AI Bootcamp

Neural Networks for Classification

Module 18 Day 2

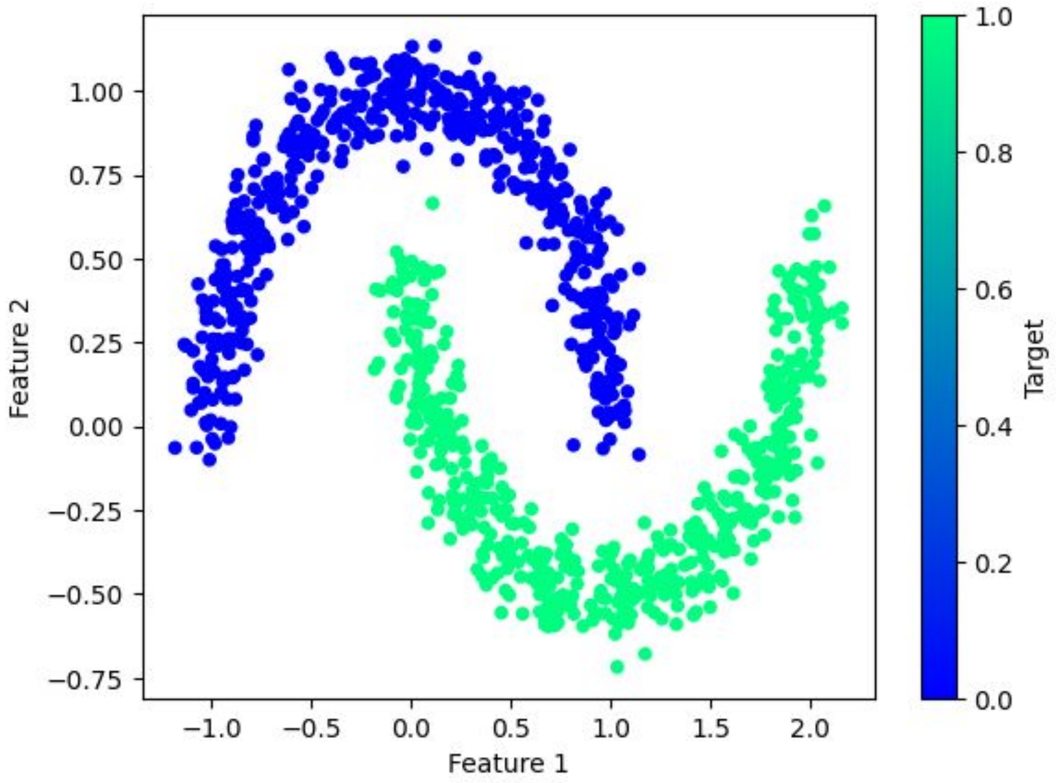
Class Objectives

By the end of class, you will be able to:

1. Implement deep neural network models using TensorFlow.
2. Explain how different neural network structures change algorithm performance.
3. Use KerasTuner to assist with finding optimal neural network structures.
4. Save trained TensorFlow models for later use.

We Must Dig Deeper Into **Neural Networks**

Over the Moon on Basic Neural Networks



Similar to the swirl input data we saw in **TensorFlow Playground**, the moons dataset is not linearly separable.

Our next steps are **exactly the same** as those in the last class.

Over the Moon on Basic Neural Networks

Next steps:

01 Split the dataset, then scale and standardize each feature.

02 Fit the model to our training data.

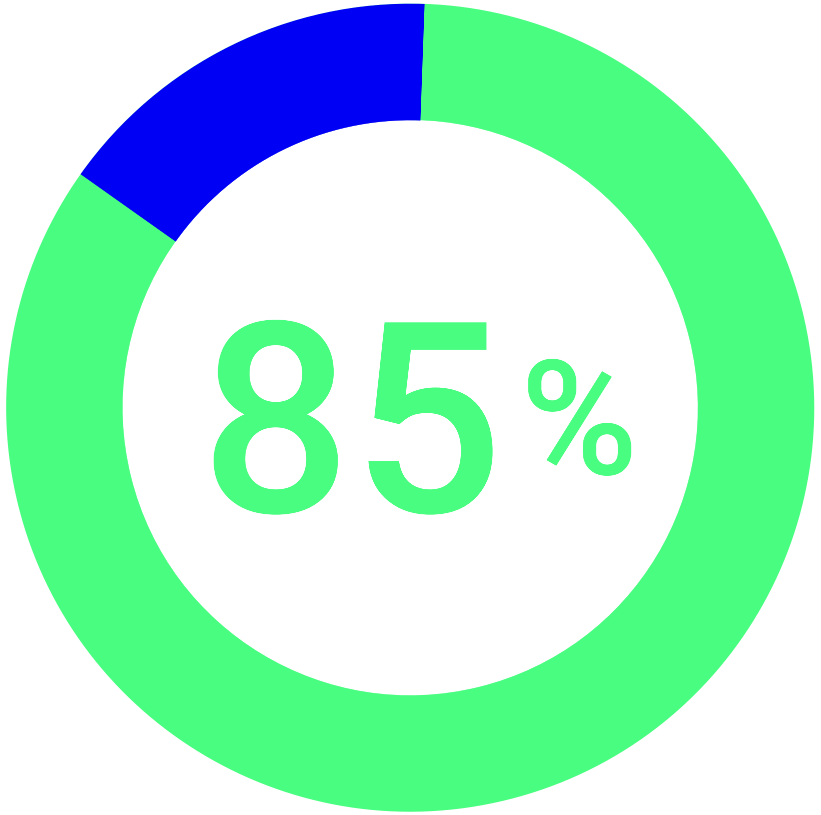
03 Transform the data.

04 Create a Sequential model.

05 Add the first layer and output layer, and then get the structure of the Sequential model.

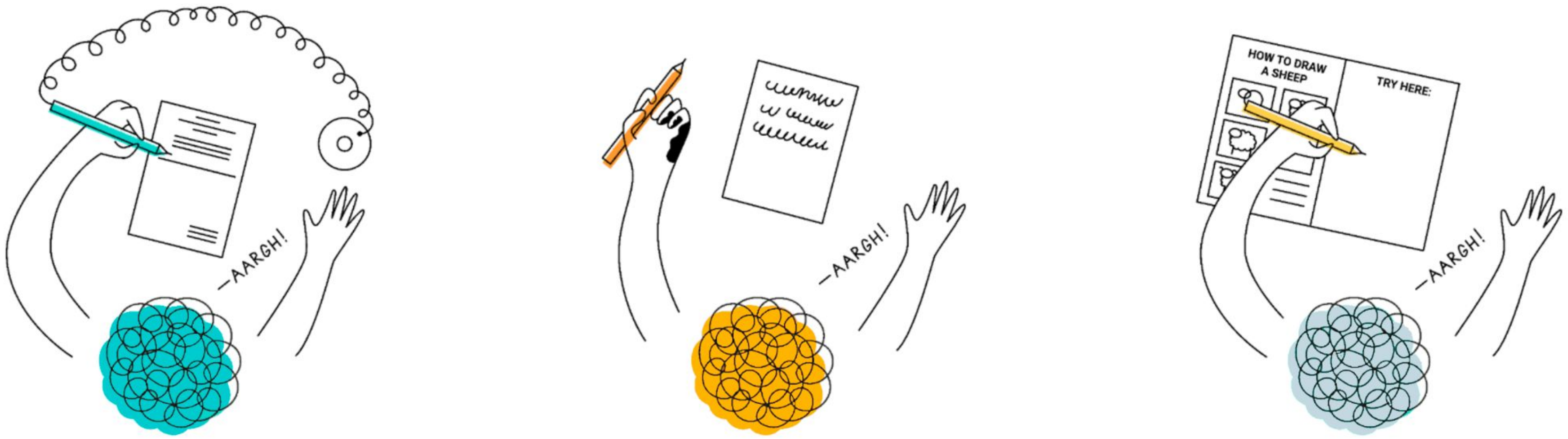
Over the Moon on Basic Neural Networks

Depending on the dataset or the use case for the model, a predictive accuracy of approximately 85% may be sufficient for a first-pass model.



Over the Moon on Basic Neural Networks

For example, let’s say we were trying to build a neural network that can predict if students are left-handed or right-handed. A model that was able to make correct predictions 85% of the time would be pretty accurate!



Over the Moon on Basic Neural Networks

Many industrial and medical use cases require a machine learning model to exceed 95% or even 99% classification accuracy.

In these cases, we could not use the basic single-neuron, single-layer model.

Over the Moon on Basic Neural Networks

01 One possible solution to our performance problem is to add more neurons.

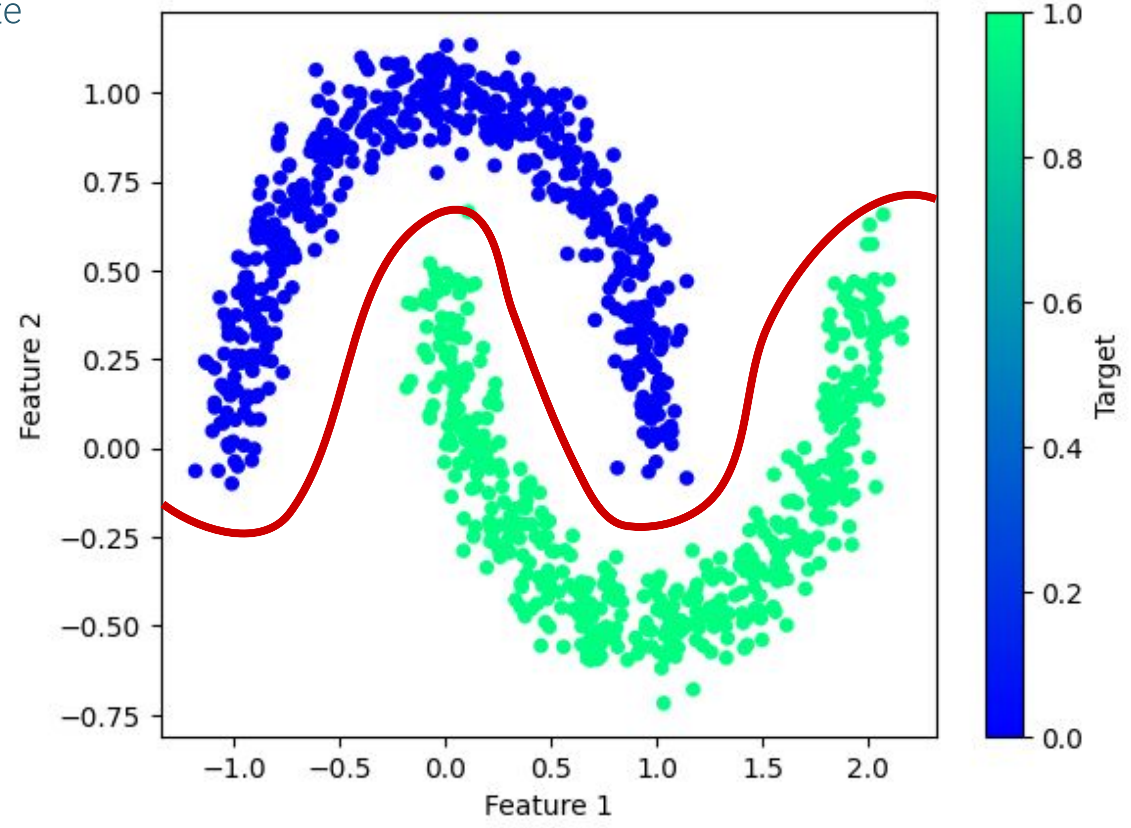
02 This is the most straightforward solution, but it is not the most robust.

03 Adding more neurons to a single hidden layer only boosts performance if there are subtle differences between values.

Over the Moon on Basic Neural Networks

For our dataset, the only way to separate the two groups is with a complex polynomial line. A single-layer, multiple-neuron model would still struggle to adequately classify our two groups with only two inputs.

In these cases, we must create a neural network model capable of identifying complex nonlinear relationships.



Questions?

