Open in Colab

00. Getting started with TensorFlow: A guide to the fundamentals

What is TensorFlow?

TensorFlow is an open-source end-to-end machine learning library for preprocessing data, modelling data and serving models (getting them into the hands of others).

Why use TensorFlow?

Rather than building machine learning and deep learning models from scratch, it's more likely you'll use a library such as TensorFlow. This is because it contains many of the most common machine learning functions you'll want to use.

What we're going to cover

TensorFlow is vast. But the main premise is simple: turn data into numbers (tensors) and build machine learning algorithms to find patterns in them.

In this notebook we cover some of the most fundamental TensorFlow operations, more specificially:

- Introduction to tensors (creating tensors)
- Getting information from tensors (tensor attributes)
- Manipulating tensors (tensor operations)
- Tensors and NumPy
- Using @tf.function (a way to speed up your regular Python functions)
- Using GPUs with TensorFlow
- Exercises to try

Things to note:

- Many of the conventions here will happen automatically behind the scenes (when you build a model) but it's worth knowing so if you see any of these things, you know what's happening.
- For any TensorFlow function you see, it's important to be able to check it out in the documentation, for example, going to the Python API docs for all functions and searching for what you need: https://www.tensorflow.org/api_docs/python/ (don't worry if this seems overwhelming at first, with enough practice, you'll get used to navigating the documentaiton).

```
# Create timestamp
import datetime

print(f"Notebook last run (end-to-end): {datetime.datetime.now()}")
```

```
Notebook last run (end-to-end): 2023-04-25 05:22:53.455288
```

Introduction to Tensors

If you've ever used NumPy, tensors are kind of like NumPy arrays (we'll see more on this later).

For the sake of this notebook and going forward, you can think of a tensor as a multi-dimensional numerical representation (also referred to as n-dimensional, where n can be any number) of something. Where something can be almost anything you can imagine:

- It could be numbers themselves (using tensors to represent the price of houses).
- It could be an image (using tensors to represent the pixels of an image).
- It could be text (using tensors to represent words).
- Or it could be some other form of information (or data) you want to represent with numbers.

The main difference between tensors and NumPy arrays (also an n-dimensional array of numbers) is that tensors can be used on GPUs (graphical processing units) and TPUs (tensor processing units).

The benefit of being able to run on GPUs and TPUs is faster computation, this means, if we wanted to find patterns in the numerical representations of our data, we can generally find them faster using GPUs and TPUs.

Okay, we've been talking enough about tensors, let's see them.

The first thing we'll do is import TensorFlow under the common alias `tf`.

```
# Import TensorFlow
import tensorflow as tf
print(tf.__version__) # find the version number (should be 2.x+)
```

```
2.12.0
```

Creating Tensors with `tf.constant()`

As mentioned before, in general, you usually won't create tensors yourself. This is because TensorFlow has modules built-in (such as `tf.io` and `tf.data`) which are able to read your data sources and automatically convert them to tensors and then later on, neural network models will process these for us.

But for now, because we're getting familar with tensors themselves and how to manipulate them, we'll see how we can create them ourselves.

We'll begin by using `tf.constant()`.

```
# Create a scalar (rank 0 tensor)
scalar = tf.constant(7)
scalar
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=7>
```

A scalar is known as a rank 0 tensor. Because it has no dimensions (it's just a number).

> 🔑 **Note:** For now, you don't need to know too much about the different ranks of tensors (but we will see more on this later). The important point is knowing tensors can have an unlimited range of dimensions (the exact amount will depend on what data you're representing).

```
# Check the number of dimensions of a tensor (ndim stands for number of dimensions)
scalar.ndim
```

```
0
```

```
# Create a vector (more than 0 dimensions)
vector = tf.constant([10, 10])
vector
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([10, 10], dtype=int32)>
```

```
# Check the number of dimensions of our vector tensor
vector.ndim
```

```
1
```

```
# Create a matrix (more than 1 dimension)
matrix = tf.constant([[10, 7],
                      [7, 10]])
matrix
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[10,  7],
       [ 7, 10]], dtype=int32)>
```

```
matrix.ndim
```

```
2
```

By default, TensorFlow creates tensors with either an `int32` or `float32` datatype.

