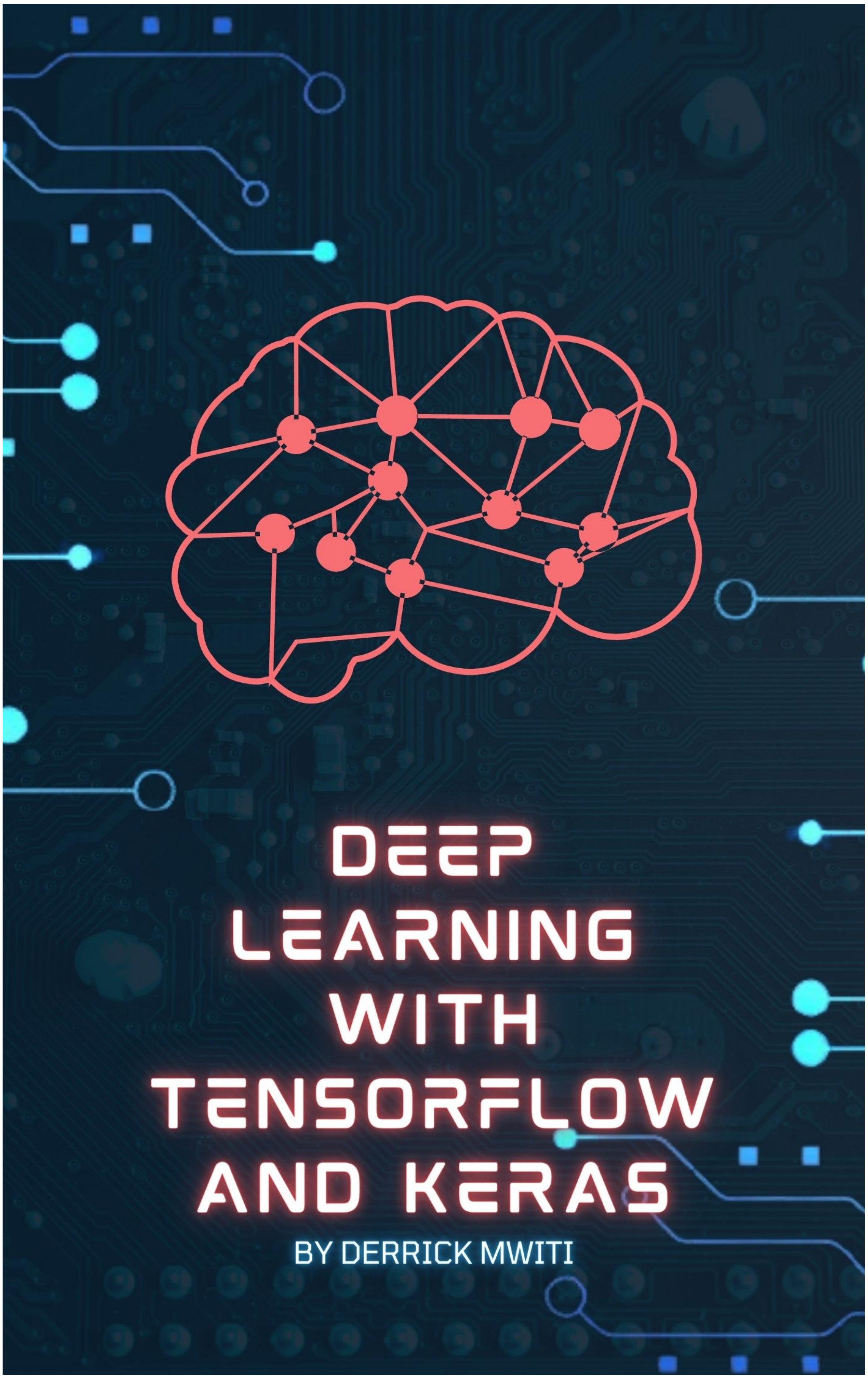
Deep learning sample


DEEP LEARNING WITH TENSORFLOW AND KERAS
BY DERRICK MWITI

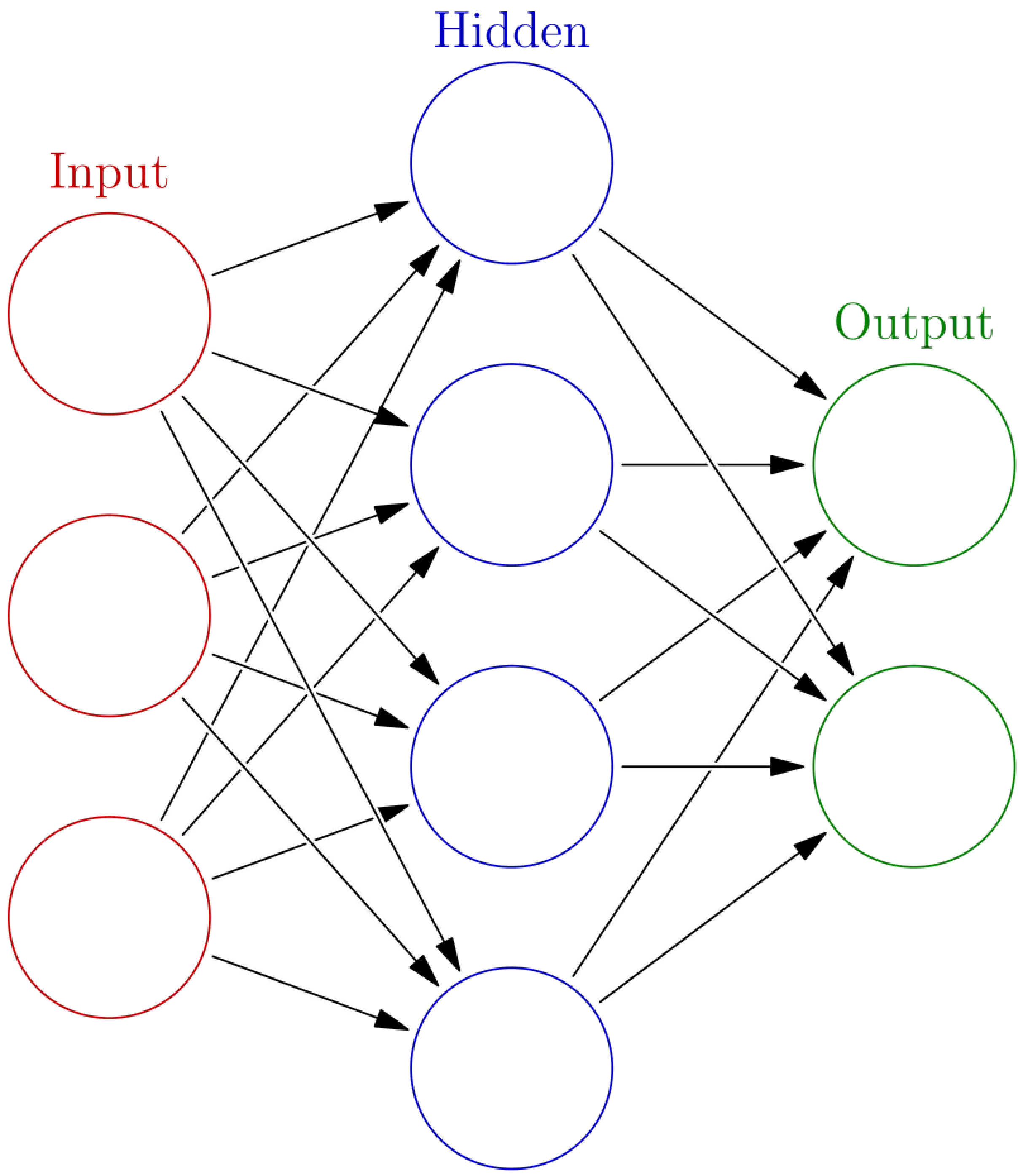
How to build artificial neural networks with Keras and TensorFlow

Building artificial neural networks with TensorFlow and Keras requires understanding some key concepts. After learning these concepts, you'll install TensorFlow and start designing neural networks. This article will cover the concepts you need to comprehend to build neural networks in TensorFlow and Keras. Without further ado, let's get the ball rolling.

What is deep learning?

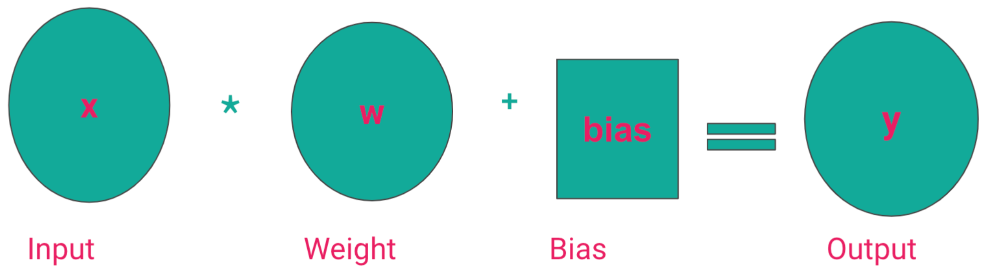
Deep learning is a branch of machine learning that involves building networks that try to mimic the working of the human brain. The dendrite in the human brain represents the input to the network, while the axion terminals represent the output. The cell is where computation would take place before we get the output.

The image below shows a simple network with an input, hidden, and output layer. A network with multiple hidden layers is called a **deep neural network**.



Random weights and biases are initialized when data is passed to a network. Some computation happens in the hidden layers leading to output.

This computation involves multiplying the input by the weights and adding the bias. This is what gives the output. The bias ensures that there is no zero output in case the input and weights are zero.



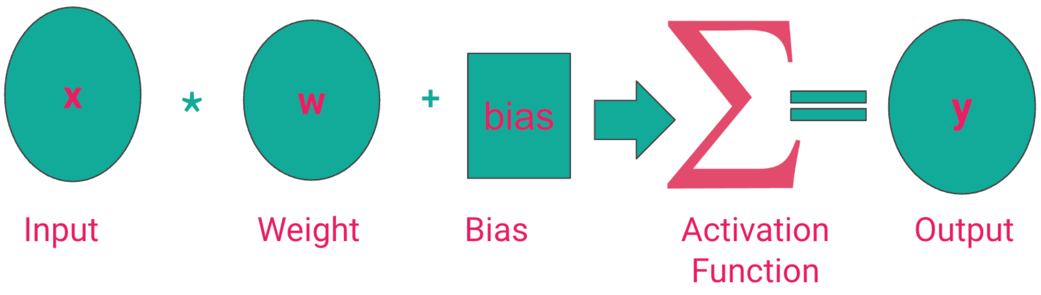
There are various ways of initializing the weights and biases. The common ones include:

- Initialize with ones.
- Initialize with zeros.

- Use a uniform distribution.
- Apply a normal distribution.

What is an activation function?

The desired output of a neural network depends on the problem being solved. For instance, in a regression problem, the output should be a number predicting the quantity in question. However, in classification problems, it's more desirable to output a probability that is used to determine the category of the prediction. Therefore, to make sure the network outputs the desired results, we pass the computed result through a function that ensures that the result is within a specific range. For example, for probabilities, this number is between 0 and 1. We can't have negative probabilities. The function responsible for capping the result is known as an **activation function**.



From the above image, we can conclude that the activation function determines the neural network's output. Let's, therefore, mention some of the common activation functions in the deep learning realm.

Sigmoid function

The **sigmoid activation function** caps output to a number between 0 and 1 and is majorly used for binary classification tasks. Sigmoid is used where the classes are non-exclusive. For example, an image can have a car, a building, a tree, etc. Just because there is a car in the image doesn’t mean a tree can’t be in the picture. Use the sigmoid function when there is more than one correct answer.