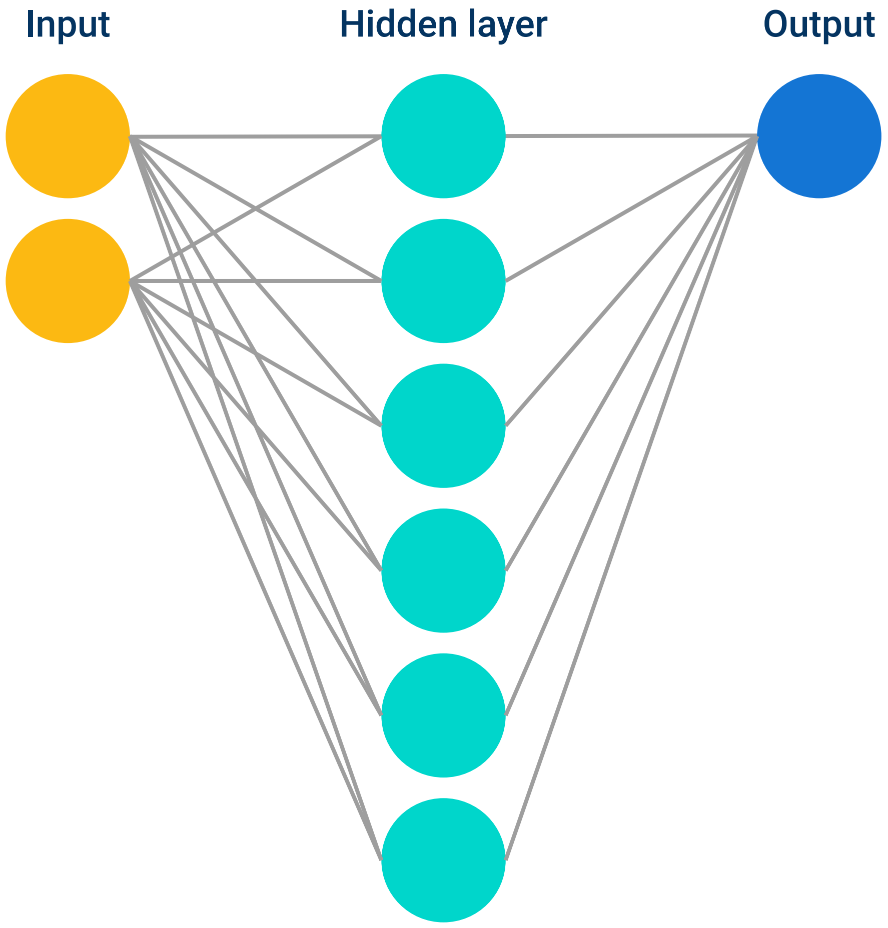
Instructor **Demonstration**

Getting Deep with Deep Learning Models

The Neural Network

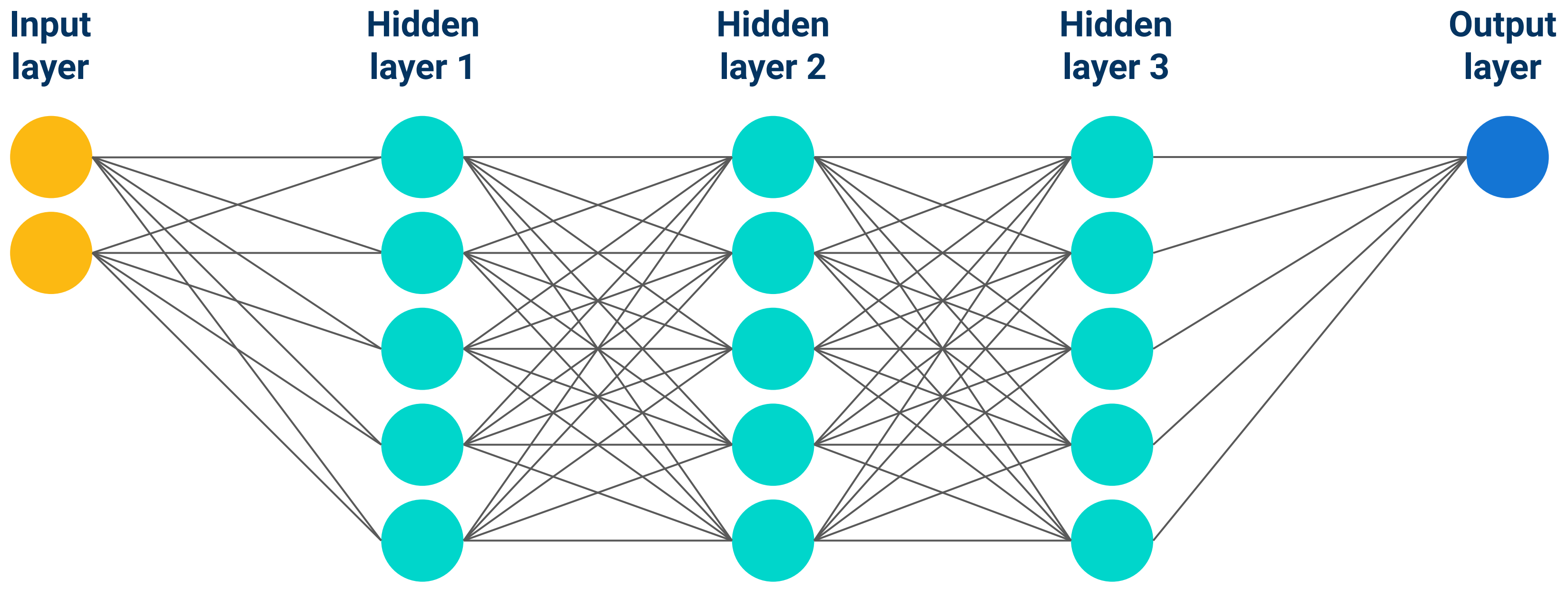
Basic neural network models are:

- Designed to evaluate input values *once* before they are used in an output classification or regression equation.
- Limited to interpreting simple linear relationships and data with few confounding factors, or factors that have hidden effects on more than one variable.



The Neural Network

To address and overcome the limitations of the basic neural network, we can implement a more robust neural network model by adding more hidden layers.



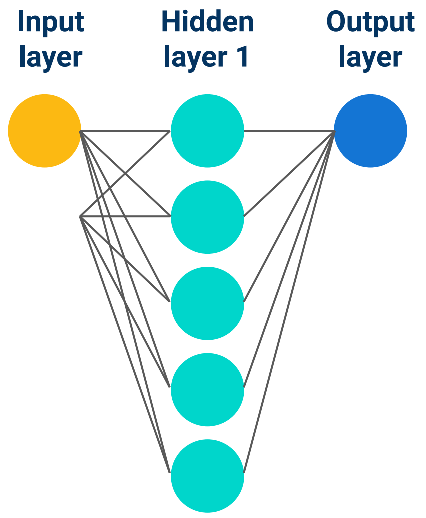
A neural network model with more than one hidden layer is known as a **deep neural network** or **deep learning model**.

Deep learning models function similarly to the basic neural network, with **one major difference**.

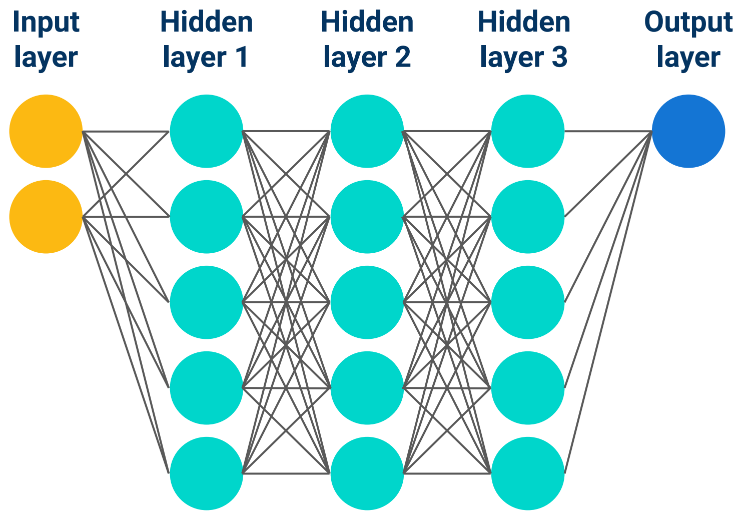
The Neural Network

The outputs of one hidden layer become the inputs to additional hidden layers of neurons. This enables the next layer of neurons to evaluate higher-order interactions between weighted variables and to identify complex, nonlinear relationships.

Basic Neural Network

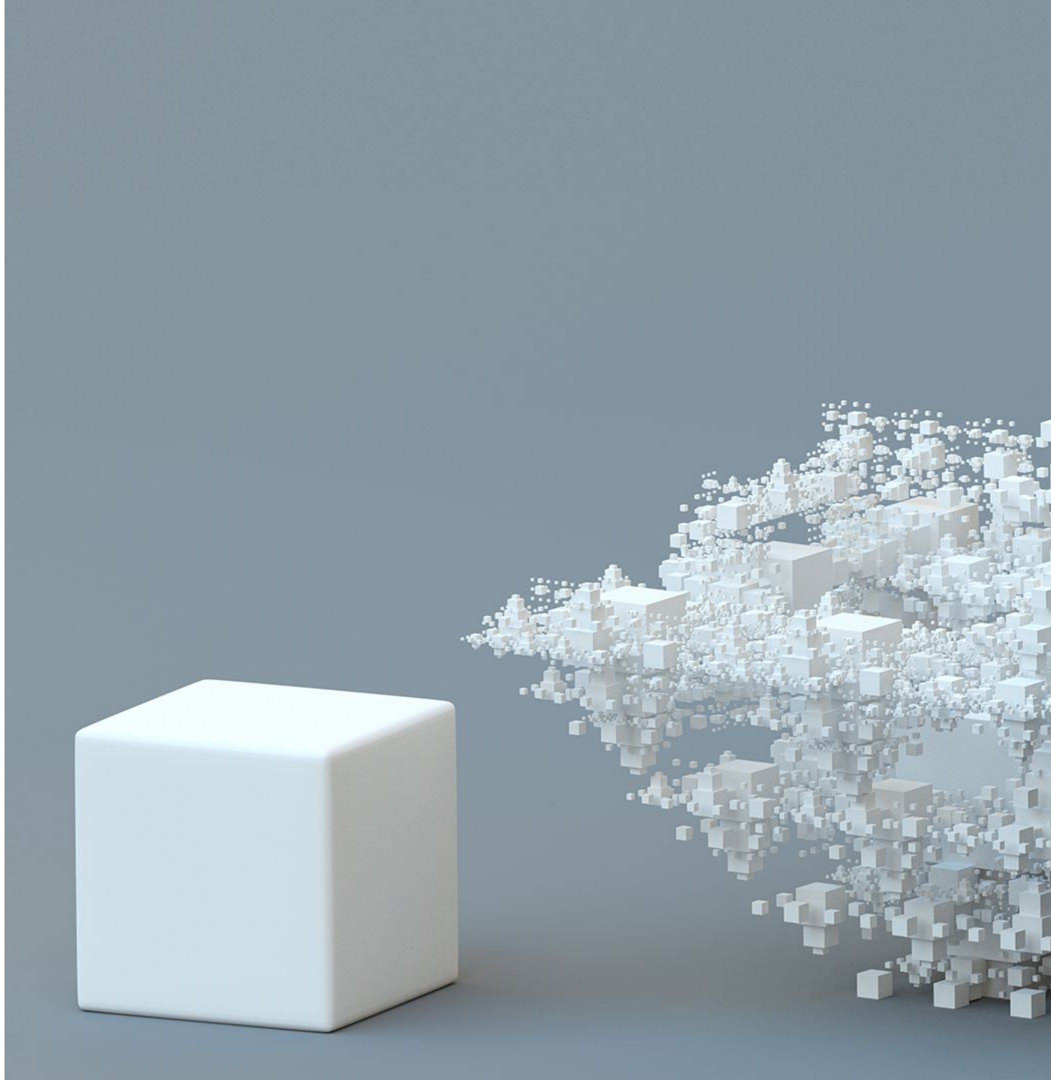


Deep Learning Model

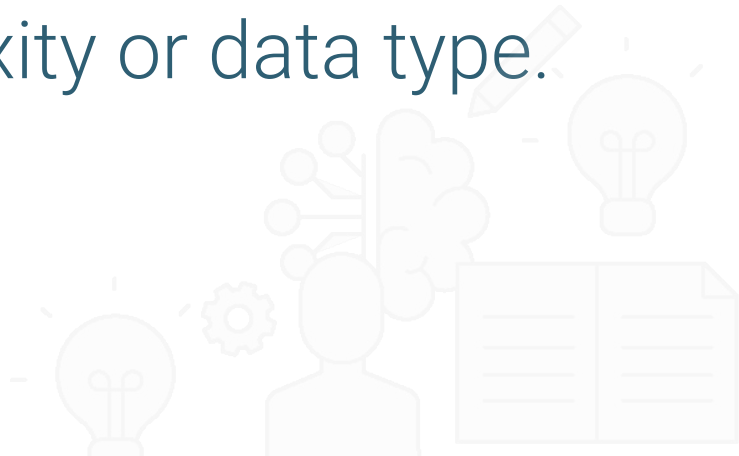


Getting Deep with Deep Learning Models

A deep learning model can identify and account for more information than any number of neurons in any single hidden layer.



