download?wrap=1)** and place it in a folder with a new Jupyter Notebook. Next, we'll make a new Jupyter Notebook and name it "SVM_DeepLearning" (or something similar)—this will help us easily locate the comparison example at another time. Once we have created our notebook and placed the dataset into the corresponding folder, we'll start by importing our libraries and reading in the dataset.

Copy and run the following code into the notebook:

```
# Import our dependencies
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler,OneHotEncoder
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC
import pandas as pd
import tensorflow as tf

# Import our input dataset
tele_df = pd.read_csv('bank_telemarketing.csv')
tele_df.head()
```

Out[1]:

	Age	Job	Marital_Status	Education	Default_Credit	Housing_Loan
0	56	other	married	Primary_Education	no	no
1	37	services	married	Secondary_Education	no	yes
2	40	admin	married	Primary_Education	no	no
3	56	services	married	Secondary_Education	no	no
4	59	admin	married	Professional_Education	no	no

Unlike neural networks and deep learning models, SVMs can handle unprocessed and processed tabular data. Regardless, we'll preprocesses the dataset and use the preprocessed data for training both models—this ensures a fair comparison. For our first preprocessing workflow, let's encode our categorical variables using Scikit-Learn's `OneHotEncoder` class.

First, we must make sure that none of our categorical variables require bucketing. To check this, let's get the column names of categorical variables and check their number of unique values. Add and run the following code to the notebook:

```
# Generate our categorical variable list
tele_cat = tele_df.dtypes[tele_df.dtypes == "object"].index.tolist()
```

```
# Check the number of unique values in each column
tele_df[tele_cat].nunique()
```

```
Out[2]: Job               9
        Marital_Status    3
        Education         4
        Default_Credit    2
        Housing_Loan      2
        Personal_Loan     2
        Subscribed        2
        dtype: int64
```

Looking at the number of unique values for each categorical variable, there were no categories that require bucketing prior to encoding. Therefore, we're ready to encode by adding and running the following code to our notebooks:

```
# Create a OneHotEncoder instance
enc = OneHotEncoder(sparse=False)

# Fit and transform the OneHotEncoder using the categorical variable list
encode_df = pd.DataFrame(enc.fit_transform(tele_df[tele_cat]))

# Add the encoded variable names to the dataframe
encode_df.columns = enc.get_feature_names(tele_cat)
encode_df.head()
```

Out[3]:

	Job_admin	Job_blue-collar	Job_entrepreneur	Job_management	Job_other
0	0.0	0.0	0.0	0.0	1.0
1	0.0	0.0	0.0	0.0	0.0
2	1.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	1.0	0.0	0.0	0.0	0.0

5 rows × 24 columns

Once we have our encoded categorical variables, we need to merge our encoded columns back into our original DataFrame (as well as remove the unencoded columns). To replace the unencoded categorical variables with the encoded variables, add and run the following code to the notebook:

```
# Merge one-hot encoded features and drop the originals
tele_df = tele_df.merge(encode_df,left_index=True, right_index=True)
tele_df = tele_df.drop(tele_cat,1)
tele_df.head()
```

Out[4]:

	Age	Job_admin	Job_blue-collar	Job_entrepreneur	Job_management
0	56	0.0	0.0	0.0	0.0
1	37	0.0	0.0	0.0	0.0
2	40	1.0	0.0	0.0	0.0
3	56	0.0	0.0	0.0	0.0
4	59	1.0	0.0	0.0	0.0

5 rows × 25 columns

Now, we must split our data into the training and testing sets prior to standardization to not incorporate the testing values into the scale—testing values are only for evaluation. To perform the training/test split and standardize our numerical variables, add and run the following code in the notebook:

```
# Remove loan status target from features data
y = tele_df.Subscribed_yes.values
X = tele_df.drop(columns=["Subscribed_no","Subscribed_yes"]).values

# Split training/test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, s

# Create a StandardScaler instance
```

```
scaler = StandardScaler()

# Fit the StandardScaler
X_scaler = scaler.fit(X_train)

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

After standardizing variables in both the training and testing data, our dataset is ready for both models. First, we'll train and evaluate our SVM.

REWIND

SVMs can be built using Scikit-learn's `SVC` class in the `svm` module.

For our purposes, we'll use the SVM's `linear` kernel to try and create a linear boundary between the "Subscribed_yes" versus "Subscribed_no" groups. To create our SVM model and test the performance, add and run the following code:

```
# Create the SVM model
svm = SVC(kernel='linear')
```

```
# Train the model
svm.fit(X_train, y_train)
```

```
# Evaluate the model
y_pred = svm.predict(X_test_scaled)
```

```
print(f" SVM model accuracy: {accuracy_score(y_test,y_pred):.3f}")
```

Looking at the output of our SVM model, the model was able to correctly predict the customers who subscribed roughly 87% of the time, which is a respectable first-pass model. Next, we need to compile and evaluate our deep learning model. Again, we'll use our typical binary classifier parameters:

- Our first hidden layer will have an `input_dim` equal to the length of the scaled feature data X , 10 neuron `units`, and will use the `relu` activation function.
- Our second hidden layer will have 5 neuron `units` and also will use the `relu` activation function.
- The loss function should be `binary_crossentropy`, using the `adam` optimizer.

NOTE

> Unlike our basic neural network model, we don't want to use two to three times the number of neurons as input variables—we don't want our deeper layers to overfit the input data.

To build and compile our deep learning model, we must add and run the following code:

```
# Define the model - deep neural net
number_input_features = len(X_train_scaled[0])
hidden_nodes_layer1 =  10
hidden_nodes_layer2 = 5

nn = tf.keras.models.Sequential()

# First hidden layer
nn.add(
    tf.keras.layers.Dense(units=hidden_nodes_layer1, input_dim=number_input_
```

```
)

# Second hidden layer
nn.add(tf.keras.layers.Dense(units=hidden_nodes_layer2, activation="relu"))


# Output layer
nn.add(tf.keras.layers.Dense(units=1, activation="sigmoid"))

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

Lastly, we need to train and evaluate our deep learning model. Because this dataset contains fewer features than other datasets we have used previously, we only need to train over a maximum of 50 epochs. Again, we must add and run the following code:

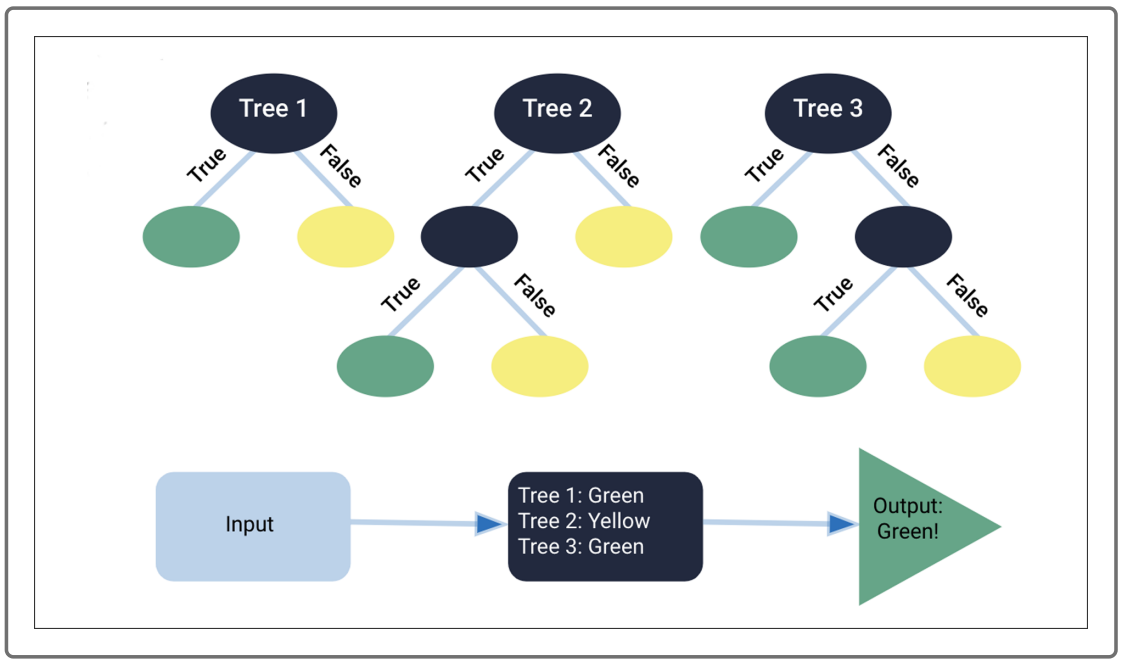
```
# Train the model
fit_model = nn.fit(X_train_scaled, y_train, epochs=50)
# Evaluate the model using the test data
model_loss, model_accuracy =
nn.evaluate(X_test_scaled,y_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

Looking at the results of our comparative analysis, the SVM and deep learning models both achieved a predictive accuracy around 87%. Additionally, both models take similar amounts of time to train on the input data. The only noticeable difference between the two models is implementation—the amount of code required to build and train the SVM is notably less than the comparable deep learning model. As a result, many data scientists will prefer to use SVMs by default, then turn to deep learning models, as needed.

19.5.4 Random Forest Vs. Deep Learning Model

Random forest classifiers are one of Bek's favorites. (Not only is it a powerful model type, she can't help thinking about actual forests when she sees the name!) She's curious how it stacks up against a deep learning model.

Random forest classifiers are a type of ensemble learning model that combines multiple smaller models into a more robust and accurate model. Random forest models use a number of weak learner algorithms (decision trees) and combine their output to make a final classification (or regression) decision. Structurally speaking, random forest models are very similar to their neural network counterparts. Random forest models have been a staple in machine learning algorithms for many years due to their robustness and scalability. Both output and feature selection of random forest models are easy to interpret, and they can easily handle outliers and nonlinear data.



What is the definition of random forest?

- Random forest is a supervised ensemble learning model that combines decision trees to analyze input data.
- Random forest is an unsupervised machine learning model that groups similar objects into clusters.

Check Answer

Finish ▸

SKILL DRILL

Take a moment to consider a few different reasons to the following question:

If random forest models are fairly robust and clear, why would you want to replace them with a neural network?

The answer depends on the type and complexity of the entire dataset. First and foremost, random forest models will only handle tabular data, so data such as images or natural language data cannot be used in a random

forest without heavy modifications to the data. Neural networks can handle all sorts of data types and structures in raw format or with general transformations (such as converting categorical data).

In addition, each model handles input data differently. Random forest models are dependent on each weak learner being trained on a subset of the input data. Once each weak learner is trained, the random forest model predicts the classification based on a consensus of the weak learners. In contrast, deep learning models evaluate input data within a single neuron, as well as across multiple neurons and layers.

As a result, the deep learning model might be able to identify variability in a dataset that a random forest model could miss. However, a random forest model with a sufficient number of estimators and tree depth should be able to perform at a similar capacity to most deep learning models.

To compare the implementation and performance of a random forest model versus a deep learning model, we'll train and evaluate both models on the same data. This time, we'll use a dataset that has been adapted from **bank loan data (https://www.kaggle.com/zaurbegiev/my-dataset#credit_train.csv)** with more than 36,000 rows and 16 feature columns. From this dataset, we want to build a classifier that can predict whether or not a loan will or will not be paid provided their current loan status and metrics.

First, we'll download the **bank loan status dataset (loan_status.csv) (https://courses.bootcampspot.com/courses/138/files/28490/download?wrap=1)** **(https://courses.bootcampspot.com/courses/138/files/28490/download?wrap=1)** and place it in a folder with a new Jupyter Notebook. Next, we'll make a new Jupyter Notebook and name it "RandomForest_DeepLearning" (or something similar). This will help us easily locate the comparison example at another time. Once we have created our notebook and placed the dataset into the corresponding folder, we'll start by importing our libraries and reading in the dataset. Copy and run the following code into the notebook:

```
# Import our dependencies
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import OneHotEncoder
import pandas as pd
import tensorflow as tf

# Import our input dataset
loans_df = pd.read_csv('loan_status.csv')
loans_df.head()
```

Out[1]:

	Loan_Status	Current_Loan_Amount	Term	Credit_Score	Annual_Income	Years_in_current_job	Home_Ownership
0	Fully_Paid	99999999	Short_Term	741.0	2231892.0	8_years	Own_Home
1	Fully_Paid	217646	Short_Term	730.0	1184194.0	<_1_year	Home_Mortgage
2	Fully_Paid	548746	Short_Term	678.0	2559110.0	2_years	Rent
3	Fully_Paid	99999999	Short_Term	728.0	714628.0	3_years	Rent
4	Fully_Paid	99999999	Short_Term	740.0	776188.0	<_1_year	Own_Home

Because both Scikit-Learn's `RandomForestClassifier` class and TensorFlow's `Sequential` class require preprocessing, we can perform our preprocessing steps on all of the data—no need to keep track of separate scaled and unscaled data. For our first preprocessing workflow, let's encode our categorical variables using Scikit-Learn's `OneHotEncoder` class.

First, we must make sure that none of our categorical variables require bucketing. To check this, let's get the column names of categorical variables and check their number of unique values. Add and run the following code to the notebook:

```
# Generate our categorical variable list
loans_cat = loans_df.dtypes[loans_df.dtypes == "object"].index.tolist()

# Check the number of unique values in each column
loans_df[loans_cat].nunique()
```

```
Out[2]: Loan_Status              2
        Term                     2
        Years_in_current_job    11
        Home_Ownership           4
        Purpose                  7
        dtype: int64
```

Looking at the number of unique values in our categorical variable, the “Years_in_current_job” column does have 11 unique values. Therefore, we should check the number of data points for each unique value to find out if any categorical variables can be bucketed together. Again, add and run the following code to the notebook:

```
# Check the unique value counts to see if binning is required
loans_df.Years_in_current_job.value_counts()
```

```
Out[3]: 10+_years    13149
        2_years       3225
        3_years       2997
        <_1_year      2699
        5_years       2487
        4_years       2286
        1_year        2247
        6_years       2109
        7_years       2082
        8_years       1675
        9_years       1467
        Name: Years_in_current_job, dtype: int64
```

Looking at the number of data points for each unique value, all of the categorical values have a substantial number of data points. In this case, we have reason to leave the “Years_in_current_job” column alone because we don’t want to bucket common values together and cause confusion in the model.

Since all of the categorical variables are ready for encoding, we can add and run the following code to the notebook:

```
# Create a OneHotEncoder instance
enc = OneHotEncoder(sparse=False)
```

```
# Fit and transform the OneHotEncoder using the categorical variable list
encode_df = pd.DataFrame(enc.fit_transform(loans_df[loans_cat]))

# Add the encoded variable names to the DataFrame
encode_df.columns = enc.get_feature_names(loans_cat)
encode_df.head()
```

Out[4]:

	Loan_Status_Fully_Paid	Loan_Status_Not_Paid	Term_Long_Term	Term_Short_Term
0	1.0	0.0	0.0	1.0
1	1.0	0.0	0.0	1.0
2	1.0	0.0	0.0	1.0
3	1.0	0.0	0.0	1.0
4	1.0	0.0	0.0	1.0

5 rows × 26 columns

Now that our categorical variables have been encoded, we need to merge them back into our original data frame and remove the unencoded columns. To do this, add and run the following code in the notebook:

```
# Merge one-hot encoded features and drop the originals
loans_df = loans_df.merge(encode_df,left_index=True, right_index=True)
loans_df = loans_df.drop(loans_cat,1)
loans_df.head()
```

Out[5]:

	Current_Loan_Amount	Credit_Score	Annual_Income	Monthly_Debt	Years_of_Credit_History
0	99999999	741.0	2231892.0	29200.53	14.9
1	217646	730.0	1184194.0	10855.08	19.6
2	548746	678.0	2559110.0	18660.28	22.6
3	99999999	728.0	714628.0	11851.06	16.0
4	99999999	740.0	776188.0	11578.22	8.5

5 rows × 38 columns

Next, we need to standardize our numerical variables using Scikit-Learn's `StandardScaler` class. Again, we must split our data into the training and testing sets prior to standardization to not incorporate the testing values into the scale. To perform the training/test split and standardize our numerical variables, add and run the following code in the notebook:

```
# Remove loan status target from features data
y = loans_df.Loan_Status_Fully_Paid
X = loans_df.drop(columns=["Loan_Status_Fully_Paid","Loan_Status_Not_Paid"])

# Split training/test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, s

# Create a StandardScaler instance
scaler = StandardScaler()

# Fit the StandardScaler
X_scaler = scaler.fit(X_train)

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

After standardizing variables in both the training and testing data, our dataset is ready for both models. First, we'll train and evaluate our random forest classifier.

REWIND

Random forest models can be built using Scikit-learn's `RandomForestClassifier` class in the `ensemble` module.

For our purposes, we'll use a random forest classifier with the `n_estimators` parameter set to 128. Typically, 128 estimators is the largest number of estimators we would want to use in a model. To create our random forest classifier model and test the performance, add and run the following code:

```
# Create a random forest classifier.
rf_model = RandomForestClassifier(n_estimators=128, random_state=78)

# Fitting the model
rf_model = rf_model.fit(X_train_scaled, y_train)

# Evaluate the model
y_pred = rf_model.predict(X_test_scaled)
print(f" Random forest predictive accuracy: {accuracy_score(y_test,y_pred):.
```

What is the predictive accuracy of your model? Round to the nearest hundredth.

- 0.75
- 0.8
- 0.85

Check Answer

Finish ▸

Next, we need to build, compile, and evaluate our deep learning model. Again, we'll use our typical binary classifier parameters:

- Our first hidden layer will have an `input_dim` equal to 38, 24 neuron `units`, and will use the `relu` activation function.
- Our second hidden layer will have 12 neuron `units` and also will use the `relu` activation function.

- The loss function should be `binary_crossentropy`, using the `adam` optimizer.

Our model should provide the additional `accuracy` scoring metric and train over a maximum of 50 epochs.

To build and evaluate our deep learning model, we must add and run the following code to the notebook:

```
# Define the model - deep neural net
number_input_features = len(X_train_scaled[0])
hidden_nodes_layer1 =  24
hidden_nodes_layer2 = 12

nn = tf.keras.models.Sequential()

# First hidden layer
nn.add(
    tf.keras.layers.Dense(units=hidden_nodes_layer1, input_dim=number_input_
)

# Second hidden layer
nn.add(tf.keras.layers.Dense(units=hidden_nodes_layer2, activation="relu"))


# Output layer
nn.add(tf.keras.layers.Dense(units=1, activation="sigmoid"))

# Compile the Sequential model together and customize metrics
nn.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"

# Train the model
fit_model = nn.fit(X_train_scaled, y_train, epochs=50)

# Evaluate the model using the test data
model_loss, model_accuracy = nn.evaluate(X_test_scaled,y_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