Softmax activation function

The **softmax activation function** is a variant of the sigmoid function used in multi-class problems where labels are mutually exclusive. For example, a picture is either grayscale or color. Use the softmax activation when there is only one correct answer.

Rectified linear unit (ReLU)

The **Rectified linear unit (ReLU) activation function** limits the output to 0 and above. It is used in the hidden layer of neural networks. It, therefore, ensures no negative outputs from the hidden layers.

How does a neural network learn?

A neural network learns by evaluating predictions against the true values and adjusting the weights. The objective is to obtain the weights that minimize the error, also known as the **loss function** or cost function. The choice of a loss function, therefore, depends on the problem. Classification tasks require classification loss functions, while regression problems require regression loss functions. As the network learns, the loss functions should decrease.

You might see nans in the loss function while training the network. This means that the network is not learning. In most cases, nans will be developer errors, meaning that there is something you have done or failed to do that is causing the nans. For example:

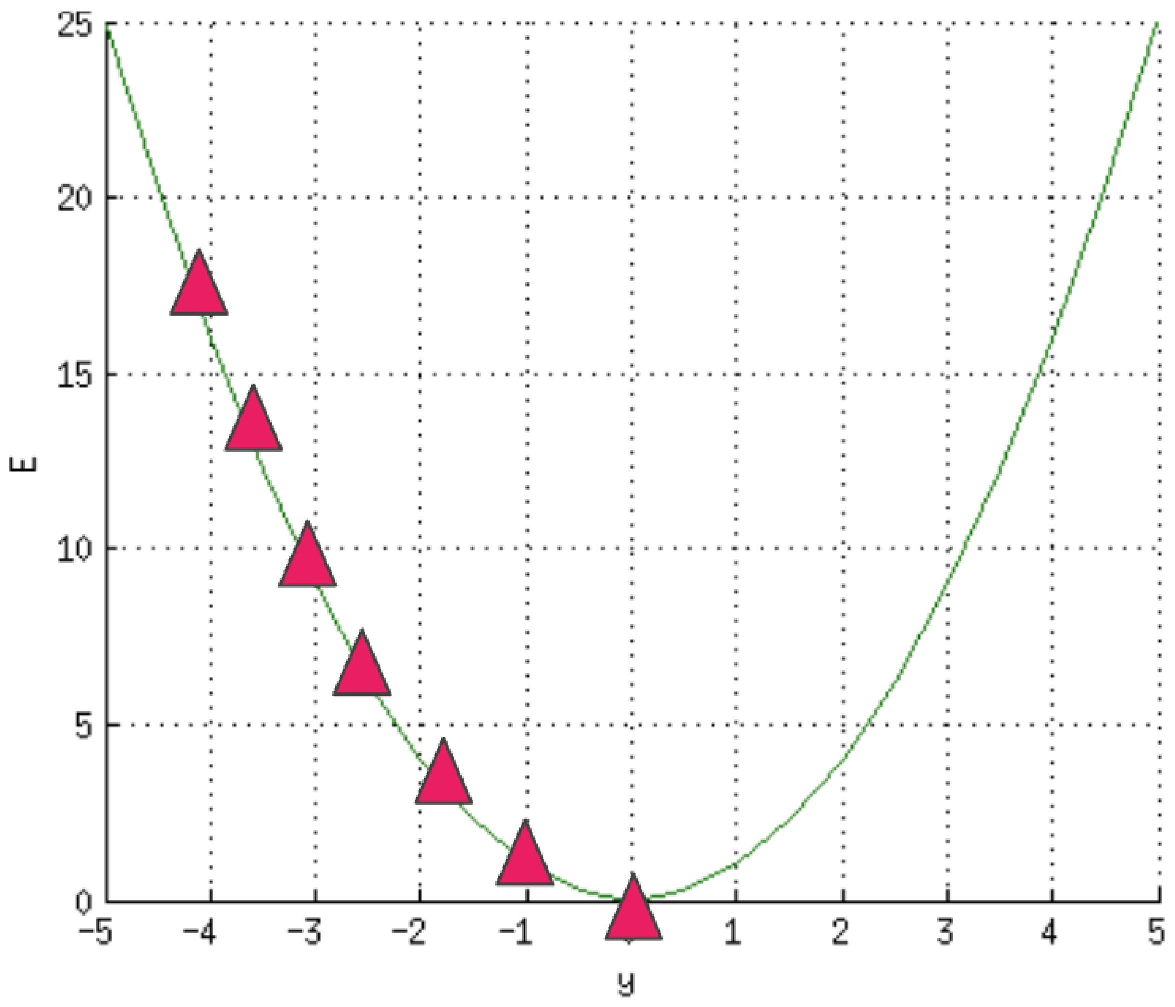
- The training data contains nans.

- You have not scaled the data.
- Performing operations that lead to nans, for example, division by zero or the square root of a negative number.
- Choosing the wrong optimizer function.