Deep learning models are named for their **ability to learn** from example data, regardless of the complexity or data type.



Features of Deep Learning Models

Just like humans, deep learning models can:

- Identify patterns
- Determine severity
- Adapt to changing input data from a wide variety of sources

Getting Deep with Deep Learning Models

Many data scientists believe that even the most complex interactions can be characterized by as few as **three hidden layers** (there is an ongoing debate among scientists about this).

With minimal preprocessing and direction, deep learning models can train on:

- Images
- Natural language data
- Soundwaves
- Traditional tabular data

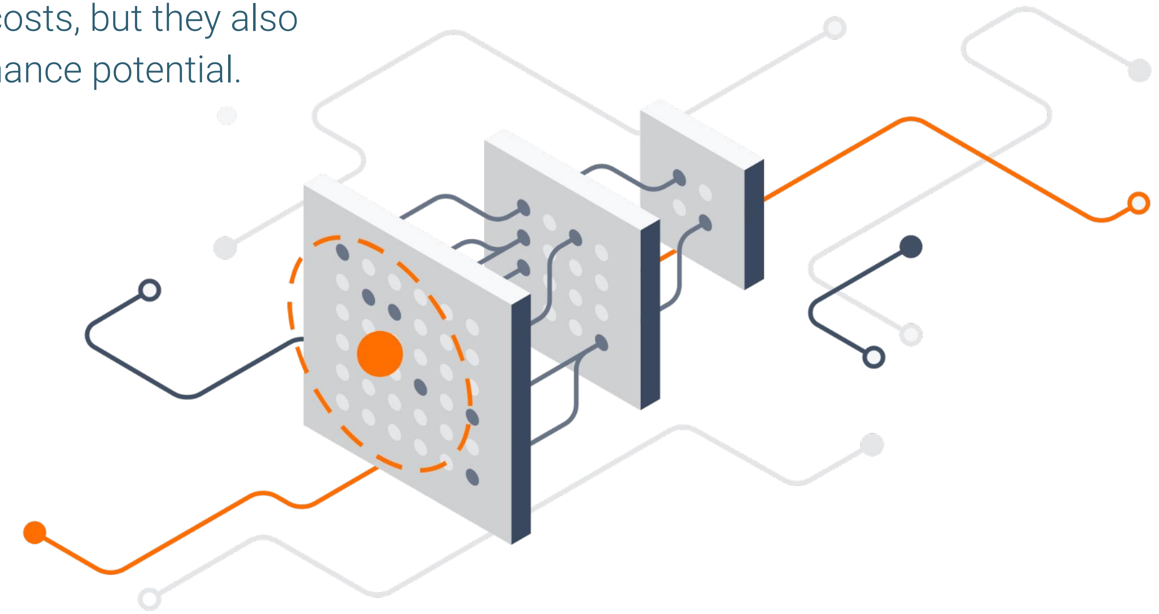
3

Without access to powerful computers and libraries like **TensorFlow**, data scientists were limited in their ability to create and run complex models.

Deep Learning Models: TensorFlow

Deep learning models typically require longer training iterations and memory resources than their basic neural network counterparts, but they achieve higher degrees of accuracy and precision.

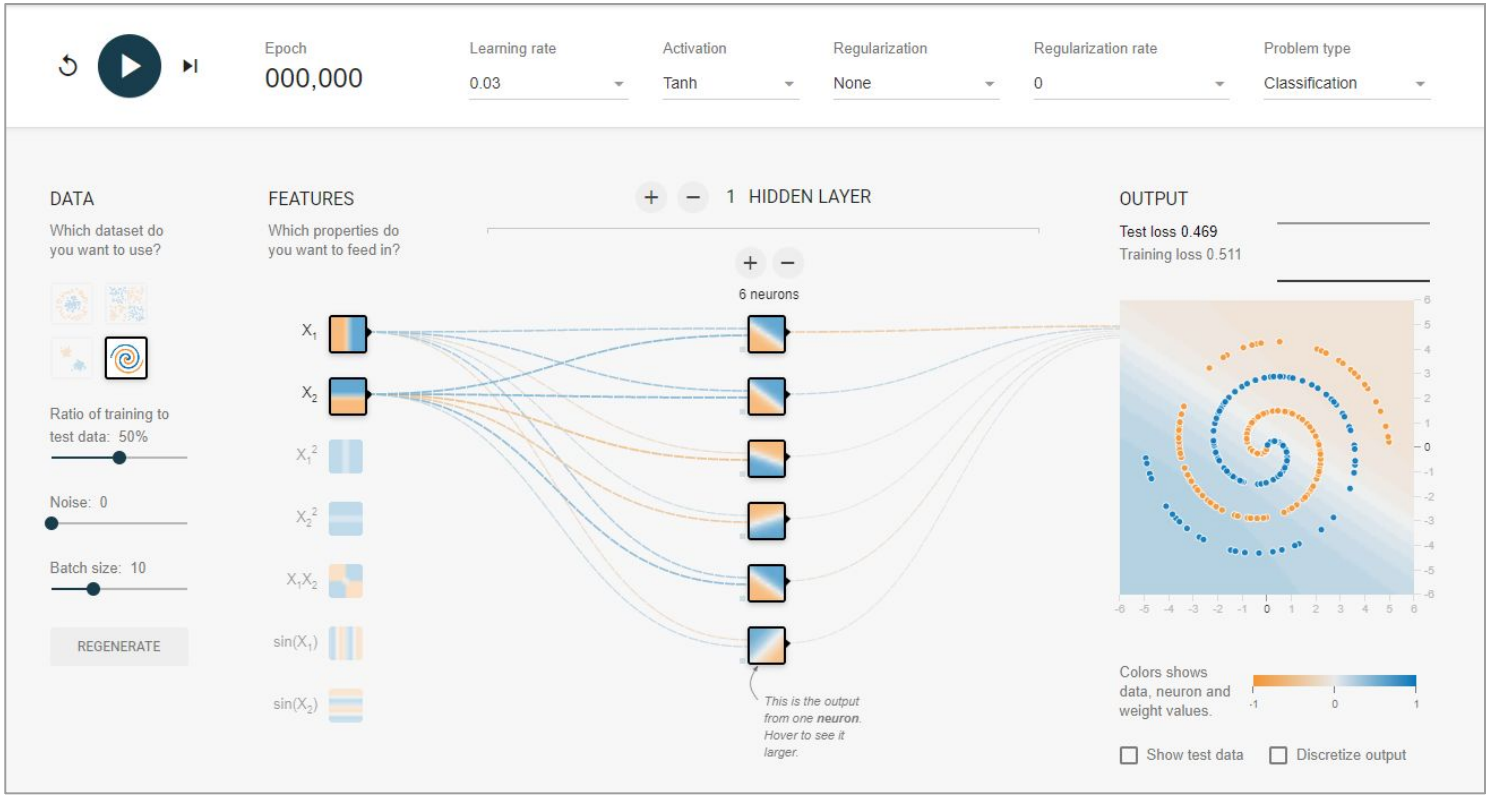
In other words, deep learning models may have more upfront costs, but they also have higher performance potential.



Source: https://www.tensorflow.org

Deep Learning Models: TensorFlow

The easiest way to conceptualize the performance differences between basic neural networks and deep learning models is to return to the **TensorFlow Playground**.



Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

1 HIDDEN LAYER

6 neurons

This is the output from one ***neuron****. Hover to see it larger.*

OUTPUT

Test loss 0.469

Training loss 0.511

Colors shows data, neuron and weight values.

Show test data

Discretize output

There are **drawbacks** to building a deep learning model with too many layers.

Drawbacks

Deep learning models:

01 Require more computational resources (memory and CPU power) for each layer.
If we have limited resources or time, a larger deep learning model may not be possible.

02 Take considerably more time to train than a basic neural network. Each hidden layer adds complexity and more computations.

03 With multiple hidden layers will require more training data to produce an acceptable level of performance.

Activity:

Back to the Moon

In this activity, you will build a deep learning classification model that can adequately predict the class from an example moon-shaped dataset.

Suggested Time:

15 Minutes

Time’s up!

Let’s review

Questions?

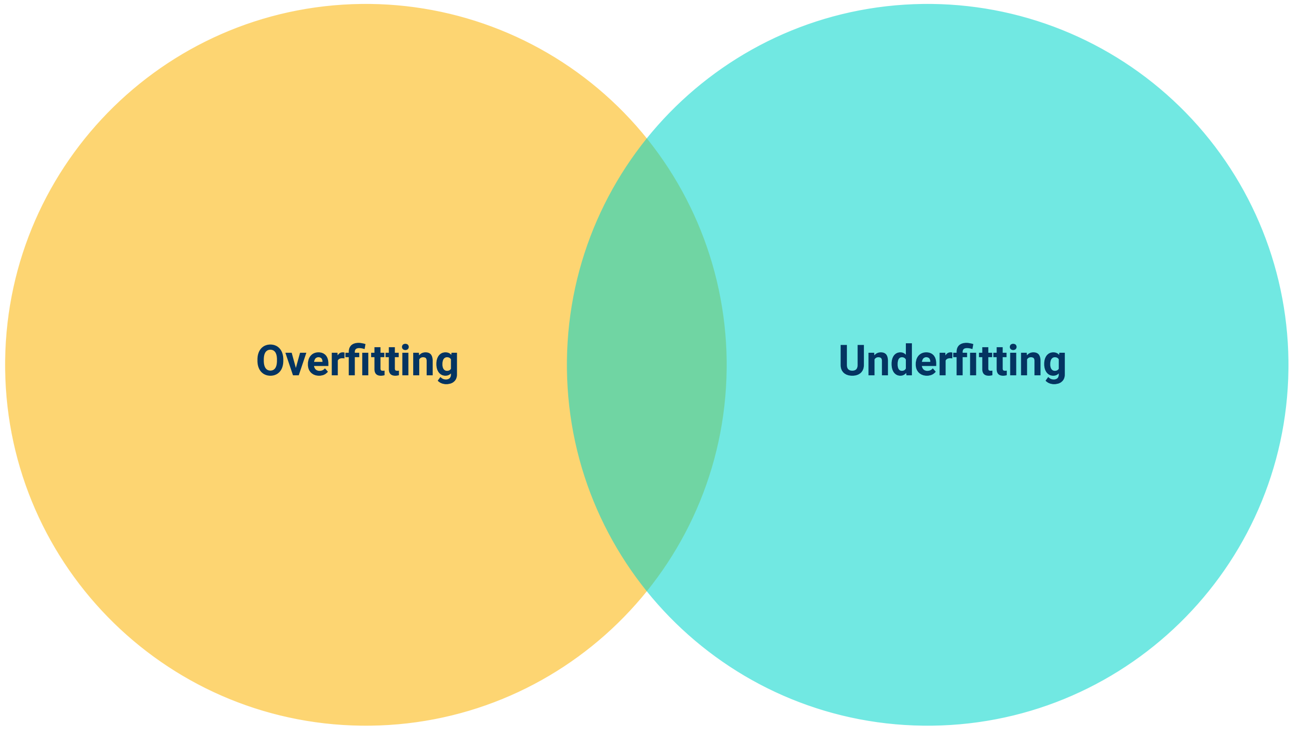
Instructor **Demonstration**

Getting Hands On with Model Optimization

As with any machine learning model, neural networks and deep learning models are **not perfect**.