This is known as 32-bit precision (the higher the number, the more precise the number, the more space it takes up on your computer).

```
# Create another matrix and define the datatype
another_matrix = tf.constant([[10., 7.],
                              [3., 2.],
                              [8., 9.]], dtype=tf.float16) # specify the datatype with 'dt
another_matrix
```

```
<tf.Tensor: shape=(3, 2), dtype=float16, numpy=
array([[10.,  7.],
       [ 3.,  2.],
       [ 8.,  9.]], dtype=float16)>
```

```
# Even though another_matrix contains more numbers, its dimensions stay the same
another_matrix.ndim
```

```
2
```

```
# How about a tensor? (more than 2 dimensions, although, all of the above items are also t
tensor = tf.constant([[[1, 2, 3],
                       [4, 5, 6]],
                      [[7, 8, 9],
                       [10, 11, 12]],
                      [[13, 14, 15],
                       [16, 17, 18]],
                      [[19,20,21],[22,23,24]],
                      [[25,26,27],[28,29,30]]])
tensor
```

```
<tf.Tensor: shape=(5, 2, 3), dtype=int32, numpy=
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]],

       [[13, 14, 15],
        [16, 17, 18]],

       [[19, 20, 21],
        [22, 23, 24]],

       [[25, 26, 27],
        [28, 29, 30]]], dtype=int32)>
```

```
tensor.ndim
```

```
3
```

This is known as a rank 3 tensor (3-dimensions), however a tensor can have an arbitrary (unlimited) amount of dimensions.

For example, you might turn a series of images into tensors with shape (224, 224, 3, 32), where:

- 224, 224 (the first 2 dimensions) are the height and width of the images in pixels.
- 3 is the number of colour channels of the image (red, green blue).
- 32 is the batch size (the number of images a neural network sees at any one time).