```
27317/27317 [==============================] - 1s 51us/sample - loss: 0.3610 - accuracy: 0.8533s
ccuracy:
Epoch 44/50
27317/27317 [==============================] - 1s 53us/sample - loss: 0.3605 - accuracy: 0.8538s
Epoch 45/50
27317/27317 [==============================] - 1s 51us/sample - loss: 0.3607 - accuracy: 0.8536
Epoch 46/50
27317/27317 [==============================] - 1s 50us/sample - loss: 0.3601 - accuracy: 0.8539
Epoch 47/50
27317/27317 [==============================] - 1s 51us/sample - loss: 0.3600 - accuracy: 0.8536
Epoch 48/50
27317/27317 [==============================] - 1s 51us/sample - loss: 0.3600 - accuracy: 0.8543s
ccuracy - ETA: 0s - los
Epoch 49/50
27317/27317 [==============================] - 1s 51us/sample - loss: 0.3591 - accuracy: 0.8537
Epoch 50/50
27317/27317 [==============================] - 1s 50us/sample - loss: 0.3591 - accuracy: 0.8536
9106/1 - 0s - loss: 0.4502 - accuracy: 0.8452
Loss: 0.3937950540630995, Accuracy: 0.8451570272445679
```

Again, if we compare both model's predictive accuracy, their output is very similar. Both the random forest and deep learning models were able to predict correctly whether or not a loan will be repaid over 80% of the time. Although their predictive performance was comparable, their implementation and training times were not—the random forest classifier was able to train on the large dataset and predict values in seconds, while the deep learning model required a couple minutes to train on the tens of thousands of data points. In other words, the random forest model is able to achieve comparable predictive accuracy on large tabular data with less code and faster performance. The ultimate decision of whether to use a random forest versus a neural network comes down to preference. However, if your dataset is tabular, random forest is a great place to start.

19.6.1 Checkpoints Are Not Just for Video Games

During their weekly check-in, Beks catches up Andy on all the work she's been doing exploring neural networks. Andy is quite impressed. He lets Beks know that if she needs more resources on her team, such as an analyst, to just let him know and they'll put together a budget.

Beks is delighted—she would love to hire a new bootcamp grad, just like herself! But first, how might she make sure she can actually share her models with her potential new teammate?

Neural networks, especially complex neural networks, are resource-hungary algorithms. When it comes to training neural networks on medium to large datasets, the amount of computation time to adequately train a model can take hours (or even days!) With simple modelling problems, like the ones covered in this module, training a model in the same notebook as an analysis is no problem. However, with more formal applications of neural network and deep learning models, data scientists cannot afford the time or resources to build and train a model each time they analyze data. In these cases, a trained model must be stored and accessed outside of the training environment.

With TensorFlow, we have the ability to save and load neural network models at any stage, including partially trained models. When building a TensorFlow model, if we use Keras' `ModelCheckpoint` method, we can save the model weights after it tests a set number of data points. Then, at any point, we can reload the checkpoint weights and resume model training. Saving checkpoints while training has a number of benefits:

- We can short-circuit our training loop at any time (stop the function by pressing CTRL+C, or by pressing the stop button at the top of the notebook). This can be helpful if the model is showing signs of overfitting.
- The model is protected from computer problems (power failure, computer crash, etc.). Worst-case scenario: We would lose five epochs' worth of optimization.
- We can restore previous model weight coefficients to try and revert overfitting.

Let's practice generating checkpoint files and loading model weights from different epochs. To make things simple, we'll implement checkpoints to our previous deep learning example notebook. To start, we'll open our "DeepLearning_Tabular" (or whatever similar name you may have used) notebook and rerun all of the code in the notebook for the following steps:

1. Import dependencies.
2. Import the input dataset.
3. Generate categorical variable list.
4. Create a `OneHotEncoder` instance.
5. Fit and transform the `OneHotEncoder`.
6. Add the encoded variable names to the DataFrame.
7. Merge one-hot encoded features and drop the originals.
8. Split the preprocessed data into features and target arrays.
9. Split the preprocessed data into training and testing dataset.
10. Create a `StandardScaler` instance.

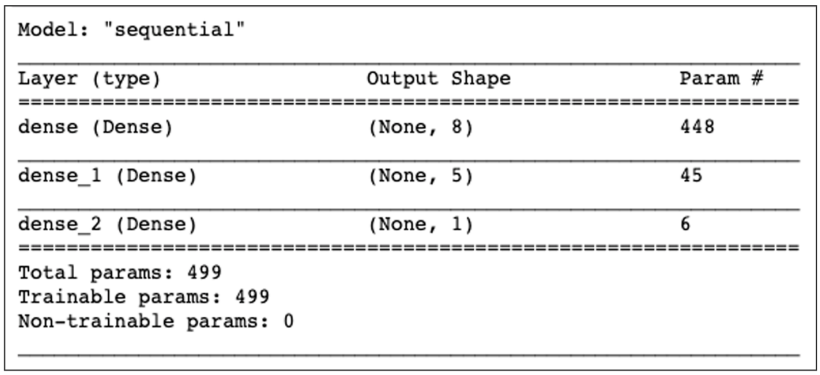
11. Fit the `StandardScaler`.
12. Scale the data.
13. Define the model.
14. Add first and second hidden layers.
15. Add the output layer.
16. Check the structure of the model.

REWIND

You have coded all of these steps within this module. If you get stuck, or something is not working, try going back to earlier sections and recopy the code blocks.

Now that we have our training data ready, we can implement checkpoints to our deep learning model.

Rerun all of the code in the “DeepLearning_Tabular” to prepare your input data and define your deep learning model. Does your output of `nn.summary()` look like the following image?



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 8)                 448
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 45
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 6
=================================================================
Total params: 499
Trainable params: 499
Non-trainable params: 0
_________________________________________________________________
```

- ○ Yes, everything looks the same.
- ○ No, my code works, but the number of parameters are different.
- ○ My code is throwing errors, and I can’t get to that step.

Check Answer

Finish ▸

Now that we have our training data and our model defined, we’re ready to compile and train our model using checkpoints. To use checkpoints, we need to define the checkpoint file name and directory path. For our purposes, we’ll label our checkpoints by epoch number and contain them within their own folder. This ensures that our checkpoint files are neat, organized, and easily identifiable. Add and run the following code to our notebook:

```
# Import checkpoint dependencies
import os
from tensorflow.keras.callbacks import ModelCheckpoint

# Define the checkpoint path and filenames
os.makedirs("checkpoints/",exist_ok=True)
checkpoint_path = "checkpoints/weights.{epoch:02d}.hdf5"
```

Once we have defined the file structure and filepath, we need to create a **callback** object for our deep learning model. A callback object is used in the Keras module to define a set of functions that will be applied at specific stages of the training process.There are a number of different callback functions available that can create log files, force training to stop, send training status messages, or in our case save model checkpoints. To create an effective checkpoint callback using the `ModelCheckpoint` method, we need to provide the following parameters:

- `filepath=checkpoint_path`—the checkpoint directory and file structure we defined previously
- `verbose=1`—we'll be notified when a checkpoint is being saved to the directory
- `save_weights_only=True`—saving the full model each time can fill up a hard drive very quickly; this ensures that the checkpoint files take up minimal space
- `save_freq=1000`—checkpoints will be saved every thousand samples tested (across all epochs)

Bringing it all together, we can compile, train, and evaluate our deep learning model by adding and running the following code:

```
# Compile the model
nn.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"

# Create a callback that saves the model's weights every 5 epochs
cp_callback = ModelCheckpoint(
    filepath=checkpoint_path,
    verbose=1,
    save_weights_only=True,
    save_freq=1000)

# Train the model
fit_model = nn.fit(X_train_scaled,y_train,epochs=100,callbacks=[cp_callback]

# Evaluate the model using the test data
model_loss, model_accuracy = nn.evaluate(X_test_scaled,y_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

```
Epoch 97/100
 800/1102 [====================>.........] - ETA: 0s - loss: 0.1354 - accuracy: 0.9575
Epoch 00097: saving model to checkpoints/weights.97.hdf5
1102/1102 [==============================] - 0s 75us/sample - loss: 0.1347 - accuracy: 0.9564
Epoch 98/100
 800/1102 [====================>.........] - ETA: 0s - loss: 0.1258 - accuracy: 0.9613
Epoch 00098: saving model to checkpoints/weights.98.hdf5
1102/1102 [==============================] - 0s 77us/sample - loss: 0.1338 - accuracy: 0.9546
Epoch 99/100
 672/1102 [=================>............] - ETA: 0s - loss: 0.1219 - accuracy: 0.9628
Epoch 00099: saving model to checkpoints/weights.99.hdf5
1102/1102 [==============================] - 0s 84us/sample - loss: 0.1328 - accuracy: 0.9564
Epoch 100/100
  32/1102 [..............................] - ETA: 0s - loss: 0.0944 - accuracy: 0.9688
Epoch 00100: saving model to checkpoints/weights.100.hdf5
1102/1102 [==============================] - 0s 64us/sample - loss: 0.1319 - accuracy: 0.9574
368/1 - 0s - loss: 0.3284 - accuracy: 0.8560
Loss: 0.5140212197666583, Accuracy: 0.85597825050354
```

After running the previous code, we have created our trained model within the Python session, as well as a folder of checkpoints we can use to restore previous model weights. Now if we ever need to restore weights, we can use the Keras Sequential model's `load_weights` method to restore the model weights. To test this functionality, let's define another deep learning model, but restore the weights using the checkpoints rather than training the model. Once again we must add and run the following to our notebooks:

```
# Define the model - deep neural net
number_input_features = len(X_train[0])
hidden_nodes_layer1 =  8
hidden_nodes_layer2 = 5

nn_new = tf.keras.models.Sequential()

# First hidden layer
nn_new.add(
    tf.keras.layers.Dense(units=hidden_nodes_layer1, input_dim=number_input_
)

# Second hidden layer
nn_new.add(tf.keras.layers.Dense(units=hidden_nodes_layer2, activation="relu

# Output layer
nn_new.add(tf.keras.layers.Dense(units=1, activation="sigmoid"))
```

```
# Compile the model
nn_new.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accur

# Restore the model weights
nn_new.load_weights("checkpoints/weights.100.hdf5")

# Evaluate the model using the test data
model_loss, model_accuracy = nn_new.evaluate(X_test_scaled,y_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

```
# Define the model - deep neural net
number_input_features = len(X_train[0])
hidden_nodes_layer1 =  8
hidden_nodes_layer2 = 5

nn_new = tf.keras.models.Sequential()

# First hidden layer
nn_new.add(
    tf.keras.layers.Dense(units=hidden_nodes_layer1, input_dim=number_input_features, activation="relu")
)

# Second hidden layer
nn_new.add(tf.keras.layers.Dense(units=hidden_nodes_layer2, activation="relu"))

# Output layer
nn_new.add(tf.keras.layers.Dense(units=1, activation="sigmoid"))

# Compile the model
nn_new.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])

# Restore the model weights
nn_new.load_weights("checkpoints/weights.100.hdf5")

# Evaluate the model using the test data
model_loss, model_accuracy = nn_new.evaluate(X_test_scaled,y_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

```
368/1 - 0s - loss: 0.3321 - accuracy: 0.8560
Loss: 0.5171250726865686, Accuracy: 0.85597825050354
```

Using the checkpoints, we were able to regenerate the model instantaneously and confirm the model was able to produce the exact same results.

19.6.2 For Best Results, Please Save After Training

Saving the weights is great, but what about saving—and sharing—the entire model? Is that possible? Beks is determined to find out.

Checkpoints are a great way to save model weights during training, but they fall short when it comes to sharing a trained model. In data science, trained models are published in scientific papers, deployed in software, open-sourced on GitHub, not to mention passed along to colleagues. In these cases, it is not practical to pass along only model weights, which can cause frustration and confusion. Instead, we can use the Keras Sequential model's `save` method to export the entire model (weights, structure, and configuration settings) to an Hierarchical Data Format (**HDF5** **(https://en.wikipedia.org/wiki/Hierarchical_Data_Format)**) file. Once saved, anyone can import the *exact* same trained model to their environment by using the Keras `load_model` method and use it for analysis.

NOTE

> Even though we can save full neural network and deep learning models using Keras checkpoints, each full model file is almost ten times the

> size of a weight-only file. For those with limited hard drive space, saving full models using checkpoints is not feasible.

To practice exporting and importing our entire model, we'll use the same notebook as our previous section.

NOTE

> If you closed your "DeepLearning_Tabular" notebook from the previous section, open it and run all of the code in the notebook to create a trained model.

Currently, in our notebook environment, we should have a fully trained classification model that can predict employee attrition based on features in the dataset. To export the trained model, we need to add and run the following code:

```
# Export our model to HDF5 file
nn_new.save("trained_attrition.h5")
```

After running the code, we should see a file named "trained_attrition.h5," which contains the complete model and configuration. Now that we have the model saved, we can create the model at any point. Let's try importing the model into the notebook without providing any structure or context. To import the model, add and run the following code:

```
# Import the model to a new object
nn_imported = tf.keras.models.load_model('trained_attrition.h5')
```

Lastly, we can test the performance of the model on our test dataset by adding and running the following code:

```
# Evaluate the model using the test data
model_loss, model_accuracy = nn_new.evaluate(X_test_scaled,y_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

```
368/1 - 0s - loss: 0.3321 - accuracy: 0.8560
Loss: 0.5171250726865686, Accuracy: 0.85597825050354
```

Looking at the final results, our imported model was able to reproduce the exact same performance metrics as the original model. Using this same procedure, we can import any type of Keras model for evaluation on a dataset with the same features.

Module 19 Challenge

Submit Assignment

Due Sunday by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> Beks is finally ready to put her skills to work to help the foundation predict where to make investments. She's come a long way since her first day at that boot camp five years ago—and since her first day at Alphabet Soup—and since earlier this week, when she just started learning about neural networks!
>
> Who knows how much farther she will go?

In this challenge, you'll have to build your own machine learning model that will be able to predict the success of a venture paid by Alphabet soup. Your trained model will be used to determine the future decisions of the company—only those projects likely to be a success will receive any future funding from Alphabet Soup.

Background

From Alphabet Soup's business team, Beks received a CSV containing more than 34,000 organizations that have received various amounts of funding from Alphabet Soup over the years. Within this dataset are a number of columns that capture metadata about each organization such as the following:

- **EIN** and **NAME**—Identification columns
- **APPLICATION_TYPE**—Alphabet Soup application type
- **AFFILIATION**—Affiliated sector of industry
- **CLASSIFICATION**—Government organization classification
- **USE_CASE**—Use case for funding
- **ORGANIZATION**—Organization type
- **STATUS**—Active status
- **INCOME_AMT**—Income classification
- **SPECIAL_CONSIDERATIONS**—Special consideration for application
- **ASK_AMT**—Funding amount requested
- **IS_SUCCESSFUL**—Was the money used effectively

Using your knowledge of machine learning and neural network model building, you must create a binary classifier that is capable of predicting whether or not an applicant will be successful if funded by Alphabet Soup using the features collected in the provided dataset.

Objectives

The goals of this challenge are for you to:

- Import, analyze, clean, and preprocess a “real-world” classification dataset.
- Select, design, and train a binary classification model of your choosing.
- Optimize model training and input data to achieve desired model performance.

Instructions

1. Create a new Jupyter Notebook within a new folder on your computer. Name this new notebook file “AlphabetSoupChallenge.ipynb” (or something easily identifiable).
2. Download the **Alphabet Soup Charity dataset (charity_data.csv)** and place it in the same directory as your notebook.
3. Import and characterize the input data. **Hint:** Be sure to identify the following in your dataset:
    - What variable(s) are considered the target for your model?
    - What variable(s) are considered to be the features for your model?
    - What variable(s) are neither and should be removed from the input data?
4. Using the methods described in this module, preprocess all numerical and categorical variables, as needed:
    - Combine rare categorical values via bucketing.
    - Encode categorical variables using one-hot encoding.
    - Standardize numerical variables using Scikit-Learn’s `StandardScaler` class.
5. Using a TensorFlow neural network design of your choice, create a binary classification model that can predict if an Alphabet Soup funded organization will be successful based on the features in the dataset.
    - You may choose to use a neural network or deep learning model.

- **Hint:** Think about how many inputs there are before determining the number of neurons and layers in your model.