Pain Points

There are two major pain points that you will commonly encounter:

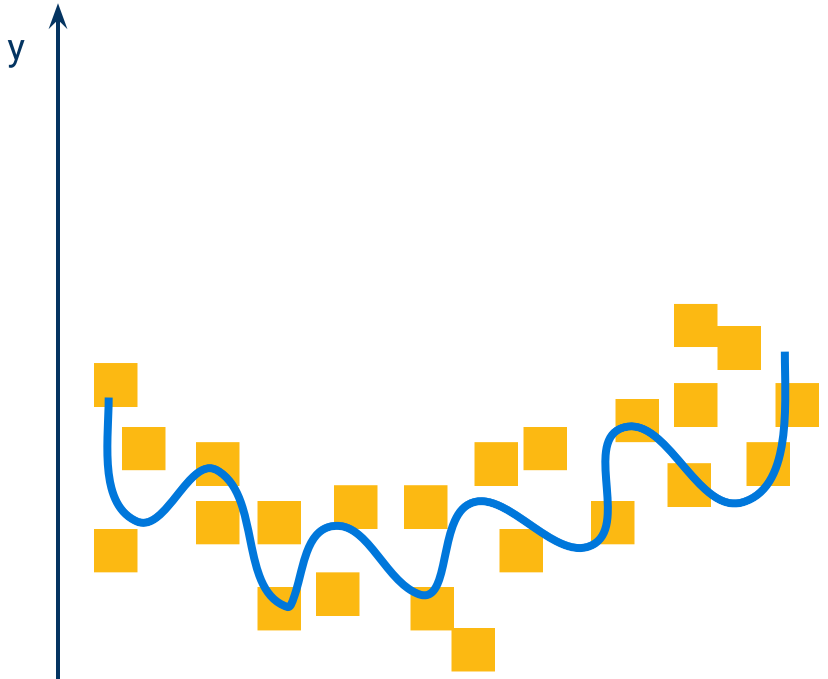


Overfitting

A model has **high variance**.

It adjusts too much to fit the training data and will not generalize well.

As a reminder, this is known as **overfitting** the model.



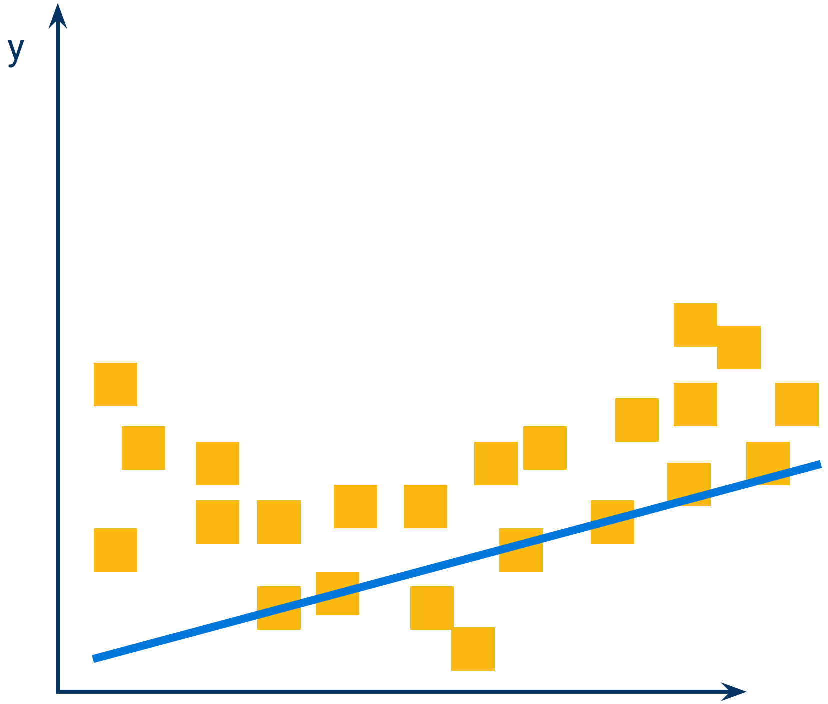
Overfitting

Underfitting

A model can have **high bias**.

Very noisy input data causes the model to **underfit** the data.

In other words, our model struggles to classify or predict our training dataset.



Underfitting

Model Optimization: Overfitting

There are two reasons for the model to **overfit the data** and not meet performance expectations.

01

The training and test data are unbalanced or the training data is not representative of the test data.

02

There is not enough complexity in the training data, and the model converges too quickly.

We will discuss convergence in detail later in the lesson.

The most straightforward way to fix a high-variance (overfitting) model is by **adding more training data**.

Add More Training Data

The first way to increase the training data is to **collect more data for your input dataset**.

Pro

This is the safest method of increasing training data, but the data must be collected properly, using the same protocol as initial data collection.

Con

The problem with collecting more input data is that it might be logistically or financially impossible to collect more data.

Add More Training Data

The second method of increasing the training data is **changing the split of training and testing data in the original input data**.

Pro

The benefit of this method is that we won’t need to collect new data; there is no additional financial or logistical cost.

Con

We’ll have less data available for testing and validating the model. This creates a higher risk that we might incorrectly consider an underperforming model to be adequate.

Add More Training Data

We can **keep our training data the same and retrain the model using fewer epochs**.

Pro

This can be a safer alternative for smaller, simpler datasets.

Con

It may be ineffective for larger datasets with many features.

Model Optimization: Underfitting

There are two reasons for the model to **underfit the data** and not meet performance expectations.

01

The training data contains too many outliers/confusing variables.

02

Model design parameters, often referred to as **hyperparameters**, are inadequate/inappropriate.

We should always start by checking the training data because the process is fast and straightforward.

Instructor **Demonstration**

Reaction Times in Tennis

Neural networks are tolerant of noisy characteristics in a dataset, but they can learn bad habits, too (like the brain does).

We have to identify the variables that contain a number of **potential outliers**.

These variables can affect our data preprocessing and cause more important variables and features to disappear.

Hyperparameters

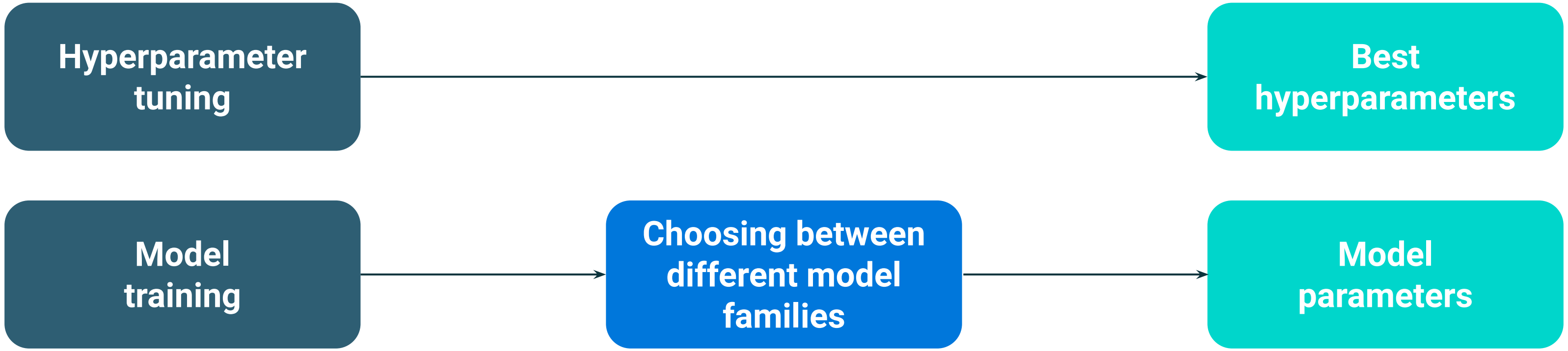
We will focus on higher-level hyperparameters that can be altered to achieve desired performance, such as:

- The number of neurons in a hidden layer
- The number of hidden layers in a deep learning model
- The activation function for each hidden layer
- The number of epochs in the training regimen

Hyperparameters

There are general rules we can apply to our hyperparameter tuning to make our models more effective.

- When building the initial model, we should use 2–3 times as many neurons as there are input features.
- If this does not achieve desired performance, we can always add more neurons if we have the computational resources.

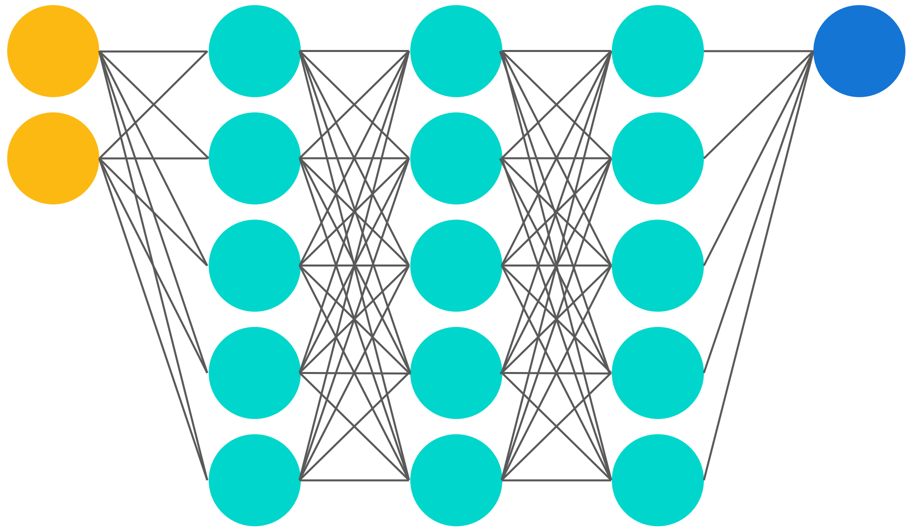


Hyperparameters

Similarly, we can try to boost the performance of a deep learning model by creating additional hidden layers.

1. Deep learning models require substantially more training iterations and memory resources with each additional hidden layer.
2. A good starting point for deep learning model optimization is limiting the model to 2–4 hidden layers.
3. Depending on the size and complexity of the input data, we may need to exceed the recommended number of hidden layers.

Deep Learning Model



One of the most effective methods of optimizing our neural network and deep learning models is to alter the **activation functions** for each hidden layer.

Activation Functions

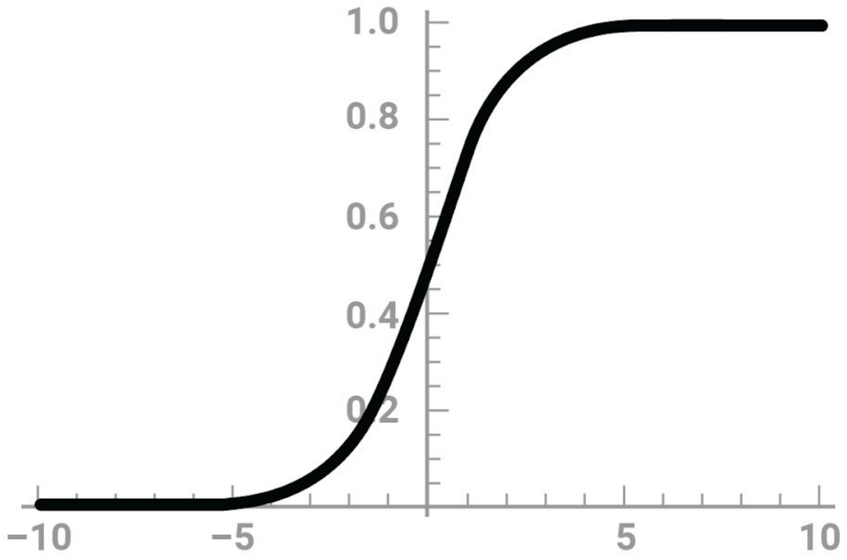
Depending on the shape and dimensionality of the input data, each **activation function** may focus on specific and different characteristics of the input values.