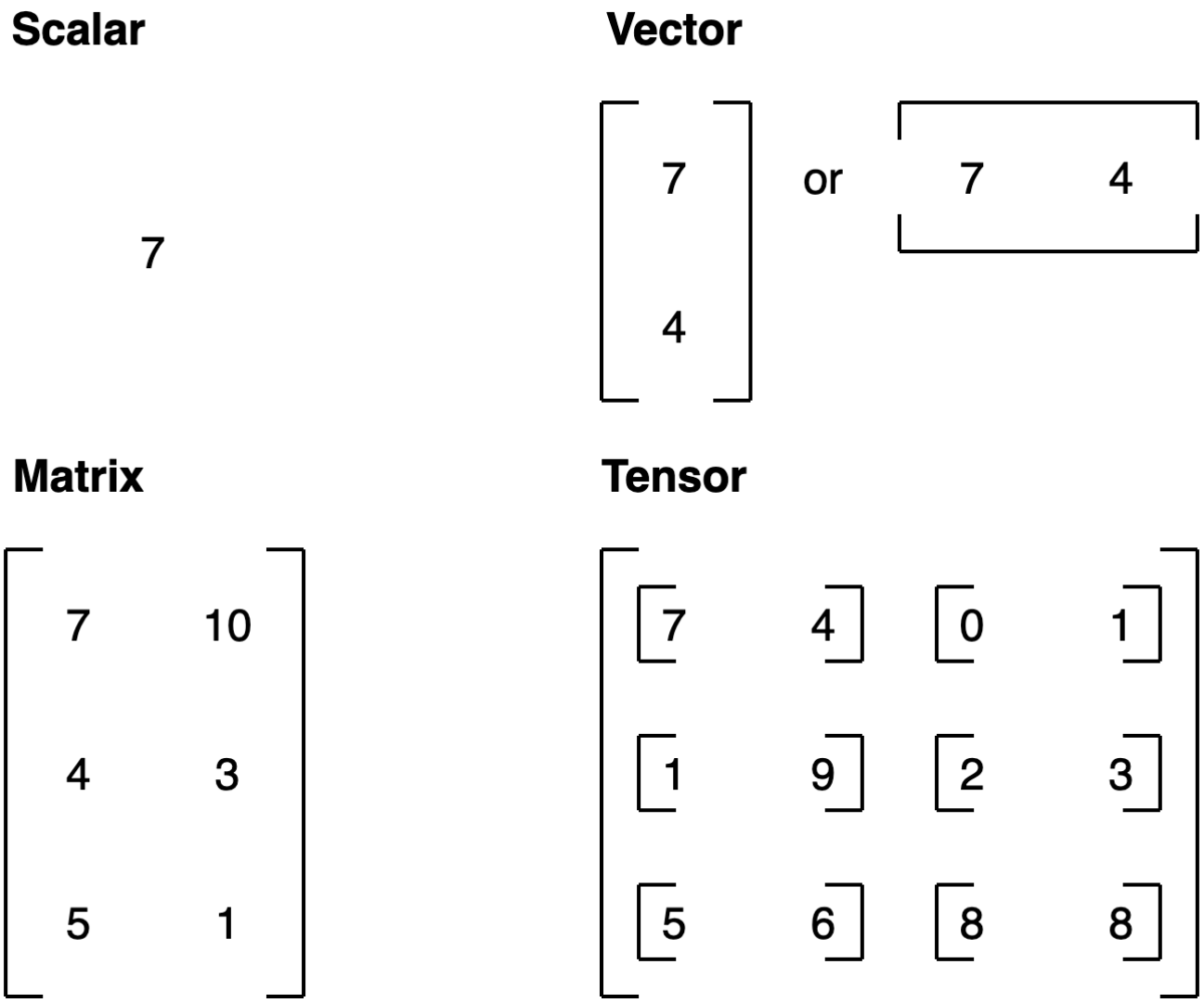
All of the above variables we've created are actually tensors. But you may also hear them referred to as their different names (the ones we gave them):

- **scalar**: a single number.
- **vector**: a number with direction (e.g. wind speed with direction).
- **matrix**: a 2-dimensional array of numbers.
- **tensor**: an n-dimensional arrary of numbers (where n can be any number, a 0-dimension tensor is a scalar, a 1-dimension tensor is a vector).

To add to the confusion, the terms matrix and tensor are often used interchangably.

Going forward since we're using TensorFlow, everything we refer to and use will be tensors.

For more on the mathematical difference between scalars, vectors and matrices see the visual algebra post by Math is Fun.



Creating Tensors with `tf.Variable()`

You can also (although you likely rarely will, because often, when working with data, tensors are created for you automatically) create tensors using `tf.Variable()`.

The difference between `tf.Variable()` and `tf.constant()` is tensors created with `tf.constant()` are immutable (can't be changed, can only be used to create a new tensor), where as, tensors created with `tf.Variable()` are mutable (can be changed).

```
# Create the same tensor with tf.Variable() and tf.constant()
changeable_tensor = tf.Variable([10, 7])
unchangeable_tensor = tf.constant([10, 7])
changeable_tensor, unchangeable_tensor
```

```
(<tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([10,  7],
dtype=int32)>,
 <tf.Tensor: shape=(2,), dtype=int32, numpy=array([10,  7], dtype=int32)>)
```

Now let's try to change one of the elements of the changable tensor.

```
# Will error (requires the .assign() method)
changeable_tensor[0] = 7
changeable_tensor
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-daecfbad2415> in <cell line: 2>()
      1 # Will error (requires the .assign() method)
----> 2 changeable_tensor[0] = 7
      3 changeable_tensor

TypeError: 'ResourceVariable' object does not support item assignment
```