6. Compile, train, and evaluate your binary classification model. Be sure that your notebook produces the following outputs:
    - Final model loss metric
    - Final model predictive accuracy

7. Do your best to optimize your model training and input data to achieve a target predictive accuracy **higher than 75%.**
    - Look at **Page 19.2.6** for ideas on how to optimize and boost model performance.
    - **Note:** You will not be penalized if your model does not achieve target performance, as long as you demonstrate an attempt at model optimization within your notebook.

8. Create a new README.txt file within the same folder as your AlphabetSoupChallenge.ipynb notebook. Include a 5–10 sentence writeup in your README that addresses the following questions:
    - How many neurons and layers did you select for your neural network model? Why?
    - Were you able to achieve the target model performance? What steps did you take to try and increase model performance?
    - If you were to implement a different model to solve this classification problem, which would you choose? Why?

Submission

Make sure your repo is up to date and includes the following:

- An AlphabetSoupChallenge.ipynb file containing all of your model building code
- A README.txt file containing your writeup

Submit a link to your repository through Canvas.

Rubric

Please **download the detailed rubric** to access the assessment criteria.

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Module 19 Rubric

Criteria	Ratings					Pts
Written Analysis Please see detailed rubric linked in Challenge description.	**30.0 pts Full Marks**	**23.0 pts Approaching Mastery**	**16.0 pts Progressing**	**7.0 pts Emerging**	**0.0 pts No Marks**	30.0 pts
Data Pre-Processing Please see detailed rubric linked in Challenge description.	**30.0 pts Full Marks**	**23.0 pts Approaching Mastery**	**16.0 pts Progressing**	**7.0 pts Emerging**	**0.0 pts No Marks**	30.0 pts
Compile, Train, and Evaluate Model Please see detailed rubric linked in Challenge description.	**20.0 pts Full Marks**	**15.0 pts Approaching Mastery**	**10.0 pts Progressing**	**5.0 pts Emerging**	**0.0 pts No Marks**	20.0 pts
Optimize Model Please see detailed rubric linked in Challenge description.	**20.0 pts Full Marks**	**15.0 pts Approaching Mastery**	**10.0 pts Progressing**	**5.0 pts Emerging**	**0.0 pts No Marks**	20.0 pts
						Total Points: 100.0

	Mastery	Approaching Mastery	Progressing	Emerging	Incomplete
Written Analysis (30 points)	Presents a cohesive written analysis via readme or text file that answers the following questions. ✓ How many neurons and layers did you select for your neural network model? ✓ Were you able to achieve target model performance? ✓ What steps did you take to try and increase model performance? ✓ If you were to implement a different model to solve this classification problem, which would you choose and why?	Presents a cohesive written analysis via readme or text file that answers three of the following questions. ✓ How many neurons and layers did you select for your neural network model? ✓ Were you able to achieve target model performance? ✓ What steps did you take to try and increase model performance? ✓ If you were to implement a different model to solve this classification problem, which would you choose and why?	Presents a developing written analysis via readme or text file that answers two the following questions. ✓ How many neurons and layers did you select for your neural network model? ✓ Were you able to achieve target model performance? ✓ What steps did you take to try and increase model performance? ✓ If you were to implement a different model to solve this classification problem, which would you choose and why?	Presents a limited written analysis via readme or text file that answers one the following questions. ✓ How many neurons and layers did you select for your neural network model? ✓ Were you able to achieve target model performance? ✓ What steps did you take to try and increase model performance? ✓ If you were to implement a different model to solve this classification problem, which would you choose and why?	No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty
Data Pre-Processing (30 points)	Data is appropriately pre-processed, including all of the following steps. ✓ Filters out non-feature variables from dataset if needed ✓ Applies categorical binning when appropriate ✓ Encodes categorical variables using one-hot encoding ✓ Standardizes and scales numerical variables using StandardScalar	Data is appropriately pre-processed, including two or three of the following steps. ✓ Filters out non-feature variables from dataset if needed ✓ Applies categorical binning when appropriate ✓ Encodes categorical variables using one-hot encoding ✓ Standardizes and scales numerical variables using StandardScalar	Data is appropriately pre-processed, including at least one of the following steps. ✓ Filters out non-feature variables from dataset if needed ✓ Applies categorical binning when appropriate ✓ Encodes categorical variables using one-hot encoding ✓ Standardizes and scales numerical variables using StandardScalar	Student attempts to produce working code that produces one of the following steps: ✓ Filters out non-feature variables from dataset if needed ✓ Applies categorical binning when appropriate ✓ Encodes categorical variables using one-hot encoding ✓ Standardizes and scales numerical variables using StandardScalar -OR-	

				Student reads in a preprocessed data frame into the notebook without any preprocessing code.	
Compile, Train, and Evaluate Model (20 points)	The notebook contains working code that performs the following steps. ✓ Defines a neural network model using Tensorflow Keras module ✓ Defines the number of layers and number of neurons per layer ✓ Defines a neural network model using Tensorflow Keras module ✓ Compiles model that defines loss metric and optimization function ✓ Train and evaluate model using predictive accuracy	The notebook contains working code that performs four of the following steps. ✓ Defines a neural network model using Tensorflow Keras module ✓ Defines the number of layers and number of neurons per layer ✓ Defines a neural network model using Tensorflow Keras module ✓ Compiles model that defines loss metric and optimization function ✓ Train and evaluate model using predictive accuracy	The notebook contains working code that performs at least one of the following steps. ✓ Defines a neural network model using Tensorflow Keras module ✓ Defines the number of layers and number of neurons per layer ✓ Defines a neural network model using Tensorflow Keras module ✓ Compiles model that defines loss metric and optimization function ✓ Train and evaluate model using predictive accuracy	Student attempts to produce working code that produces one of the following steps: ✓ Defines a neural network model using Tensorflow Keras module ✓ Defines the number of layers and number of neurons per layer ✓ Defines a neural network model using Tensorflow Keras module ✓ Compiles model that defines loss metric and optimization function ✓ Train and evaluate model using predictive accuracy	
Optimize Model (20 points)	Student produces model that demonstrates predictive accuracy over 75% -OR- The notebook contains working code that attempts to increase model performance at least three times using the following steps: ✓ Leaving out noisy variables from features ✓ Adds additional neurons to hidden layers ✓ Adds additional hidden layers to model ✓ Changes activation function of hidden layers or output layer	The notebook contains working code that attempts to increase model performance at least two times using the following steps ✓ Leaving out noisy variables from features ✓ Adds additional neurons to hidden layers ✓ Adds additional hidden layers to model ✓ Changes activation function of hidden layers or output layer ✓ Trains through additional epochs	The notebook contains working code that attempts to increase model performance at least one time using the following steps ✓ Leaving out noisy variables from features ✓ Adds additional neurons to hidden layers ✓ Adds additional hidden layers to model ✓ Changes activation function of hidden layers or output layer ✓ Trains through additional epochs	Student attempts to produce working code that produces one of the following steps: ✓ Leaving out noisy variables from features ✓ Adds additional neurons to hidden layers ✓ Adds additional hidden layers to model ✓ Changes activation function of hidden layers or output layer ✓ Trains through additional epochs	

	✓ Trains through additional epochs				

Module 19
Career Connection

Introduction

Welcome back to another Career Connection! This week you continued to explore machine learning and dug deeper into advanced machine learning. You discussed neural networks and got started on using the TensorFlow platform in Python.