Gradient descent

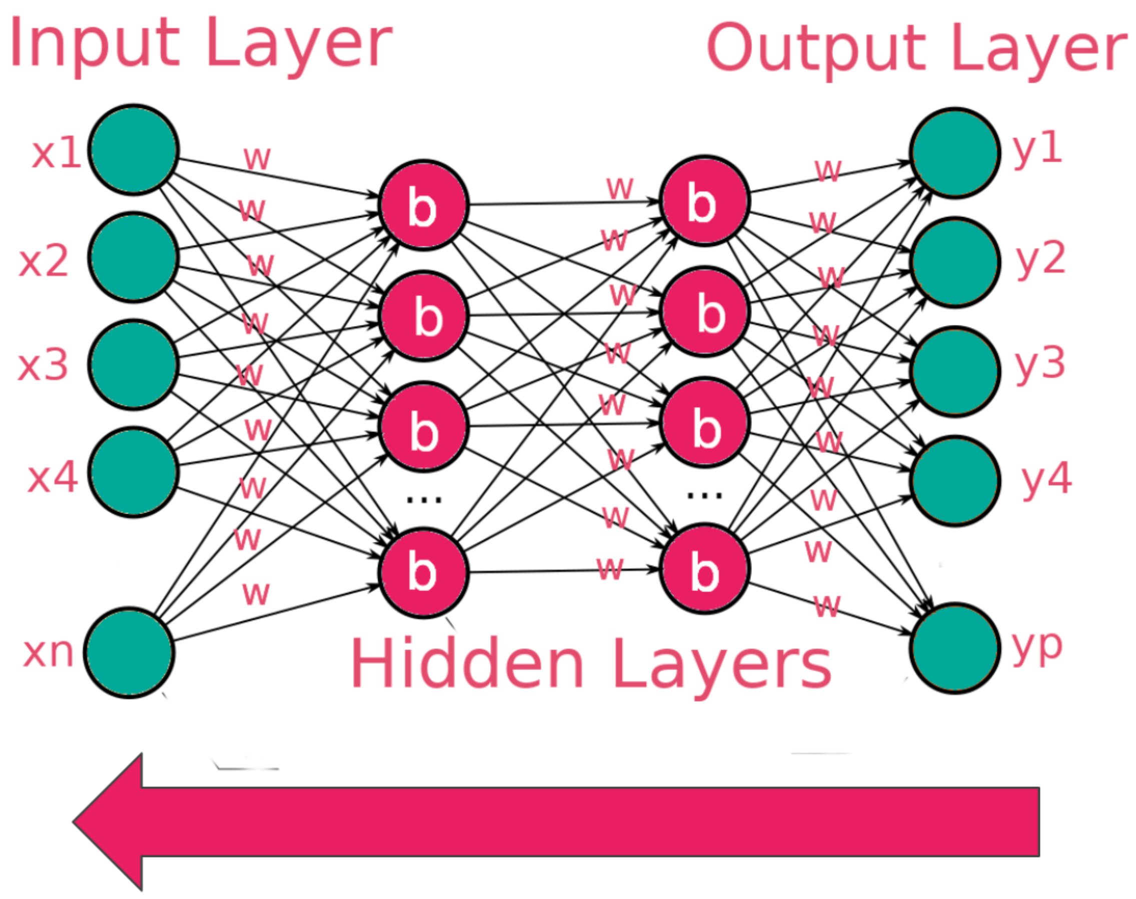
We have just mentioned that choosing the wrong optimizer could result in nans.
So **what is an optimizer function**?
During the training process, errors are reduced by the optimizer function. This optimization is done via gradient descent.

Gradient descent adjusts the errors by reducing the cost function. This is done by computing where the error is at its minimum, commonly known as the **local minimum**. You can think of this as descending on a slope where the goal is to get to the bottom of the hill, that is, the **global minimum**. This process involves computing the slope of a specific point on the "hill" via differentiation.



How backpropagation works

The computed errors are passed to the network, and the weights are adjusted. This process is known as **backpropagation**.



There are several variants of gradient descent. They include:

- **Batch Gradient Descent** that uses the entire dataset to compute the gradient of the cost function. It is slow since you have to compute the gradient of the entire dataset to perform a single update.

- **Stochastic Gradient Descent** where the gradient of the cost function is computed from a single training example in every iteration. It is faster.
- **Mini-Batch Gradient Descent** that uses a sample of the training data to compute the gradient of the cost function.

What is TensorFlow?

TensorFlow is an open-source deep learning framework that enables us to design and train deep learning networks. TensorFlow can be installed from the Python Index via the `pip` command. TensorFlow is already installed on Google Colab. You will, therefore, not install it when working in this environment.

```
# Requires the latest pip
pip install --upgrade pip

# Current stable release for CPU and GPU
pip install tensorflow

# Or try the preview build (unstable)
pip install tf-nightly
```

You can also install TensorFlow using Docker. Docker is the easiest way to install TensorFlow on Linux if GPU support is desired.

```
 docker pull tensorflow/tensorflow:latest
# Download latest stable image
 docker run -it -p 8888:8888 tensorflow/t
```

```
ensorflow:latest-jupyter  # Start Jupyter
server
```

Follow these instructions to install TensorFlow on Apple arm64 machines. This will enable you to train models with GPUs on Mac.

Why TensorFlow?

There are a couple of reasons why you would choose TensorFlow:

- Has a high-level API that makes it easy to build networks.
- Large ecosystem of tools and libraries.
- Large community that makes it easy to find solutions to common problems.

- Well documented.
- Supports deployment of models on the browser, mobile devices, edge devices, and the cloud.
- Simple and flexible architecture to make research work faster.

TensorFlow vs. Keras

As of TensorFlow 2, Keras is the high-level API for TensorFlow. Keras makes it simple to design and train deep learning networks.

TensorFlow basics

In a moment, we'll be designing neural networks with TensorFlow. However, getting some TensorFlow basics out of the way is essential before we get there.

Tensors

TensorFlow Tensors are multi-dimensional arrays similar to NumPy arrays. Tensors are immutable, meaning they can not be updated once created. Tensors can contain integers, floats, strings, and even complex numbers.

```
import tensorflow as tf
import numpy as np

x = tf.constant([[7., 8., 9.],[10., 11.,
 12.]])
print(x)
print(x.shape)
print(x.dtype)
# tf.Tensor(
# [[ 7.  8.  9.]
# [10. 11. 12.]], shape=(2, 3), dtype=flo
at32)
# (2, 3)
# <dtype: 'float32'>
```

You can perform indexing and operations on Tensors like in NumPy arrays.