SEARCH STACK OVERFLOW

To change an element of a `tf.Variable()` tensor requires the `assign()` method.

```
# Won't error
changeable_tensor[0].assign(7)
changeable_tensor
```

```
<tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([7, 7], dtype=int32)>
```

Now let's try to change a value in a `tf.constant()` tensor.

```
# Will error (can't change tf.constant())
unchangeable_tensor[0].assign(7)
```

```
unchangleable_tensor
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-16-3947b974feb9> in <cell line: 2>()
      1 # Will error (can't change tf.constant())
----> 2 unchangeable_tensor[0].assign(7)
      3 unchangleable_tensor

/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/ops.py in __getat
    441         np_config.enable_numpy_behavior()
    442       """)
--> 443     self.__getattribute__(name)
    444
    445   @staticmethod

AttributeError: 'tensorflow.python.framework.ops.EagerTensor' object has no attribute
```

SEARCH STACK OVERFLOW

Which one should you use? `tf.constant()` or `tf.Variable()`?

It will depend on what your problem requires. However, most of the time, TensorFlow will automatically choose for you (when loading data or modelling data).

Creating random tensors

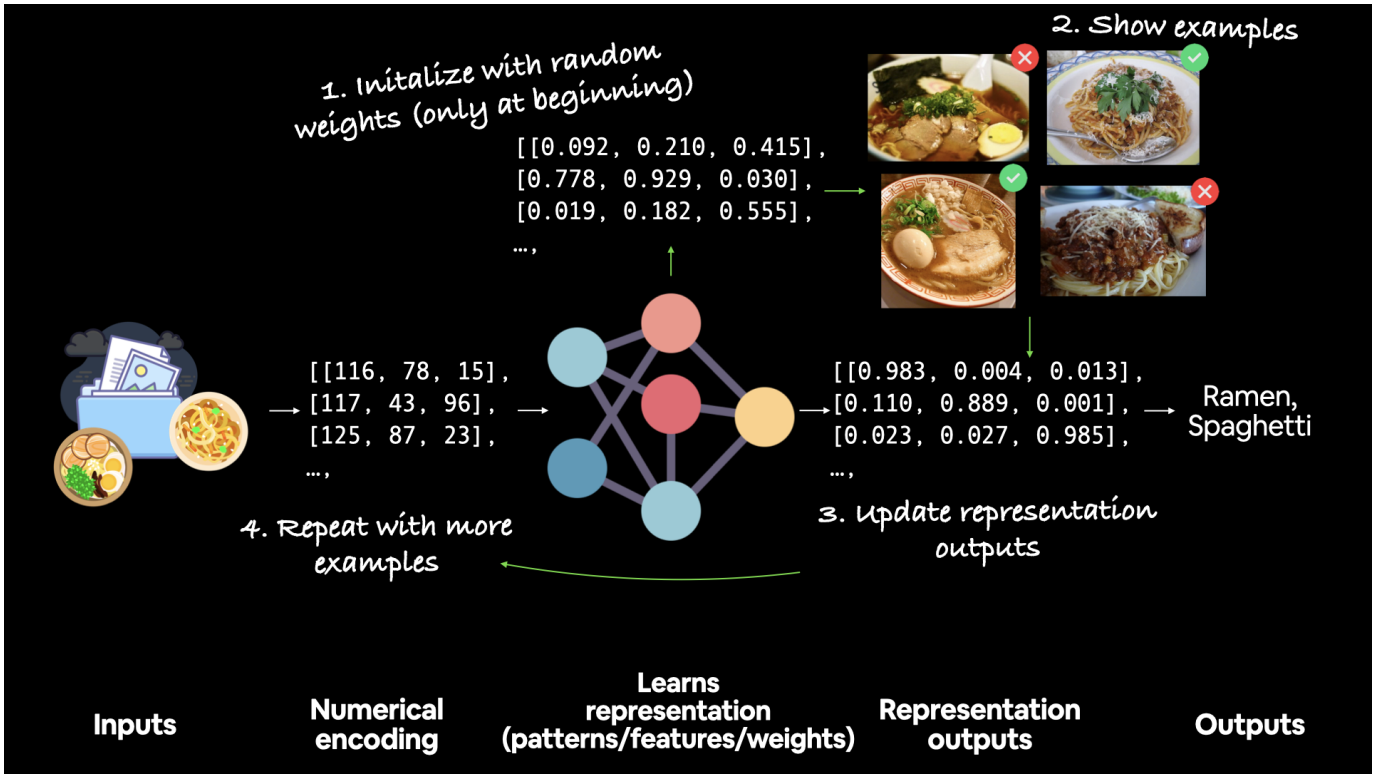
Random tensors are tensors of some abitrary size which contain random numbers.