We need to use an activation function that matches the complexity of the input data. Each of the most popular activation functions has ideal use cases and datasets.

- Sigmoid function
- Tanh function
- Rectified linear unit (ReLU)
- Leaky ReLU function

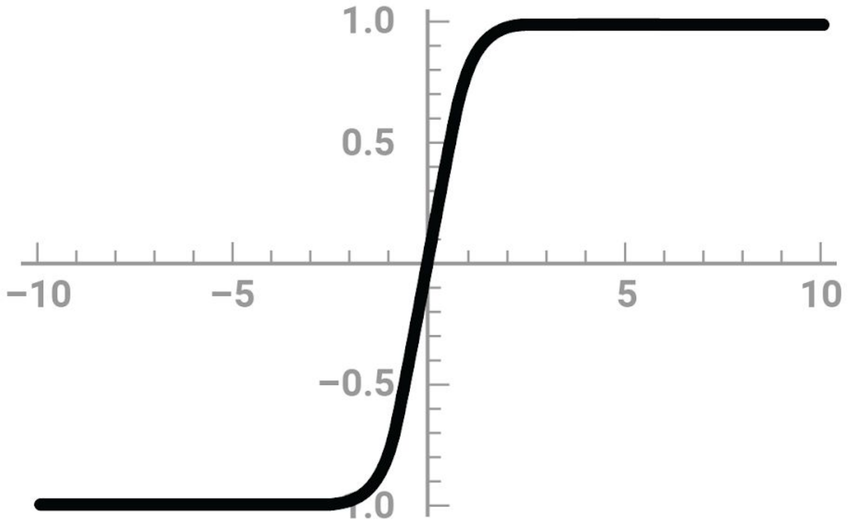
Sigmoid Function

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



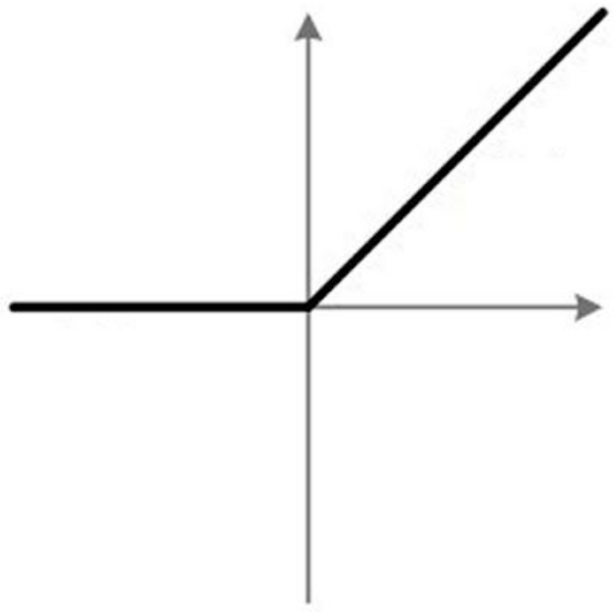
Tanh Function

We can use the **tanh function** for classification or regression because the normalized values range between −1 and 1.



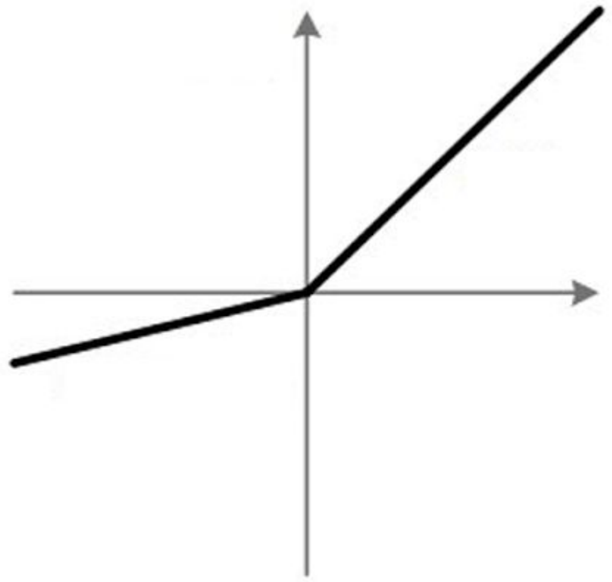
Rectified Linear Unit (ReLU)

The **rectified linear unit (ReLU)** function is ideal for modeling positive, nonlinear input data for classification or regression. The ReLU function is always a good starting point, but not all data is positive, especially when normalized.



Leaky ReLU Function

The **leaky ReLU function** is a good alternative to the ReLU function because of its ability to characterize negative input values.



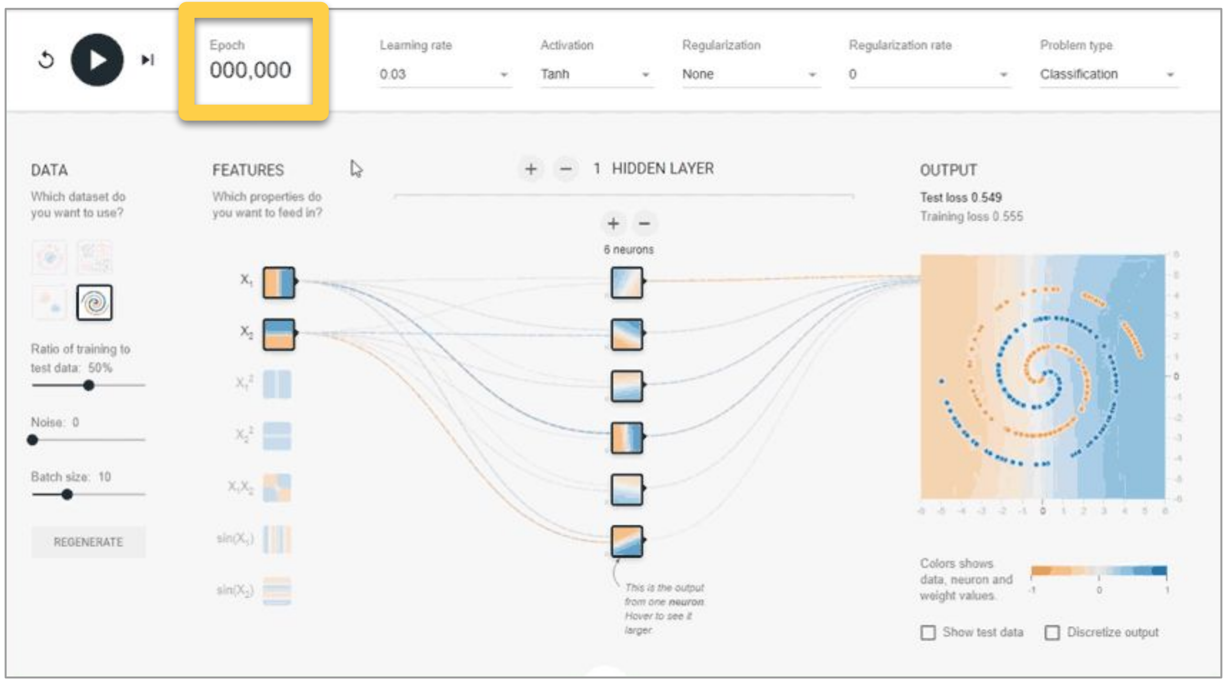
Activation Functions

It’s a good idea for the activation functions for your hidden layers to be slightly more complex than the activation functions for your output layer.

Using a higher-complexity activation function will assess the more complex components of the input data differently without any risk of censoring or ignoring lower-complexity features.

Epochs

If our model has still not met performance expectations, we can increase the number of training **epochs**.



Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

FEATURES

Which properties do you want to feed in?

1 HIDDEN LAYER

6 neurons

This is the output from one neuron. Hover to see it larger.

OUTPUT

Test loss 0.549

Training loss 0.555

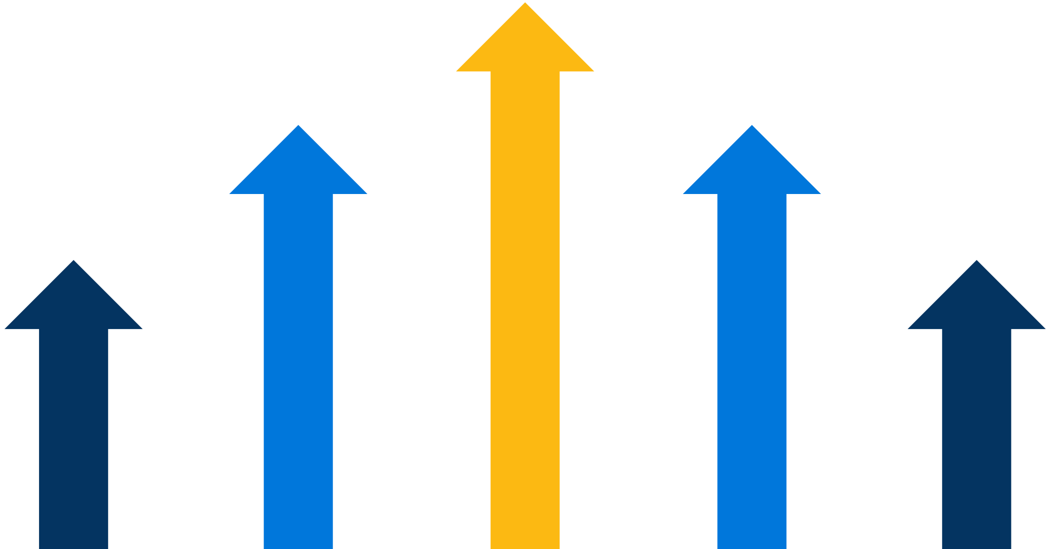
Colors shows data, neuron and weight values.

Source

Epochs

As the number of epochs increases, so does the amount of information provided to each neuron.

Each training iteration tweaks the neuron's weight coefficients; therefore, each epoch increases the likelihood that the model is utilizing effective weight coefficients.

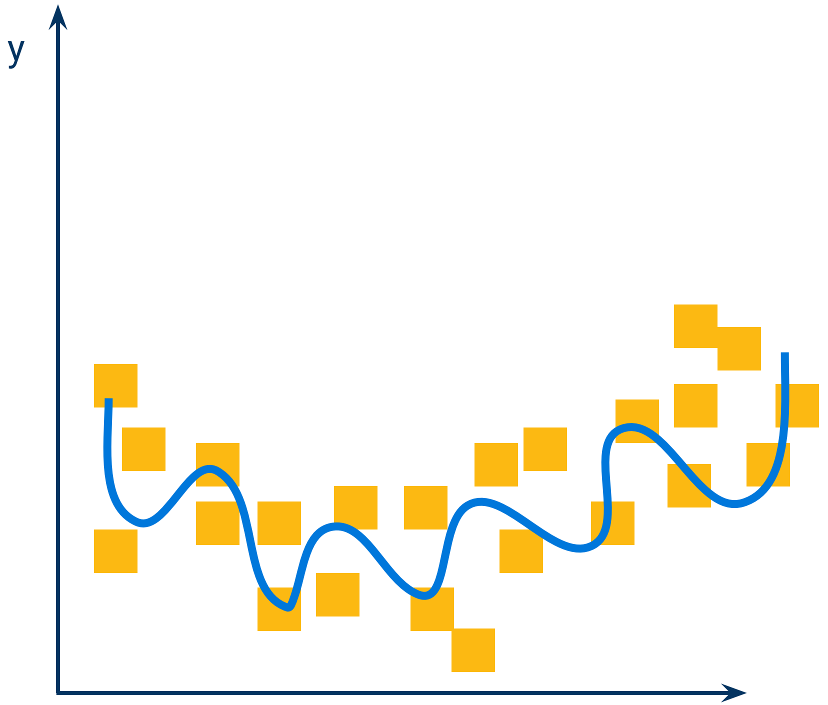


Epochs

If the model produces weight coefficients that are **too effective** at analyzing the training data (i.e., the model is custom tailored to meet the demands of the current data), then it may not generalize well.