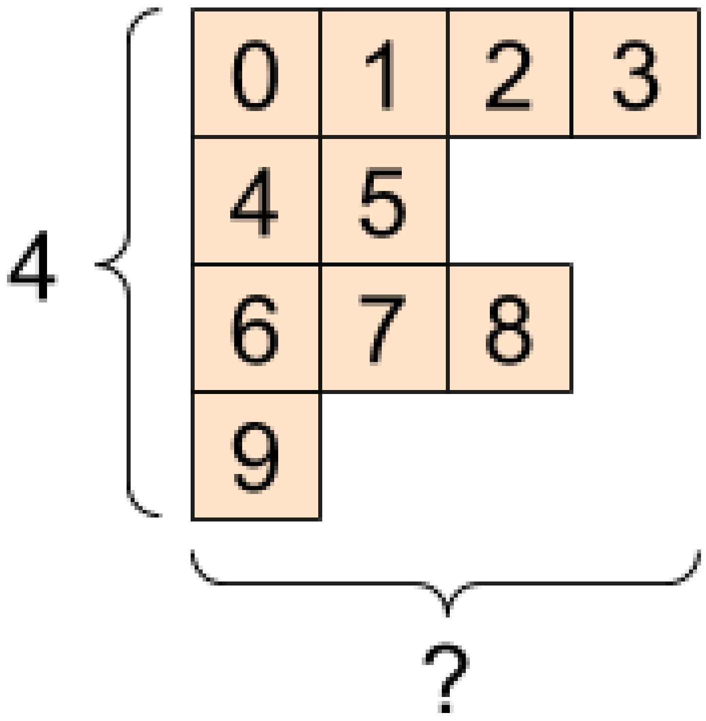
```
x[0]
# <tf.Tensor: shape=(3,), dtype=float32,
 numpy=array([7., 8., 9.],
# dtype=float32)>
x[1:4]
# <tf.Tensor: shape=(1, 3), dtype=float3
2, numpy=array([[10., 11., 12.]], # dtype
=float32)>
x**2
# <tf.Tensor: shape=(2, 3), dtype=float3
2, numpy=
# array([[ 49.,  64.,  81.],
#       [100., 121., 144.]], dtype=float3
2)>
```

```
x @ tf.transpose(x) # matrix multiplicati
on
```

TensorFlow Tensors can also be converted to NumPy arrays.

```
np.array(x)
x.numpy()
# array([[ 7.,  8.,  9.],
#        [10., 11., 12.]], dtype=float32)
```

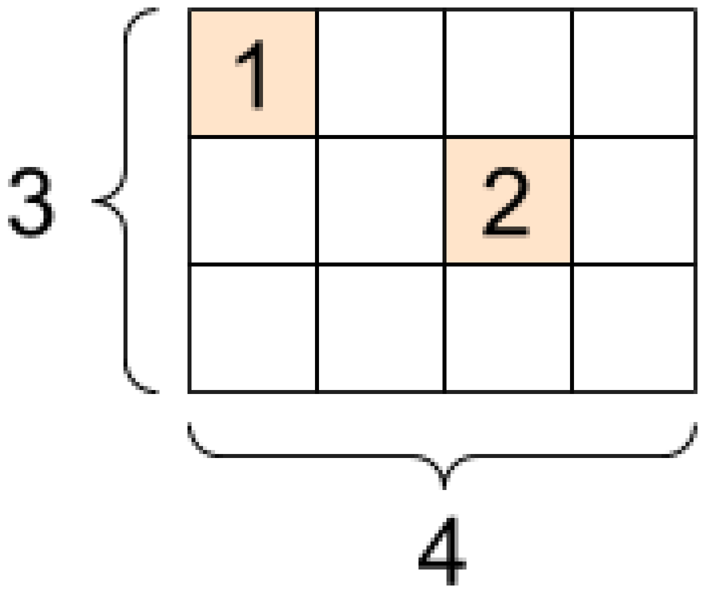
Tensors can contain a different number of elements along a certain axis. These kinds of tensors are known as **ragged tensors**.



Ragged tensors are created using the `tf.ragged.constant` function.

```
try:
  tensor = tf.constant(ragged_list)
except Exception as e:
  print(f"{type(e).__name__}: {e}")
  # ValueError: Can't convert non-rectang
ular Python sequence to Tensor.
ragged_tensor = tf.ragged.constant(ragged
_list)
print(ragged_tensor)
# <tf.RaggedTensor [[0, 1, 2, 3], [4, 5],
[6, 7, 8], [9]]>
print(ragged_tensor.shape)
# (4, None)
```

TensorFlow also supports tensors that have a lot of zeros. These tensors are known as **sparse tensors.** They can be created using the `tf.sparse.SparseTensor` function that stores them in a memory-efficient way. For example, converting text data to numerical representation in natural language processing tasks usually results in sparse tensors.



```
# Sparse tensors store values by index in
a memory-efficient manner
sparse_tensor = tf.sparse.SparseTensor(in
dices=[[0, 0], [1, 2]],
                                        va
lues=[1, 2],
                                        de
nse_shape=[3, 4])
print(sparse_tensor, "\n")
# SparseTensor(indices=tf.Tensor(
# [[0 0]
# [1 2]], shape=(2, 2), dtype=int64), val
ues=tf.Tensor([1 2], shape=(2,), # dtype=
int32), dense_shape=tf.Tensor([3 4], shap
e=(2,), dtype=int64))
```

A sparse tensor can also be converted to a dense tensor.

```
print(tf.sparse.to_dense(sparse_tensor))
# tf.Tensor(
#[[1 0 0 0]
# [0 0 2 0]
#[0 0 0 0]], shape=(3, 4), dtype=int32)
```

Variables

A **TensorFlow variable** is used to represent state in TensorFlow programs. Keras stores model parameters in a TensorFlow variable. A TensorFlow variable is also a tensor. A variable is created using `tf.Variable`.

```
my_tensor = tf.constant([[8.0, 8.0], [6.
0, 5.0]])
```

```
my_variable = tf.Variable(my_tensor)
my_variable
# <tf.Variable 'Variable:0' shape=(2, 2)
 dtype=float32, numpy=
# array([[8., 8.],
#       [6., 5.]], dtype=float32)>
```