Why would you want to create random tensors?

This is what neural networks use to intialize their weights (patterns) that they're trying to learn in the data.

For example, the process of a neural network learning often involves taking a random n-dimensional array of numbers and refining them until they represent some kind of pattern (a compressed way to represent the original data).

How a network learns



A network learns by starting with random patterns (1) then going through demonstrative examples of data (2) whilst trying to update its random patterns to represent the examples (3).

We can create random tensors by using the `tf.random.Generator` class.

```
# Create two random (but the same) tensors
random_1 = tf.random.Generator.from_seed(42) # set the seed for reproducibility
random_1 = random_1.normal(shape=(3, 2)) # create tensor from a normal distribution
random_2 = tf.random.Generator.from_seed(42)
random_2 = random_2.normal(shape=(3, 2))

# Are they equal?
random_1, random_2, random_1 == random_2
```

```
(<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
 array([[-0.7565803 , -0.06854702],
        [ 0.07595026, -1.2573844 ],
        [-0.23193765, -1.8107855 ]], dtype=float32)>,
 <tf.Tensor: shape=(3, 2), dtype=float32, numpy=
 array([[-0.7565803 , -0.06854702],
        [ 0.07595026, -1.2573844 ],
        [-0.23193765, -1.8107855 ]], dtype=float32)>,
 <tf.Tensor: shape=(3, 2), dtype=bool, numpy=
 array([[ True,  True],
        [ True,  True],
        [ True,  True]])>)
```

The random tensors we've made are actually pseudorandom numbers (they appear as random, but really aren't).

If we set a seed we'll get the same random numbers (if you've ever used NumPy, this is similar to `np.random.seed(42)`).

Setting the seed says, "hey, create some random numbers, but flavour them with X" (X is the seed).

What do you think will happen when we change the seed?

```
# Create two random (and different) tensors
random_3 = tf.random.Generator.from_seed(42)
random_3 = random_3.normal(shape=(3, 2))
random_4 = tf.random.Generator.from_seed(11)
random_4 = random_4.normal(shape=(3, 2))

# Check the tensors and see if they are equal
random_3, random_4, random_1 == random_3, random_3 == random_4
```

```
(<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
 array([[-0.7565803 , -0.06854702],
        [ 0.07595026, -1.2573844 ],
        [-0.23193765, -1.8107855 ]], dtype=float32)>,
 <tf.Tensor: shape=(3, 2), dtype=float32, numpy=
 array([[ 0.2730574 , -0.29925638],
        [-0.3652325 ,  0.61883307],
        [-1.0130816 ,  0.2829171 ]], dtype=float32)>,
 <tf.Tensor: shape=(3, 2), dtype=bool, numpy=
 array([[ True,  True],
        [ True,  True],
        [ True,  True]])>,
 <tf.Tensor: shape=(3, 2), dtype=bool, numpy=
 array([[False, False],
        [False, False],
        [False, False]])>)
```

What if you wanted to shuffle the order of a tensor?