It was a packed week! This material is difficult, but if you're interested in a career in machine learning, then everything we've covered is essential.

As we discussed last week, machine learning is an extremely popular career choice. Job postings continue to increase, salaries are high, and job security is fantastic. Deep learning engineers have opportunities across the globe. **Leading companies hiring deep learning engineers (https://www.glassdoor.com/research/ai-jobs/)** include Amazon, Nvidia, Microsoft, IBM, Accenture, Facebook, and many others.

It's not easy to define exactly what an artificial intelligence (AI) or machine learning job entails. Applying a very broad definition, we might consider any job opening at any company using or performing AI services. Alternatively, we might consider only tech roles where the employee in the role is actually building and utilizing machine learning. Using the latter definition, you can find the most common job titles for such roles by searching sites like **Indeed.com (https://www.indeed.com)** or **Glassdoor.com (https://www.glassdoor.com/)**. Expect to see titles such as

software engineer, data scientist, solutions architect, and software development engineer.

It's no secret that machine learning is making waves in the tech field in the United States and worldwide. With widespread adoption of machine learning in essentially every industry, machine learning engineers aren't going to be struggling to find open positions any time soon.

So, let's jump into more technical interviewing practice so that you can be ready to nail that interview when the time comes.

Technical Interview Preparation

When preparing for an interview in machine learning, remember that, just like any other interview, there are things you can do to prepare. Let's review those.

- **Bring clean, printed copies of your resume.** Having extra copies of your resume on hand is important in case any of the interviewers don't have a copy. Also, you can use your resume as a reference point for anything you'd like to talk about.

NOTE

> Interviews typically start with questions about who you are, what your background is, and why you're interested in the job. The question "tell me about yourself" is not an invitation to tell them about your childhood experiences. Usually, an interviewer wants to know your education and professional background that brought you to where you are now.

- **Look online for past interview questions for similar roles.** If you're interviewing at a large company, there's a good chance that the questions they'll use already exist online. Usually, you can find these at **Glassdoor (https://www.glassdoor.com)** or on employment and

interview blogs. Be sure to read them and solve the problems posed—this way, you'll be ready for the specific questions you'll likely be asked.

- **Ask the recruiter or human resources (HR) about the structure of the interview.** There's nothing wrong with asking for the structure of the interview: How long will it be? Is there a technical component? With whom will I meet? All of these are valid questions to ask so that you can know what to expect and prepare accordingly. Try to get the name of the interviewers so that you can research their backgrounds on LinkedIn.
- **Conduct mock interviews.** Mock interviews are never fun. They can be awkward and weird. Few people are naturally good at interviewing. It takes time and practice to learn how to be comfortable being uncomfortable. You can conduct mock interviews with classmates, friends, family members, or at local career meetups and events.
- **Practice relevant skills.** Use the description of the target job to inform your preparation. For example, if it cites a specific need for a Python developer, then practice solving problems in Python. If the potential employer uses a technology you're unfamiliar with, try to gain a working knowledge about the tool so that you can discuss it in the interview.

Practice, Practice, Practice

This week, we want you to practice searching and compiling interview questions, then practice answering as many you possibly can.

Below are some sites to get started:

- **Machine Learning Interview Questions on Glassdoor.com (https://www.glassdoor.com/Interview/machine-learning-interview-questions-SRCH_KO0,16.htm)**
- **Cracking the Machine Learning Interview (https://medium.com/subhrajit-roy/cracking-the-machine-learning-**

interview-1d8c5bb752d8)

- 10 Essential Machine Learning Interview Questions (https://www.toptal.com/machine-learning/interview-questions)
- Top 34 Machine Learning Interview Questions & Answers for 2020 (https://www.simplilearn.com/machine-learning-interview-questions-and-answers-article)
- 41 Essential Machine Learning Interview Questions (https://www.springboard.com/blog/machine-learning-interview-questions/)

Continue to Hone Your Skills

If you're interested in hearing more about the technical interviewing process and practicing algorithms in a mock interview setting, check out our upcoming workshops (https://careerservicesonlineevents.splashthat.com/) .

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

Please click **Start** when you are ready to begin the activity.

Start

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

1 of 20

Using R, how would you import a CSV file, "data_file.csv" into your environment? Select all R statements that could import a CSV file without error.

- [ ] `read.csv(file=data_file.csv',check.names=F,string`
- [ ] `read(file=data_file.csv')`
- [ ] `csv(file=data_file.csv',header=T)`
- [ ] `read.csv(check.names=F,stringsAsFactors = F)`
- [ ] `read(file=data_file.csv',check.names=F,stringsAsF`
- [ ] `read.csv(file=data_file.csv',stringsAsFactors = F`

- Item 1
- Item 2
- Item 3
- Item 4
- Item 5
- Item 6
- Item 7
- Item 8
- Item 9
- Item 10

◄

Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

You are trying to summarize the heights of your students using the tidyverse in R. Within your R environment you have a dataframe called “test_df” that contains the names of each student and their heights in inches.

Select the correct R statement that produces a dataframe containing summary statistics of the student’s heights:

- `sum(test_df,Mean_Height=mean(height),SD_Height=sd`
- `test_df %>% summarize(Mean_Height=mean(height),SD_`
- `test_df %>% sum(Mean_Height=mean(height),SD_Height`
- `test_df %>% summary(Mean_Height=mean(height),SD_H`

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄
Next ►

Unit Assessment: Analytical Modeling and Big Data

Consider that you have imported a dataframe into R with three columns - `model` , `year` , `Mean_Hwy` . You wish to create a heatmap that compares the mean highway fuel-efficiency across vehicle models and years.

Complete the following code below so that it will do the following tasks:

1. Create a ggplot2 object.
2. Add a heatmap plot to the ggplot2 visualization.
3. Rotate the x-axis labels 45 degrees.
4. Add labels to x-axis, y-axis, and legend.

```
plt <- ggplot(data,aes(
  
  ,
  
  ,
  
))

plt +
  
 + labs(
  
  ,
  
  ,
  
) + theme(
  
)
```

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄
Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

Match the following definitions of statistical concepts with the correct term in the word bank:

1. [ ]: builds a linear regression model with two or more independent variables.
2. [ ]: known as Ha and is generally the hypothesis that is influenced by non-random events.
3. [ ]: a data type that can be subdivided infinitely.
4. [ ]: denoted as “r” in mathematics and is used to quantify a linear relationship between two numeric variables.
5. [ ]: a statistical test used to compare the

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄ Next ►

distribution of categorical frequencies across two groups.

6. _____: a data type that is either one of two categories.
7. _____: a statistical test used to test the means of a single dependent variable across a single independent variable with multiple groups.
8. _____: a statistical test to quantify the probability of whether or not the test data came from a

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

As lead data engineer, you are tasked with handling the website shop orders. Every minute a new batch of orders comes in that needs to be processed. Which of the four V's would best describe this data?

- Velocity
- Veracity
- Variety
- Volume

◄ Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

You are working with a DataFrame that can contain the sales for each day for the past 5 years. The DataFrame has already been loaded into your notebook. In order to process the data, you first group the data by the date, then sum all the total sales for the day. After this you took a look at the resulting DataFrame to make sure it is correct, then filtered for dates in the past year before displaying the final result.