We can check the type and shape of the TensorFlow variable because it's a tensor. It can also be converted to a NumPy array. Tensor operations can also be performed on variables, except that variables can not be reshaped. By default, TensorFlow places variables in the GPU to improve performance. You can, however, override this.

```
print("Shape: ", my_variable.shape)
print("DType: ", my_variable.dtype)
print("As NumPy: ", my_variable.numpy())
# Shape:  (2, 2)
# DType:  <dtype: 'float32'>
# As NumPy:  [[8. 8.]
# [6. 5.]]
```

Automatic differentiation

Automatic differentiation is applied at the backpropagation stage of training neural networks. Automatic differentiation in TensorFlow is done using `tf.GradientTape`. Inputs to this function are usually `tf.variables`.

```
x = tf.Variable(47.0)
with tf.GradientTape() as tape:
  y = x**2
# dy = 2x * dx
dy_dx = tape.gradient(y, x)
dy_dx.numpy()
# 94.0
```

You can use `GradientTape` to define custom training functions.
The `GradientTape`

will track all trainable variables automatically. `tf.gradients` computes the gradients with respect to the weights and biases.

```
# Given a callable model, inputs, output
s, and a learning rate...
def train(model, x, y, learning_rate):

  with tf.GradientTape() as t:
    # Trainable variables are automatical
ly tracked by GradientTape
    current_loss = loss(y, model(x))

  # Use GradientTape to calculate the gra
dients with respect to W and b
  dw, db = t.gradients(current_loss, [mod
el.w, model.b])

  # Subtract the gradient scaled by the l
earning rate
```

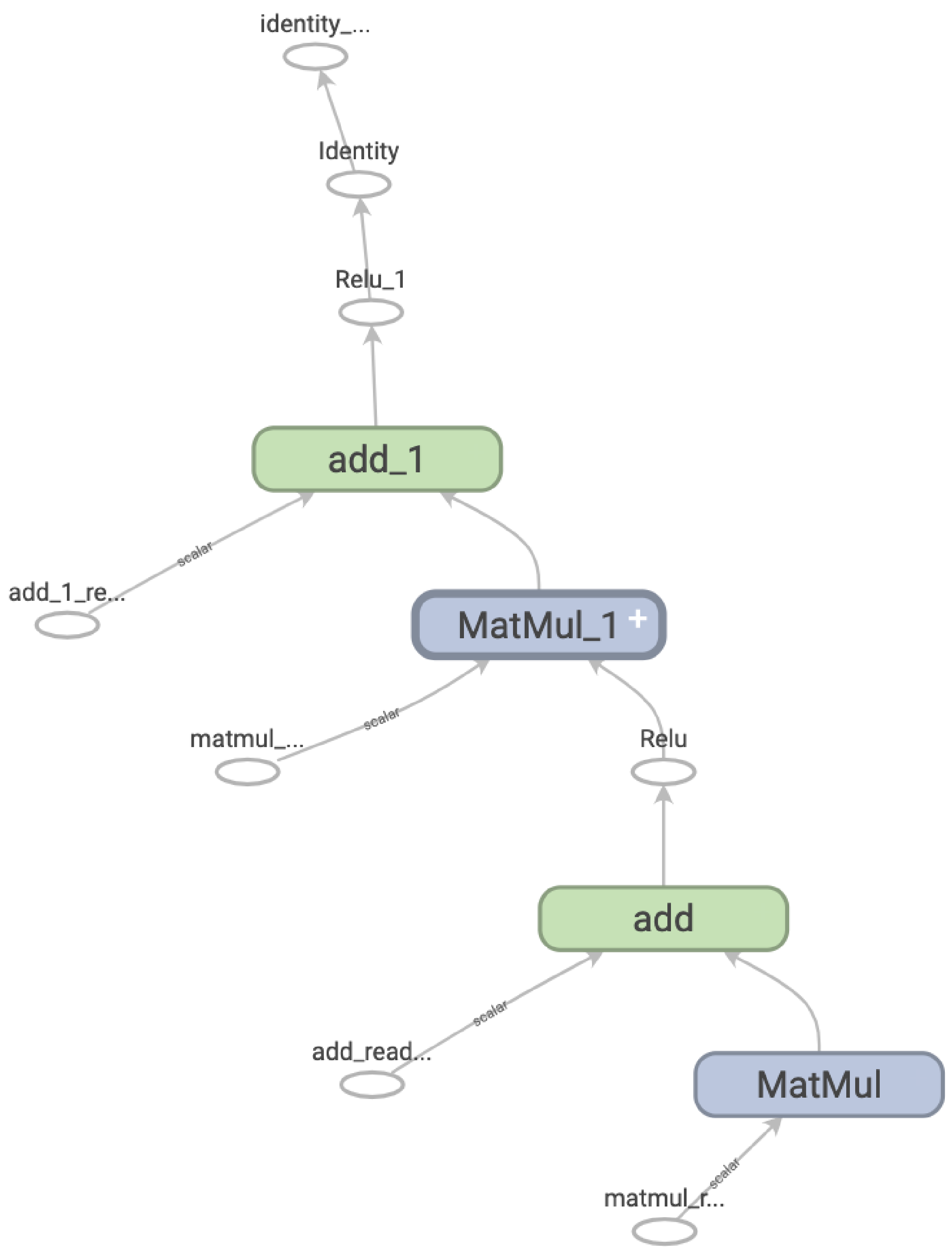
```
    model.w.assign_sub(learning_rate * dw)
    model.b.assign_sub(learning_rate * db)
```

How TensorFlow works

When TensorFlow is executed ***eagerly,*** operations are done in Python, and the results are sent back to Python. The alternative is graph execution, where the operations are executed as a **TensorFlow Graph**. A Graph represents a data structure representing a set of operations.

The fact that graphs are data structures makes it possible to save and restore them without the original Python code. As a result, these graphs can be used in non-Python environments such as **mobile devices, servers, edge devices, and**

embedded devices. Saved models in TensorFlow are exported as graphs. Graphs enable TensorFlow to run on multiple devices, run in parallel and be fast.