Wait, why would you want to do that?

Let's say you working with 15,000 images of cats and dogs and the first 10,000 images of were of cats and the next 5,000 were of dogs. This order could effect how a neural network learns (it may overfit by learning the order of the data), instead, it might be a good idea to move your data around.

```
# Shuffle a tensor (valuable for when you want to shuffle your data)
not_shuffled = tf.constant([[10, 7],
                            [3, 4],
                            [2, 5]])
# Gets different results each time
tf.random.shuffle(not_shuffled)
```

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[ 2,  5],
```

```
       [ 3,  4],
       [10,  7]], dtype=int32)>
```

```
# Shuffle in the same order every time using the seed parameter (won't acutally be the sam
tf.random.shuffle(not_shuffled, seed=42)
```

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[ 2,  5],
       [ 3,  4],
       [10,  7]], dtype=int32)>
```

Wait... why didn't the numbers come out the same?

It's due to rule #4 of the `tf.random.set_seed()` documentation.

> "4. If both the global and the operation seed are set: Both seeds are used in conjunction to determine the random sequence."

`tf.random.set_seed(42)` sets the global seed, and the `seed` parameter in `tf.random.shuffle(seed=42)` sets the operation seed.

Because, "Operations that rely on a random seed actually derive it from two seeds: the global and operation-level seeds. This sets the global seed."

```
# Shuffle in the same order every time

# Set the global random seed
tf.random.set_seed(42)

# Set the operation random seed
tf.random.shuffle(not_shuffled, seed=42)
```

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[10,  7],
       [ 3,  4],
       [ 2,  5]], dtype=int32)>
```

```
# Set the global random seed
tf.random.set_seed(42) # if you comment this out you'll get different results

# Set the operation random seed
tf.random.shuffle(not_shuffled)
```

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[ 3,  4],
       [ 2,  5],
       [10,  7]], dtype=int32)>
```

Other ways to make tensors

Though you might rarely use these (remember, many tensor operations are done behind the scenes for you), you can use `tf.ones()` to create a tensor of all ones and `tf.zeros()` to create a tensor of all zeros.

```
# Make a tensor of all ones
tf.ones(shape=(3, 2))
```

```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[1., 1.],
       [1., 1.],
       [1., 1.]], dtype=float32)>
```

```
# Make a tensor of all zeros
tf.zeros(shape=(3, 2))
```

```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[0., 0.],
       [0., 0.],
       [0., 0.]], dtype=float32)>
```

You can also turn NumPy arrays in into tensors.

Remember, the main difference between tensors and NumPy arrays is that tensors can be run on GPUs.

> 🔑 **Note:** A matrix or tensor is typically represented by a capital letter (e.g. `X` or `A`) where as a vector is typically represented by a lowercase letter (e.g. `y` or `b`).

```
import numpy as np
numpy_A = np.arange(1, 25, dtype=np.int32) # create a NumPy array between 1 and 25
A = tf.constant(numpy_A,
                shape=[2, 4, 3]) # note: the shape total (2*4*3) has to match the number o
numpy_A, A
```

```
(array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
        18, 19, 20, 21, 22, 23, 24], dtype=int32),
 <tf.Tensor: shape=(2, 4, 3), dtype=int32, numpy=
 array([[[ 1,  2,  3],
         [ 4,  5,  6],
         [ 7,  8,  9],
         [10, 11, 12]],

        [[13, 14, 15],
         [16, 17, 18],
         [19, 20, 21],
         [22, 23, 24]]], dtype=int32)>)
```

Getting information from tensors (shape, rank, size)

There will be times when you'll want to get different pieces of information from your tensors, in particuluar, you should know the following tensor vocabulary:

- **Shape:** The length (number of elements) of each of the dimensions of a tensor.
- **Rank:** The number of tensor dimensions. A scalar has rank 0, a vector has rank 1, a matrix is rank 2, a tensor has rank n.
- **Axis** or **Dimension:** A particular dimension of a tensor.
- **Size:** The total number of items in the tensor.