Which is the correct order of actions and transformations you called on the DataFrame?

- Action -> Transformation -> Transformation -> Transformation -> Action
- Transformation -> Transformation -> Action -> Transformation -> Action
- Action -> Transformation -> Transformation -> Action -> Transformation

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄
Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

You have been tasked with analyzing the common voice commands used in your company's new software for home voice activation software. You have been given the data set, and you have processed it and separated the text into a list of words.

What would be the next stage in the NLP pipeline to perform?

- Stop Word Removal
- Normalization
- Term Frequency-Inverse Document Frequency
- Tokenization

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄ Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

Which of the following is not true about using AWS's S3 to store your data?

- S3 allows for data files to be accessed easily across multiple people.
- S3 can be directly read into a notebook and stored into a DataFrame.
- S3 are public by default but can easily be made private.
- S3 allows for massive storage of data without any normalization required.

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄
Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

You're a data analyst for a school district. Based on data of high school students, such as average grade, number of missed school days, number of detentions and suspensions, and reduced lunch rate status, your task is to identify at-risk students who are predicted to drop out in the upcoming academic year.

Which of the following libraries is most directly relevant in performing your task?

- `sklearn.linear_model.LinearRegression`
- `sklearn.decomposition.FactorAnalysis`
- `sklearn.linear_model.LogisticRegression`
- `sklearn.exceptions`

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄
Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

How will `sklearn.model_selection.train_test_split` help you accomplish your task? Choose the best answer.

- Splitting the dependent variables is necessary to instantiate a machine learning model.
- Splitting the dependent and independent variables allows parametrization.
- Splitting a dataset into training and testing sets allows validation of the machine learning model used.
- Splitting the independent variables is necessary to instantiate a machine learning model.

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄
Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

In your discussion with the superintendent, she informs you that her aim is to identify as many potential drop-out candidates as possible, and to monitor them during the year. Which of the following is the most relevant validation metric for your model?

- Precision
- Recall
- Accuracy
- RMSE

Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10

◄
Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

You also decide to explore using other models. Which of the following observations is incorrect?

- A benefit of the random forest classifier is the ability to rank the importance of features.
- The choice of the kernel in SVM can yield very different results.
- SVM, unlike decision trees, does not benefit from feature scaling.
- Decision tree models can be prone to overfitting.

◄ Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

A large amount of raw financial data has just been uploaded to your data storage. You have been tasked with priming the raw data so unsupervised learning models can be applied and the results analyzed. Which one of the following is not something you should consider?

- How can I get this data to be used to create great visualizations?
- Will I be able to easily hand off this data set to other teams?
- Does the data contain excess data that we don't really need?
- Are there different types of data that ?

Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10
Item 11

◄ Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

How does the K-Means algorithm determine how many clusters are made and which data points belong to them?

- The algorithm looks at the data points based on similarity or distance then finds the optimal amount of clusters.
- The amount of clusters is given to the algorithm which then assigns data points to the cluster based on similarity or distance.
- The algorithm first finds the ideal amount of clusters then randomly assigns the data points to a cluster.
- The data points are broken into groups and the algorithm determines if those points should be placed in more or less clusters.

Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10
Item 11
Item 12
Item 13

◄
Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

Which form of dimensionality reductions does Principal Component Analysis perform?

- Feature Extraction
- Feature Elimination
- Feature Reduction
- Feature Compilation

◄ Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

Your company is looking to open a new location and you are given data sets with all of it's competitors information and are asked to cluster it to see where your demographic can best be served. Which would be the best reason to use hierarchical clustering over K-Means clustering for this data set.

- The data set does not make it clear how many different clusters should be known ahead of time.
- The data set lends itself to be grouped based off distance.
- The data is not super large and does not need to be analyzed with super high performance.
- The data set lends itself to be better visualized with a dendogram instead of an Elbow Curve

Item 5
Item 6
Item 7
Item 8
Item 9
Item 10
Item 11
Item 12
Item 13
Item 14

◄ Next ►

Unit Assessment: Analytical Modeling and Big Data

Match the following definitions of neural network and modeling concepts with the correct term in the word bank:

1. [ ]: a dataset that can be separated perfectly using a straight line
2. [ ]: reduces the number of unique categorical variables in a dataset
3. [ ]: the model performance score based on the inaccuracy of a single input
4. [ ]: returns a value from 0 to infinity, so any negative input through the function is 0
5. [ ]: a single neuron within a neural network model
6. [ ]: transforms the output to a range between 0 and 1. It is the same curve used in logistic regression
7. [ ]: a

Item 7
Item 8
Item 9
Item 10
Item 11
Item 12
Item 13
Item 14
Item 15
Item 16

◄ Next ►

Complete the following code below so that it will do the following tasks:

1. Define the deep learning classification model
2. Add the first hidden layer with a non-linear activation function
3. Add the second hidden layer with less neurons than the first hidden layer
4. Add an output layer
5. Compile the model
6. Check the model structure

```
nn_model = tf.keras.models.

nn_model.add(tf.keras.layers.
)

nn_model.add(tf.keras.layers.
)

nn_model.add(tf.keras.layers.
)

nn_model.

nn_model.
```

Item 8
Item 9
Item 10
Item 11
Item 12
Item 13
Item 14
Item 15
Item 16
Item 17

◄ Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

The following steps are used to preprocess categorical and numerical data for use in a neural network model. Arrange the steps in the correct order:

- ≡ Split input dataframe into training and testing datasets
- ≡ Encode categorical variables using `OneHotEncoder` instance
- ≡ Fit `StandardScalar` instance on training data
- ≡ Scale training and testing data using fitted `StandardScalar` instance
- ≡ Perform bucketing transformation on any applicable categorical variable
- ≡ Merge the encoded categorical dataframe with the

Item 9
Item 10
Item 11
Item 12
Item 13
Item 14
Item 15
Item 16
Item 17
Item 18

◄ Next ►

Unit Assessment: Analytical Modeling and Big Data

You are about to complete the Analytical Modeling and Big Data Unit Assessment! This Unit Assessment allows you to check your knowledge, as well as demonstrate your competency in key concepts from Modules 15 through 19.

After submitting the assessment, you will see a summary of your performance. While you will not be able to see your performance on individual questions, you are allowed unlimited attempts to complete the assessment.

Unit Assessment: Analytical Modeling and Big Data

◄ Finish ►

The following code builds, trains, and evaluates a neural network model. Which of the following code changes is the most likely to optimize the model's performance?

```
# Define the basic neural network model
nn_model = tf.keras.models.Sequential()
nn_model.add(tf.keras.layers.Dense(units=16, a
ctivation="relu", input_dim=8))
nn_model.add(tf.keras.layers.Dense(units=1, ac
tivation="sigmoid"))

# Compile the Sequential model together and cu
stomize metrics
nn_model.compile(loss="binary_crossentropy", o
ptimizer="adam", metrics=["accuracy"])

# Train the model
fit_model = nn_model.fit(X_train_scaled, y_tra
in, epochs=100)
```

- ○ Swap the training and testing datasets
- ○ Remove the output layer
- ◉ Change the number of neurons in the hidden layer