As a reminder, this is known as **overfitting** the model.

We should test and evaluate our models each time that we increase the number of epochs to reduce the risk of overfitting.



Overfitting

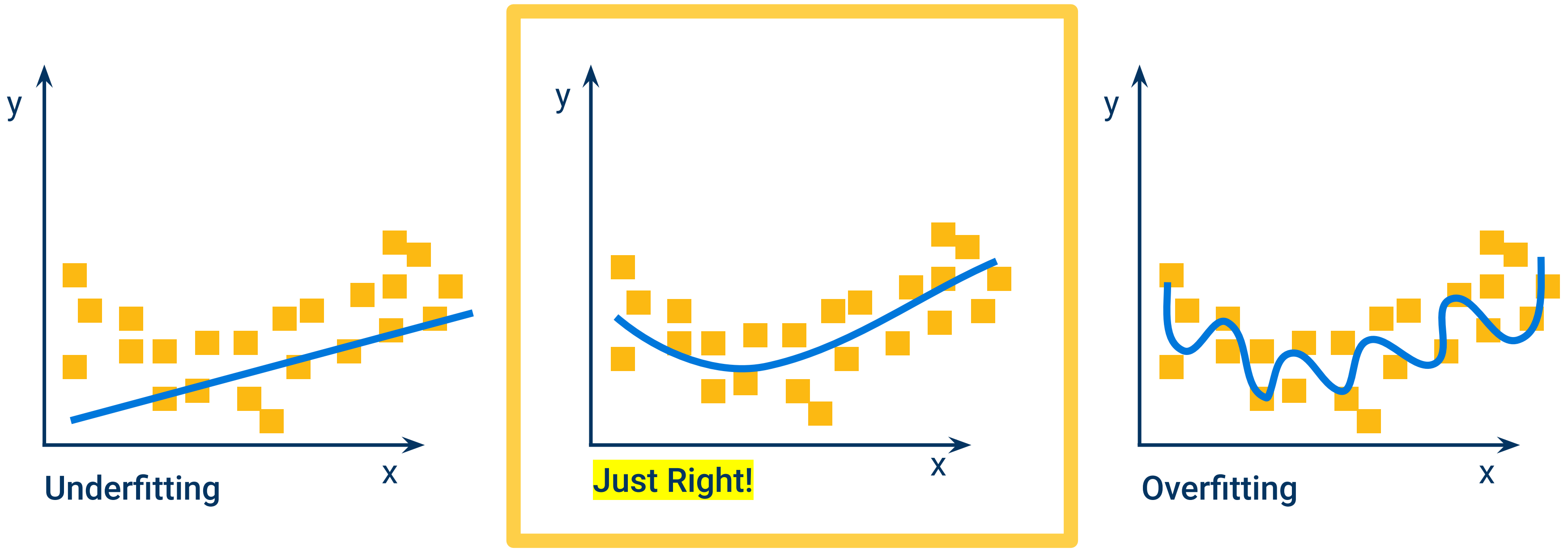
A good idea:

Start with a smaller number of training epochs (such as 100) and add more epochs until training loss starts decreasing at a slower rate.

Epochs

This threshold for the number of epochs can vary greatly between datasets.

For example, large datasets with hundreds of thousands of input values can start at 1,000 epochs (or even more!) without the risk of overfitting.



Modern Optimization Techniques

Hyperparameter Adjustment	Pros	Cons
Add more neurons	Speeds up model, may reduce loss	More computational resources required
Add more layers	Considers more interactions between variables	More computational resources required
Change activation function	Drastically changes how a model interprets inputs	Not all new interpretations are effective
Add more epochs	Increase likelihood that model will achieve optimal weight coefficients	Increased risk of overfitting

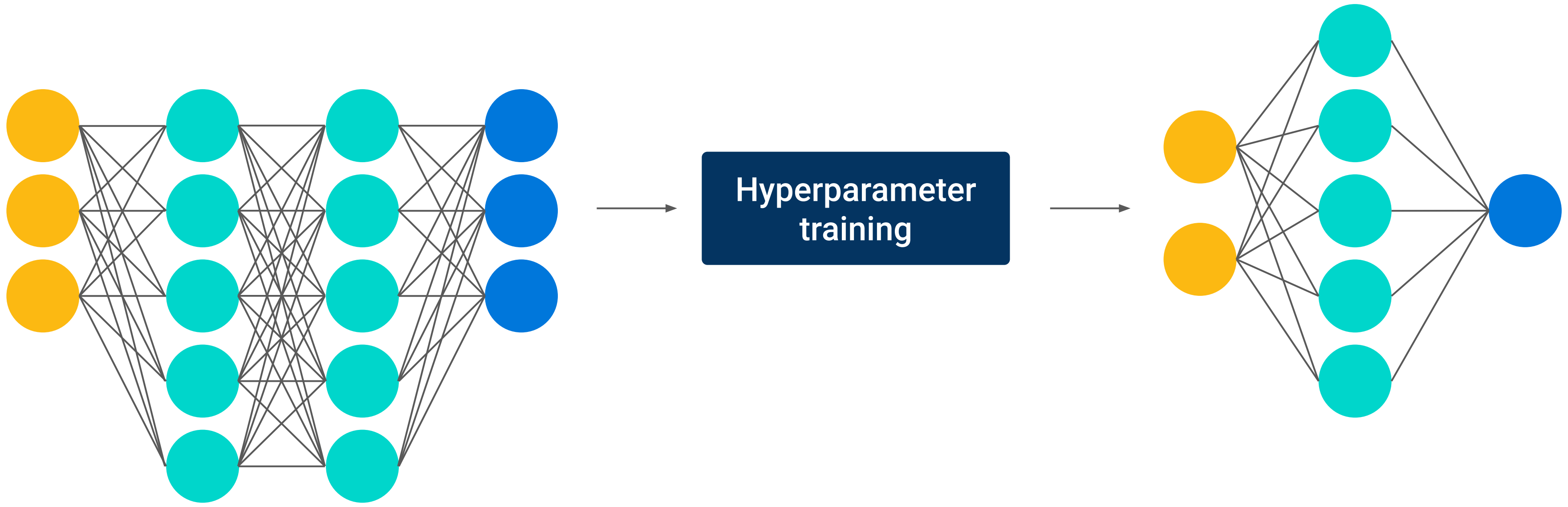
This list is not complete. There are more subtle and specific optimization tweaks that we can perform on TensorFlow neural network models.

Instructor **Demonstration**

Take the Guesswork Out of Model Optimization

Take the Guesswork Out of Model Optimization

Model optimization is often the most tedious and critical step in designing an effective machine learning model. Small changes to neural network model hyperparameters can cause large changes to overall model performance.



Take the Guesswork Out of Model Optimization

When developers released TensorFlow 2.0, they also released libraries and tools to automate neural network model optimization. These tools remove a lot of the guesswork for creating a nominal neural network and deep learning model.

Activity:

Giving Your Model Building a Tune-Up

In this activity, you will use KerasTuner to create a model that can adequately predict a nonlinear dataset that can be plotted as concentric circles.

Suggested Time:

20 Minutes

Time’s up!

Let’s review

Questions?

Break

15 mins

Instructor **Demonstration**

Getting Real with Neural Network Datasets

The last lesson of today's class covers how to prepare real-world data for our neural network models.

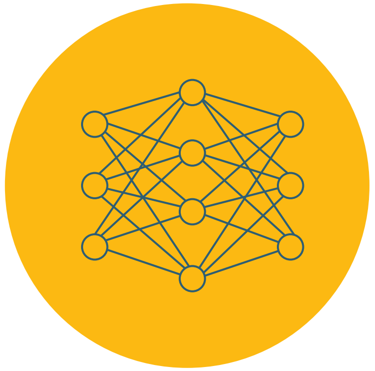
When building machine learning models, most of the design effort goes into preprocessing and cleaning up the input data, not writing the code of the model; neural networks and deep learning models are no exception to this rule.

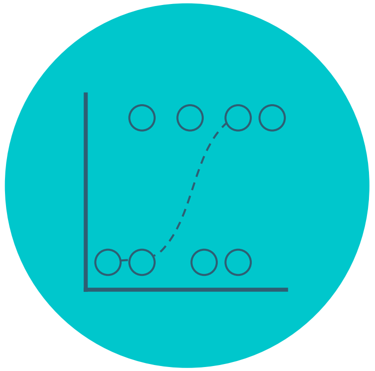
Getting Real with Neural Network Datasets

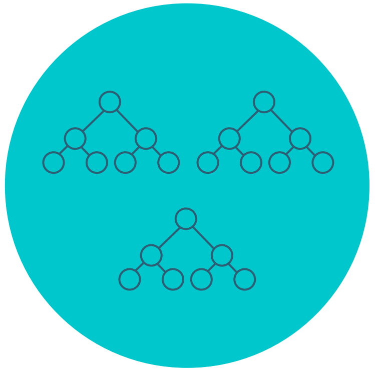
In general, neural network models require the most preprocessing of input data compared to other statistical and machine learning models.

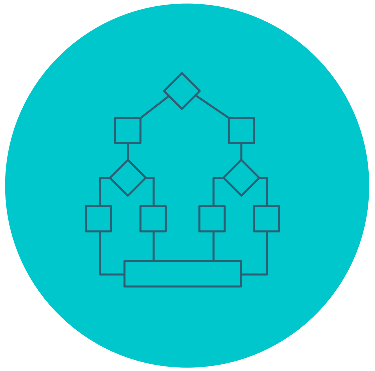
Most neural networks are really good at identifying patterns and trends in data, but they are susceptible to getting stuck when looking at abstract or raw data.

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.


Neural network


Logistic regression

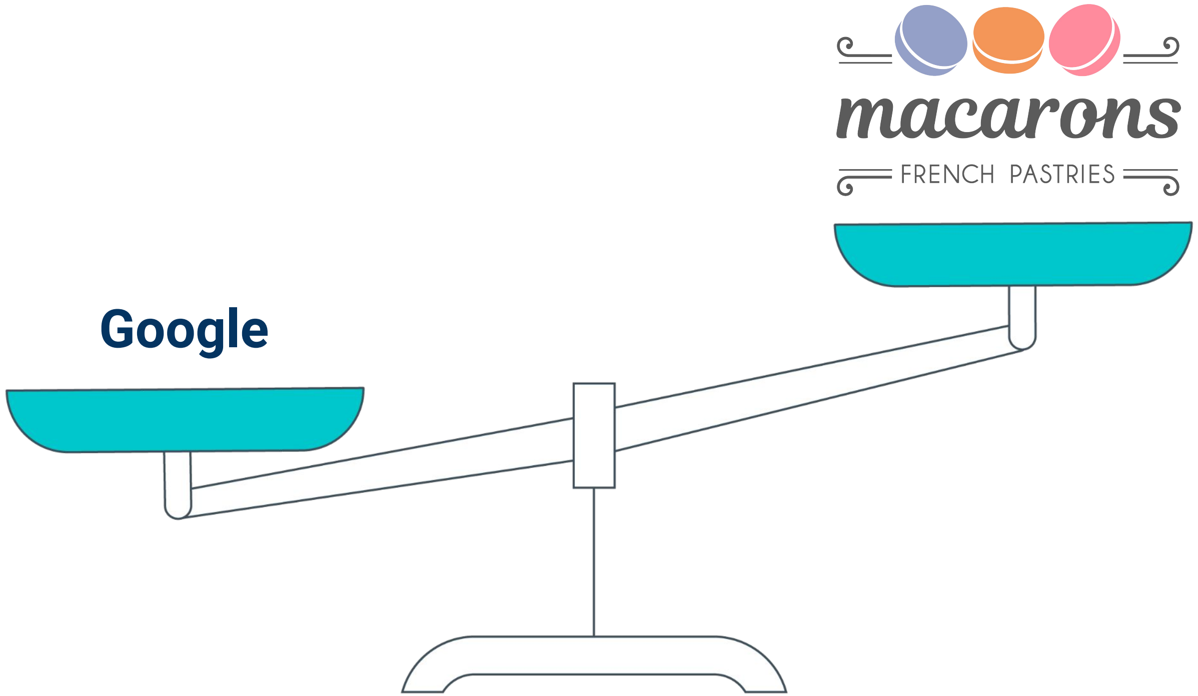

Random forest


Decision tree

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**.