You'll use these especially when you're trying to line up the shapes of your data to the shapes of your model. For example, making sure the shape of your image tensors are the same shape as your models input layer.

We've already seen one of these before using the `ndim` attribute. Let's see the rest.

```
# Create a rank 4 tensor (4 dimensions)
rank_4_tensor = tf.zeros([2, 3, 4, 5])
rank_4_tensor
```

```
<tf.Tensor: shape=(2, 3, 4, 5), dtype=float32, numpy=
array([[[[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]]],


       [[[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]]]], dtype=float32)>
```

```
rank_4_tensor.shape, rank_4_tensor.ndim, tf.size(rank_4_tensor)
```

```
(TensorShape([2, 3, 4, 5]), 4, <tf.Tensor: shape=(), dtype=int32, numpy=120>)
```

```
# Get various attributes of tensor
print("Datatype of every element:", rank_4_tensor.dtype)
print("Number of dimensions (rank):", rank_4_tensor.ndim)
print("Shape of tensor:", rank_4_tensor.shape)
print("Elements along axis 0 of tensor:", rank_4_tensor.shape[0])
print("Elements along last axis of tensor:", rank_4_tensor.shape[-1])
print("Total number of elements (2*3*4*5):", tf.size(rank_4_tensor).numpy()) # .numpy() co
```

```
Datatype of every element: <dtype: 'float32'>
Number of dimensions (rank): 4
Shape of tensor: (2, 3, 4, 5)
Elements along axis 0 of tensor: 2
Elements along last axis of tensor: 5
Total number of elements (2*3*4*5): 120
```

You can also index tensors just like Python lists.

```
# Get the first 2 items of each dimension
rank_4_tensor[:2, :2, :2, :2]
```

```
<tf.Tensor: shape=(2, 2, 2, 2), dtype=float32, numpy=
array([[[[0., 0.],
         [0., 0.]],

        [[0., 0.],
         [0., 0.]]],


       [[[0., 0.],
         [0., 0.]],

        [[0., 0.],
         [0., 0.]]]], dtype=float32)>
```

```
# Get the dimension from each index except for the final one
rank_4_tensor[:1, :1, :1, :]
```

```
<tf.Tensor: shape=(1, 1, 1, 5), dtype=float32, numpy=array([[[[0., 0., 0., 0.,
0.]]]], dtype=float32)>
```

```
# Create a rank 2 tensor (2 dimensions)
rank_2_tensor = tf.constant([[10, 7],
                             [3, 4]])

# Get the last item of each row
rank_2_tensor[:, -1]
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([7, 4], dtype=int32)>
```

You can also add dimensions to your tensor whilst keeping the same information present using `tf.newaxis`.

```
# Add an extra dimension (to the end)
rank_3_tensor = rank_2_tensor[..., tf.newaxis] # in Python "..." means "all dimensions pri
rank_2_tensor, rank_3_tensor # shape (2, 2), shape (2, 2, 1)
```

```
(<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
 array([[10,  7],
        [ 3,  4]], dtype=int32)>,
 <tf.Tensor: shape=(2, 2, 1), dtype=int32, numpy=
 array([[[10],
         [ 7]],

        [[ 3],
         [ 4]]], dtype=int32)>)
```

You can achieve the same using `tf.expand_dims()`.

```
tf.expand_dims(rank_2_tensor, axis=-1) # "-1" means last axis
```

```
<tf.Tensor: shape=(2, 2, 1), dtype=int32, numpy=
array([[[10],
        [ 7]],

       [[ 3],
        [ 4]]], dtype=int32)>
```

Manipulating tensors (tensor operations)

Finding patterns in tensors (numberical representation of data) requires manipulating them.