TensorFlow graph representing a two-layer neural network

Creating a TensorFlow graph is done via `tf.function`. It expects a normal function and returns a callable function which creates the TensorFlow graph from the Python function.

```
def simple_relu(x):
  if tf.greater(x, 0):
    return x
  else:
    return 0

# `tf_simple_relu` is a TensorFlow `Funct
ion` that wraps `simple_relu`.
tf_simple_relu = tf.function(simple_relu)

print("First branch, with graph:", tf_sim
ple_relu(tf.constant(1)).numpy())
print("Second branch, with graph:", tf_si
mple_relu(tf.constant(-1)).numpy())
# First branch, with graph: 1
# Second branch, with graph: 0
```

How TensorFlow models are defined

A TensorFlow model is made up of layers. Certain mathematical computations occur in the layers. Layers have trainable variables, meaning that these variables are updated as the network is fitted to the training data. In TensorFlow, layers and models are built by creating a class that inherits `tf.Module`.

```
class SimpleModule(tf.Module):
  def __init__(self, name=None):
    super().__init__(name=name)
    self.a_variable = tf.Variable(5.0, na
me="train_me")
    self.non_trainable_variable = tf.Vari
able(5.0, trainable=False, name="do_not_t
rain_me")
  def __call__(self, x):
    return self.a_variable * x + self.non
_trainable_variable
```

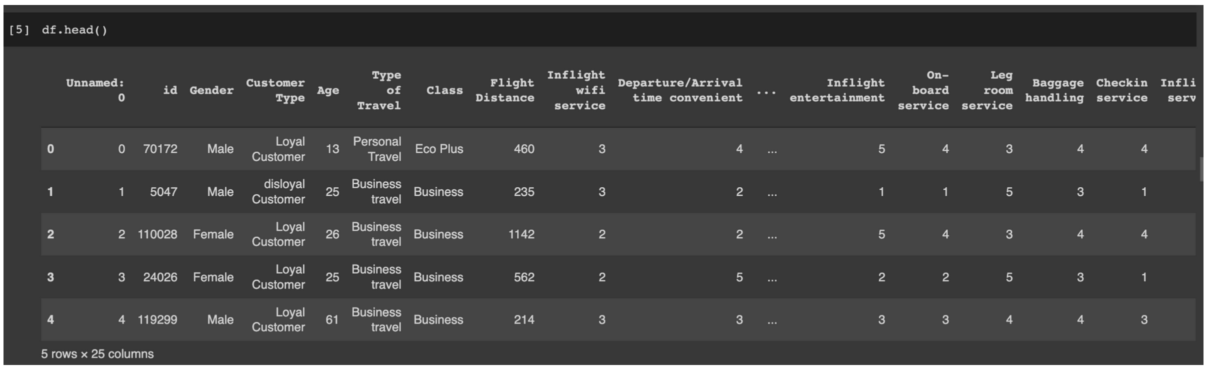
```
simple_module = SimpleModule(name="simpl
e")

simple_module(tf.constant(5.0))
# <tf.Tensor: shape=(), dtype=float32, nu
mpy=30.0>
```

How to train artificial neural networks with Keras

With the basics out of the way, we can now embark on a journey to learn how to design and train artificial neural networks in TensorFlow and Keras. We'll use a dataset from Kaggle to train a classification network. The aim is to predict the satisfaction of an airline passenger. Let's start by printing a sample of this dataset.

```
import pandas as pd
df = pd.read_csv("train.csv")
df.head()
```



```
[5] df.head()
```

	Unnamed: 0	id	Gender	Customer Type	Age	Type of Travel	Class	Flight Distance	Inflight wifi service	Departure/Arrival time convenient	...	Inflight entertainment	On-board service	Leg room service	Baggage handling	Checkin service	Infli serv
0	0	70172	Male	Loyal Customer	13	Personal Travel	Eco Plus	460	3	4	...	5	4	3	4	4	
1	1	5047	Male	disloyal Customer	25	Business travel	Business	235	3	2	...	1	1	5	3	1	
2	2	110028	Female	Loyal Customer	26	Business travel	Business	1142	2	2	...	5	4	3	4	4	
3	3	24026	Female	Loyal Customer	25	Business travel	Business	562	2	5	...	2	2	5	3	1	
4	4	119299	Male	Loyal Customer	61	Business travel	Business	214	3	3	...	3	3	4	4	3	