Getting Real with Neural Network Datasets

If the bank's input dataset contained information from large Fortune 500 companies, such as Google and Facebook, as well as small family-owned stores, the variability in net worth would be huge.



Getting Real with Neural Network Datasets

Without **normalized input data**, a neural network could look at net worth as being a strong indicator of loan eligibility and might ignore all other factors, such as:

1. Debt-to-income ratio
2. Credit status
3. Requested loan amount

Getting Real with Neural Network Datasets

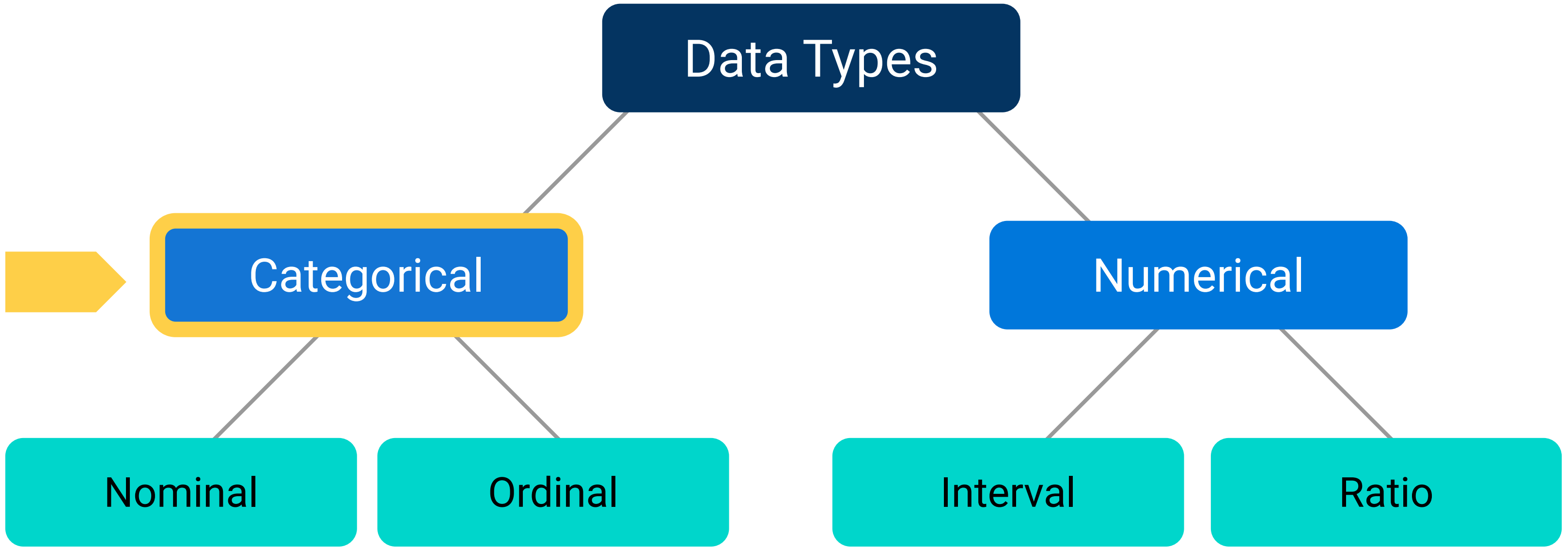
Instead, if the net worth were normalized on a factor such as number of employees, the neural network would be more likely to weigh other factors more evenly with net worth.

This would result in a neural network model that assesses loan eligibility more fairly, without introducing any additional risk.

Preprocessing the Input Data

We alter the input dataset before any computational model training or evaluation.

When working with a neural network, we must first preprocess the **categorical data** before we preprocess the numerical data.



Preprocessing the Input Data

We can preprocess our categorical data using the **OneHotEncoding method** from **scikit-learn**.

1. OneHotEncoding identifies all unique column values and then splits a single categorical column into a series of columns.

2. Each column contains information about a single unique categorical value.

Preprocessing the Input Data

Consider the following **`eye_color`** variable containing a list of eye colors from different people.

`Eye_Color`
Blue
Brown
Brown
Brown
Hazel
Green
Hazel
Brown
Blue
Brown

Preprocessing the Input Data

This table is the exact same **`eye_color`** variable encoded using OneHotEncoding.

1. Each row has only one column with a value of 1 — the corresponding categorical variable from the original dataset.
2. This binary encoding ensures that each neuron receives the same amount of information from the categorical variable.
3. Now the neural network will interpret each value independently and provide each categorical value its own weight in the algorithm.

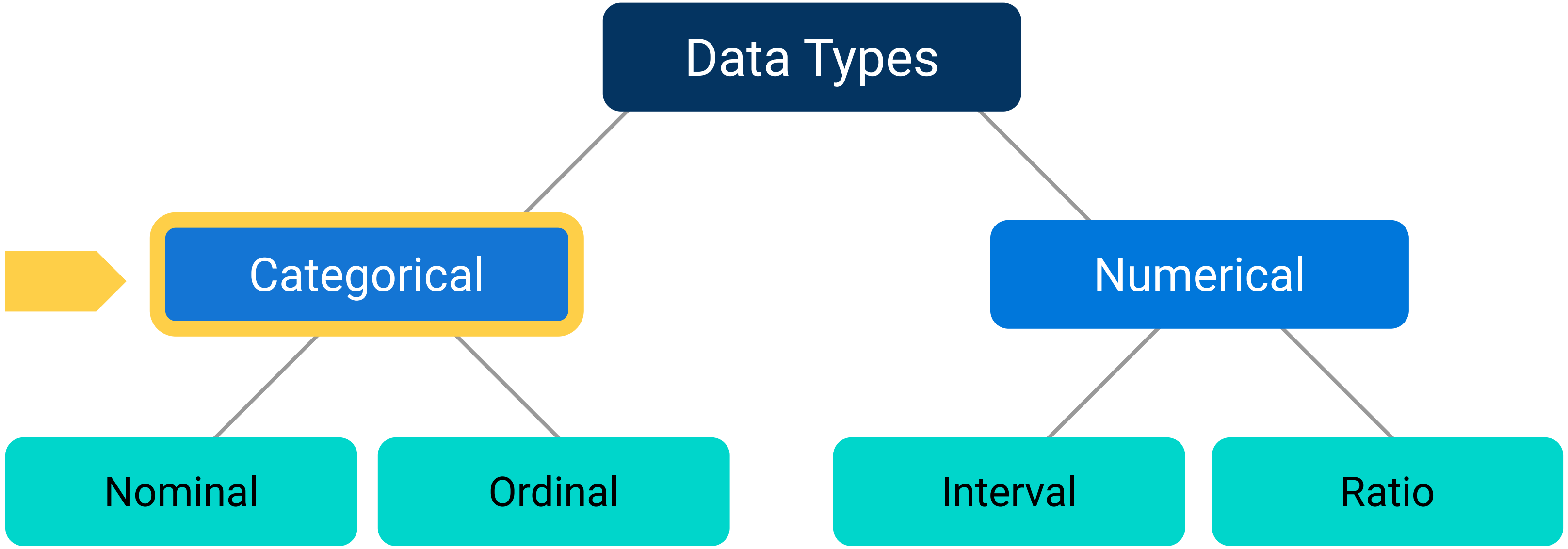
Eye_Color: Blue	Eye_Color: Brown	Eye_Color: Hazel	Eye_Color: Green
1	0	0	0
0	1	0	0
0	1	0	0
0	1	0	0
0	0	1	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0
0	1	0	0

OneHotEncoding is a very robust solution, but it can be **memory-intensive**.

Preprocessing the Input Data

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 the issue of memory resourcing, 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

There are two approaches to bucketing categorical data:

Approach #1

Collapse all of the infrequent and rare categorical values into a single "other" category.

This approach takes advantage of the fact that uncommon categories and "edge cases" are rarely statistically significant.

Approach #2

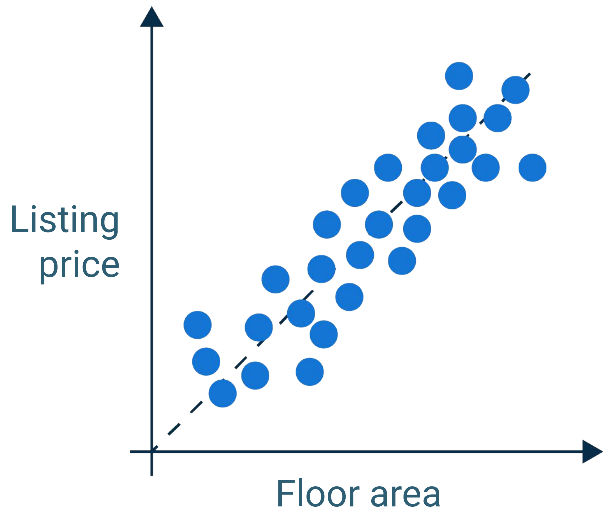
Create generalized categorical values and reassign all data points to the new corresponding values.

This approach collapses the number of unique categorical values and maintains relative order and magnitude.

This is particularly useful when we have categorical variables whose distribution of unique values is relatively even.

Bucketing

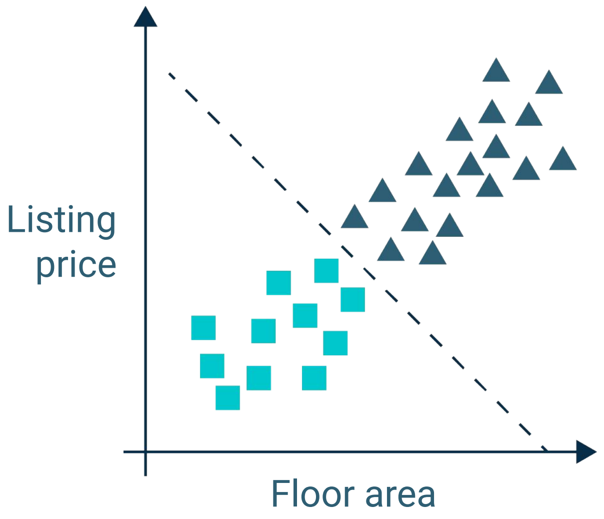
Regression and classification models typically cannot use rare categorical values to produce robust models.



Regression

● Housing

VS.



Classification

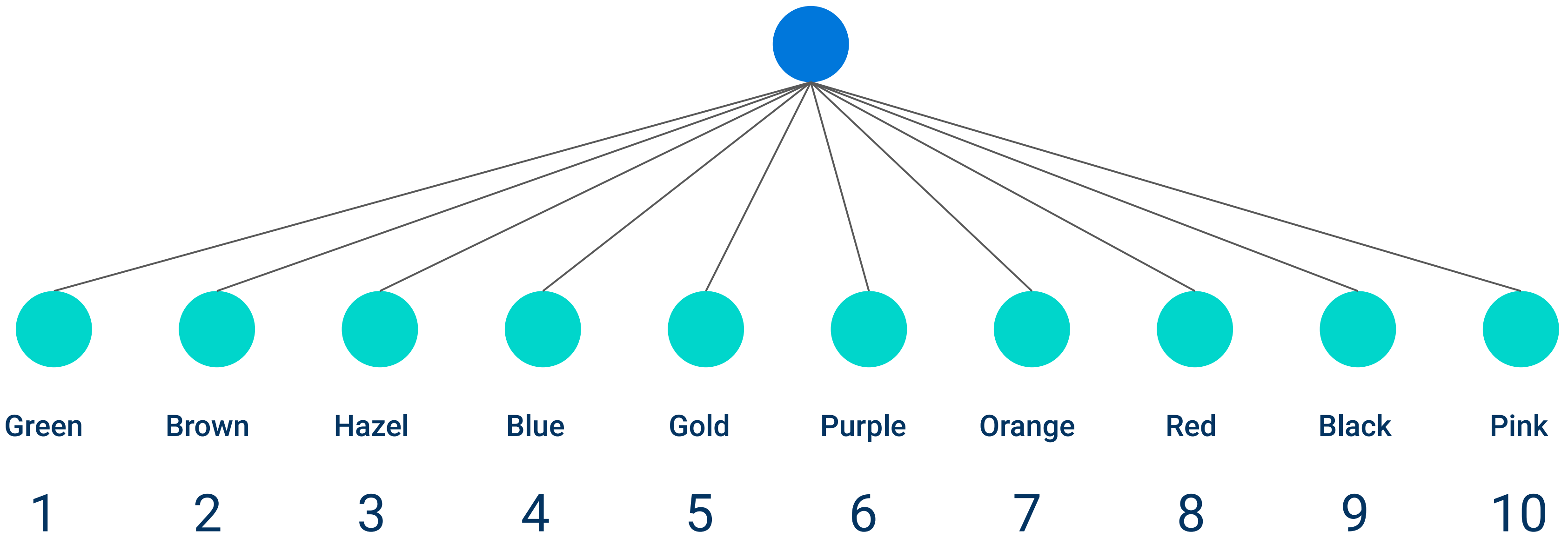
■ Apartments

▲ Houses

Bucketing

Bucketing is less effective when there are only a few unique values.