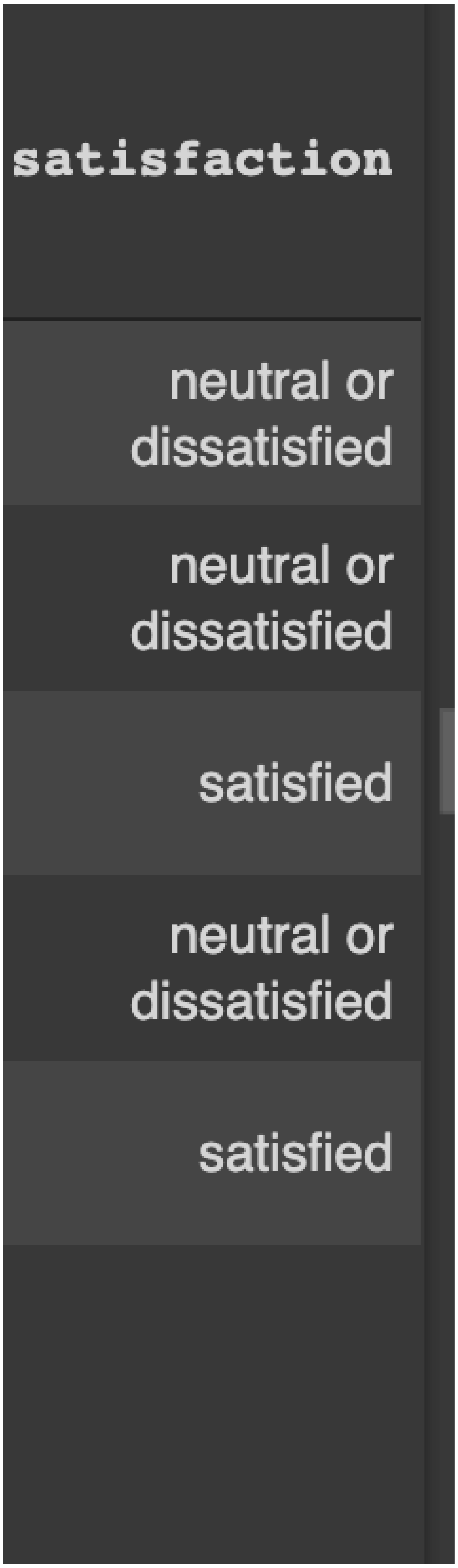
5 rows × 25 columns

Data pre-processing

Ensure the data is clean before passing it to the neural network. For instance, we need to check for null values and deal with them. The occurrence of null values in the training data leads to nans in the training loss. The `Arrival Delay in Minutes` column has null values. There are various ways of dealing with null values, but in this case, we'll replace them with the mean of the column.

```
df['Arrival Delay in Minutes'] = df['Arri
val Delay in Minutes'].mean()
```

The target column is a string.



satisfaction
neutral or dissatisfied
neutral or dissatisfied
satisfied
neutral or dissatisfied
satisfied

We can't pass strings to the neural network. Convert this column to a numerical representation. This can be done using Scikit-learn's label encoder.

```
from sklearn.preprocessing import LabelEn
coder
labelencoder = LabelEncoder()
df = df.assign(satisfaction = labelencode
r.fit_transform(df["satisfaction"]))
```

Data transformation

Apart from the target column, other columns are also in text form. They need to be converted to a numerical format.

```
categories = df.select_dtypes(include=['o
bject']).columns.tolist()
categories
```

```
#['Gender', 'Customer Type', 'Type of Tra
vel', 'Class']
```

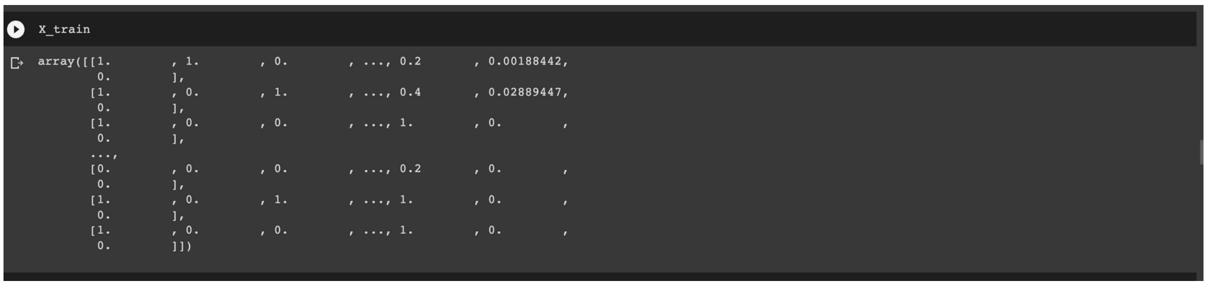
Another thing we need to do is to scale the dataset. The weights and biases of neural networks are initialized to small numbers, usually between 0 and 1. Scaling makes training easier by forcing all values to be within a certain range. Failure to scale can lead to nans in the training loss due to the large magnitude between training values. After doing all this, we'll split the data into a training and testing set.

Let's use the ColumnTransformer from Scikit-learn to apply the transformations we have mentioned above. The transformer enables us to apply more than one transformation to multiple columns. In this case, we apply the following steps:

- Transform categorical columns to numerical form via one-hot encoding.
- Scale numerical columns using `MinMaxScaler` to ensure that all values are between 0 and 1.

After obtaining the transformed data, we split it into a training and testing set.

```
from sklearn.model_selection import train
_test_split
from sklearn.preprocessing import MinMaxS
caler
X = df.drop(["Unnamed: 0", "id","satisfac
tion"], axis=1)
y = df["satisfaction"]
random_state = 13
test_size = 0.3
transformer = ColumnTransformer(transform
ers=[('cat', OneHotEncoder(handle_unknown
='ignore', drop="first"), categories)],re
mainder=MinMaxScaler())
X = transformer.fit_transform(X)
X_train, X_test, y_train, y_test = train_
test_split(X, y, test_size=test_size,rand
om_state=random_state)
```



```
X_train
```

```
array([[1.        , 1.        , 0.        , ..., 0.2       , 0.00188442,
         0.        ],
       [1.        , 0.        , 1.        , ..., 0.4       , 0.02889447,
         0.        ],
       [1.        , 0.        , 0.        , ..., 1.        , 0.        ,
         0.        ],
       ...,
       [0.        , 0.        , 0.        , ..., 0.2       , 0.        ,
         0.        ],
       [1.        , 0.        , 1.        , ..., 1.        , 0.        ,
         0.        ],
       [1.        , 0.        , 0.        , ..., 1.        , 0.        ,
         0.        ]])
```

The transformed data

How to build the artificial neural network

Keras makes it easy to design neural networks via the Sequential API. We can stack the layers we want in our network using this API. In this case, let's define a network with the following layers:

- **Input layer** with the number of units similar to the number of features in the training data.
- Two **dense layers**. We randomly define the units in these layers, but we'll look at how to select the best units later.
- A **final dense** with 1 unit and the **sigmoid activation** function because it's a binary classification problem.

```
model = Sequential(
    [
        Input(shape=(X_train.shape[1],)),
        Dense(64, activation="relu",  ker
nel_initializer="glorot_uniform",name="la
yer1"),
        Dense(32, activation="relu", kern
el_initializer="glorot_uniform", name="la
yer2"),
        Dense(1, activation="sigmoid", na
me="layer3"),
    ]
)
```

Get full book at https://www.mlnuggets.com/deep-learning-with-tensorflow-and-keras

Copyright

Other things to learn

Learn Python

Learn data science

Learn Streamlit