You should only apply a bucketing strategy when the categorical variables contain **10 or more unique values**.

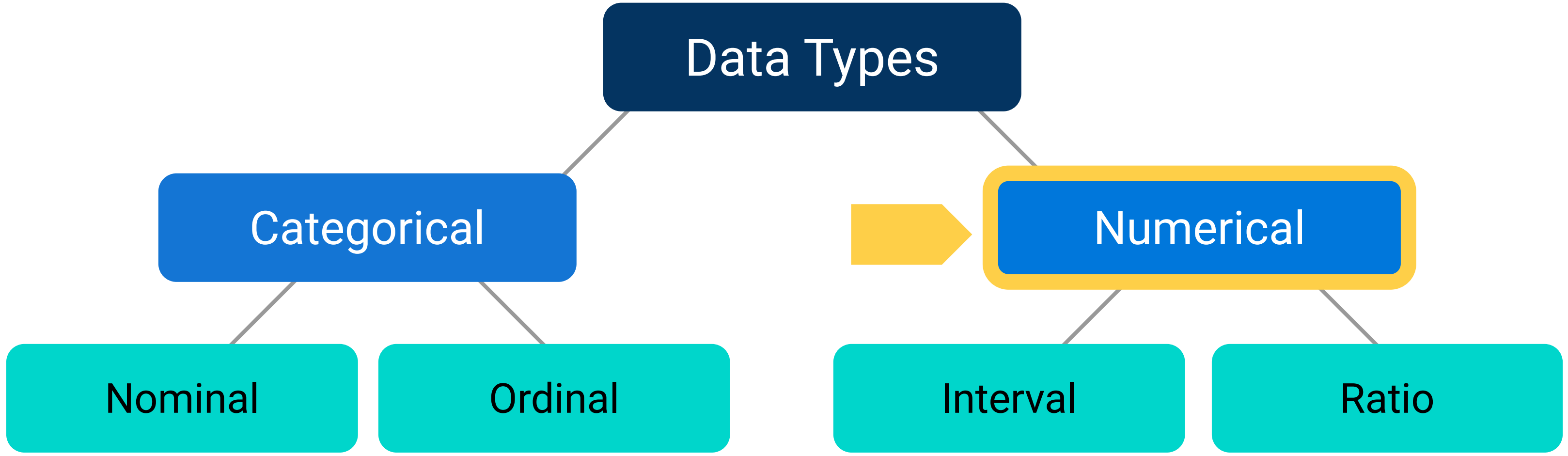


Preprocessing the Input Data

After we encode all categorical variables using OneHotEncoding, all the variables in our dataset will now be **numeric**.

1. Neural network models are not designed to handle categorical data, but they can interpret and evaluate all forms of numerical data.

2. Neural network models can train on raw numerical data, but training them on raw data often is not a good idea.



Preprocessing the Input Data

01

Raw data often has outliers or extreme values that can artificially inflate a variable’s importance.

02

Numerical data can be measured using different units across a dataset, such as time versus temperature or length versus volume.

03

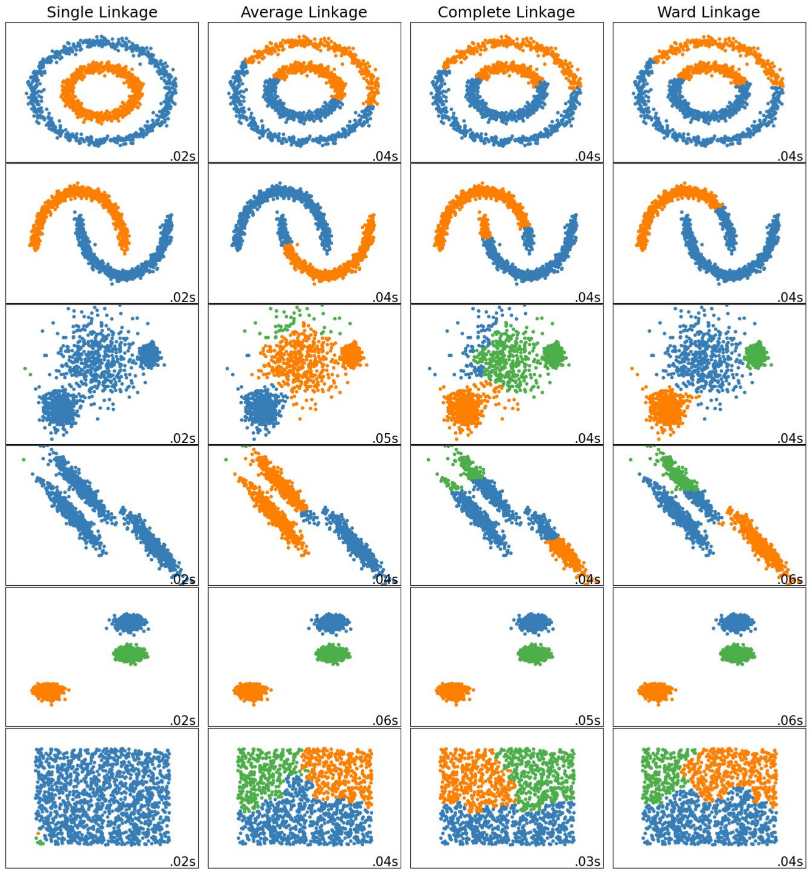
The distribution of a variable can be skewed, which results in misinterpretation of the central tendency.

Preprocessing the Input Data

The easiest way to minimize the risks associated with raw numerical data is to standardize the numerical data prior to training.

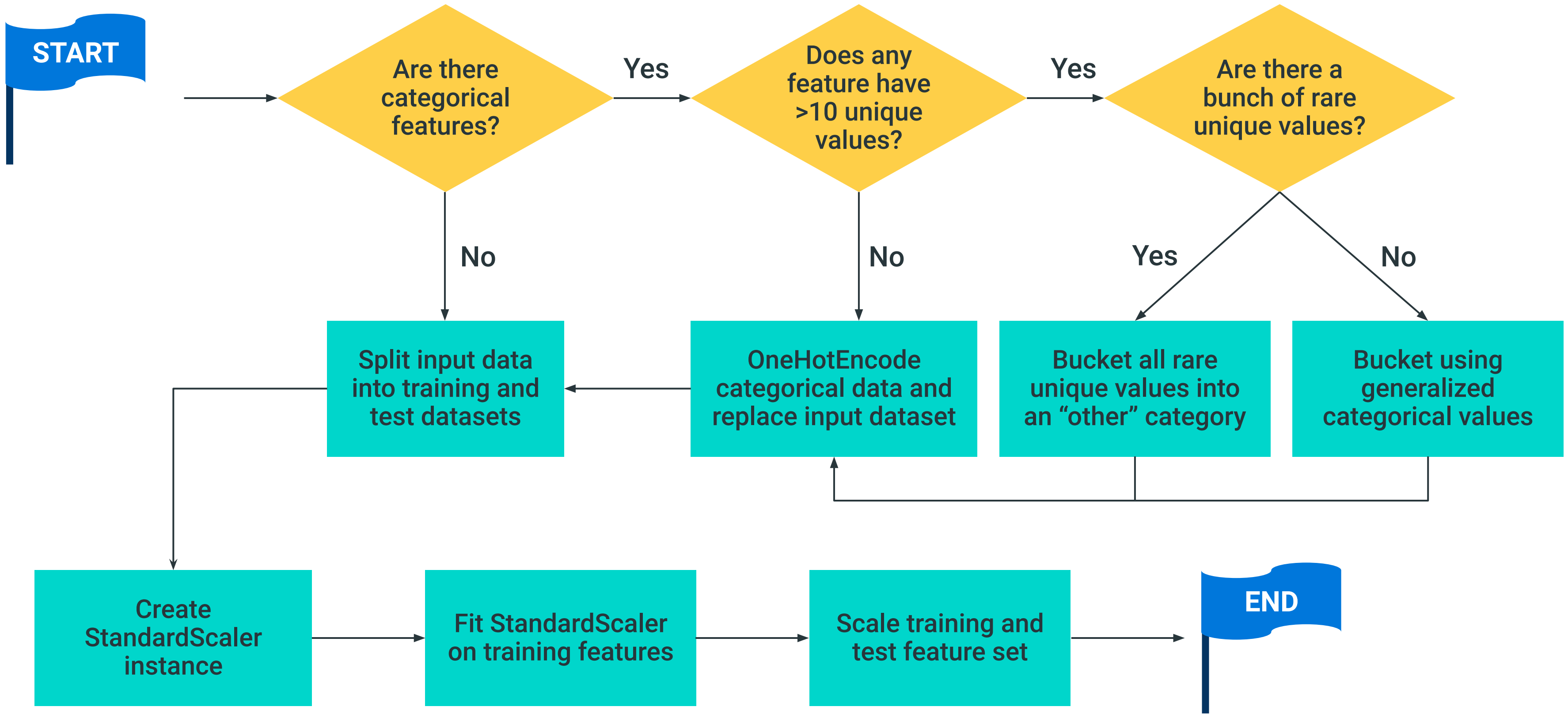
1. In Python, we can standardize the numerical data using scikit-learn's StandardScaler module.

2. By using StandardScaler 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.



Source

Preprocessing the Input Data


START
Are there categorical features?
Yes
Does any feature have >10 unique values?
Yes
Are there a bunch of rare unique values?
No
No
Yes
No
Split input data into training and test datasets
OneHotEncode categorical data and replace input dataset
Bucket all rare unique values into an “other” category
Bucket using generalized categorical values
Create StandardScaler instance
Fit StandardScaler on training features
Scale training and test feature set
END

Instructor **Demonstration**

Saving and Loading Neural Network Models

Training a complex neural network on a medium or large dataset can take hours or even days. For this reason, ML engineers must store and access trained models outside of the training environment.

Saving a Neural Network Model

Building a Keras model grants access to a **save** function, which saves these trained models in a **.keras** format.

The **save** function exports the following aspects of the model:

- The configuration of the model layers
- The weights associated with each layer
- The activation functions
- The optimizer
- The set of losses and metrics

Once the model has been saved, anyone can use the **`load_model`** function in Keras to import the exact same trained model into their environment. Then, they can use the model for analysis and predictions as if they had written it.

Activity:

Predicting Article Objectivity

In this activity, you will train a deep learning model to predict the objectivity of sports articles, then save and load the model.

Suggested Time:

20 Minutes

Time's up!

Let's review

Questions?

Let’s **recap**

Recap

After today’s lesson you are able to:

1. Implement deep neural network models using TensorFlow.
2. Explain how different neural network structures change algorithm performance.
3. Use KerasTuner to assist with finding optimal neural network structures.
4. Save trained TensorFlow models for later use.

Next

In the next lesson, you’ll learn …

Questions?

The End