Again, when building models in TensorFlow, much of this pattern discovery is done for you.

Basic operations

You can perform many of the basic mathematical operations directly on tensors using Pyhton operators such as, `+`, `-`, `*`.

```
# You can add values to a tensor using the addition operator
tensor = tf.constant([[10, 7], [3, 4]])
tensor + 10
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[20, 17],
       [13, 14]], dtype=int32)>
```

Since we used `tf.constant()`, the original tensor is unchanged (the addition gets done on a copy).

```
# Original tensor unchanged
tensor
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[10,  7],
       [ 3,  4]], dtype=int32)>
```

Other operators also work.

```
# Multiplication (known as element-wise multiplication)
tensor * 10
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[100,  70],
       [ 30,  40]], dtype=int32)>
```

```
# Subtraction
tensor - 10
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 0, -3],
       [-7, -6]], dtype=int32)>
```

You can also use the equivalent TensorFlow function. Using the TensorFlow function (where possible) has the advantage of being sped up later down the line when running as part of a TensorFlow graph.

```
# Use the tensorflow function equivalent of the '*' (multiply) operator
tf.multiply(tensor, 10)
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[100,  70],
       [ 30,  40]], dtype=int32)>
```

```
# The original tensor is still unchanged
tensor
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[10,  7],
       [ 3,  4]], dtype=int32)>
```

Matrix mutliplication

One of the most common operations in machine learning algorithms is matrix multiplication.

TensorFlow implements this matrix multiplication functionality in the `tf.matmul()` method.

The main two rules for matrix multiplication to remember are:

1. The inner dimensions must match:

- `(3, 5) @ (3, 5)` won't work
- `(5, 3) @ (3, 5)` will work
- `(3, 5) @ (5, 3)` will work

2. The resulting matrix has the shape of the outer dimensions:

- `(5, 3) @ (3, 5) -> (5, 5)`
- `(3, 5) @ (5, 3) -> (3, 3)`

> 🔑 **Note:** '`@`' in Python is the symbol for matrix multiplication.

```
# Matrix multiplication in TensorFlow
print(tensor)
tf.matmul(tensor, tensor)
```

```
tf.Tensor(
[[10  7]
 [ 3  4]], shape=(2, 2), dtype=int32)
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[121,  98],
       [ 42,  37]], dtype=int32)>
```

```
# Matrix multiplication with Python operator '@'
tensor @ tensor
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[121,  98],
       [ 42,  37]], dtype=int32)>
```

Both of these examples work because our `tensor` variable is of shape (2, 2).

What if we created some tensors which had mismatched shapes?

```
# Create (3, 2) tensor
X = tf.constant([[1, 2],
                 [3, 4],
                 [5, 6]])

# Create another (3, 2) tensor
Y = tf.constant([[7, 8],
                 [9, 10],
                 [11, 12]])
X, Y
```

```
(<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
 array([[1, 2],
        [3, 4],
        [5, 6]], dtype=int32)>,
 <tf.Tensor: shape=(3, 2), dtype=int32, numpy=
 array([[ 7,  8],
        [ 9, 10],
        [11, 12]], dtype=int32)>)
```

```
# Try to matrix multiply them (will error)
X @ Y
```

```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-43-62e1e4702ffd> in <cell line: 2>()
      1 # Try to matrix multiply them (will error)
----> 2 X @ Y

                                 1 frames

/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/ops.py in
raise_from_not_ok_status(e, name)
   7260 def raise_from_not_ok_status(e, name):
   7261   e.message += (" name: " + name if name is not None else "")
-> 7262   raise core._status_to_exception(e) from None  # pylint: disable=protected-a
   7263
   7264

InvalidArgumentError: {{function_node
__wrapped__MatMul_device_/job:localhost/replica:0/task:0/device:CPU:0}} Matrix size-i
In[0]: [3,2], In[1]: [3,2] [Op:MatMul]
```

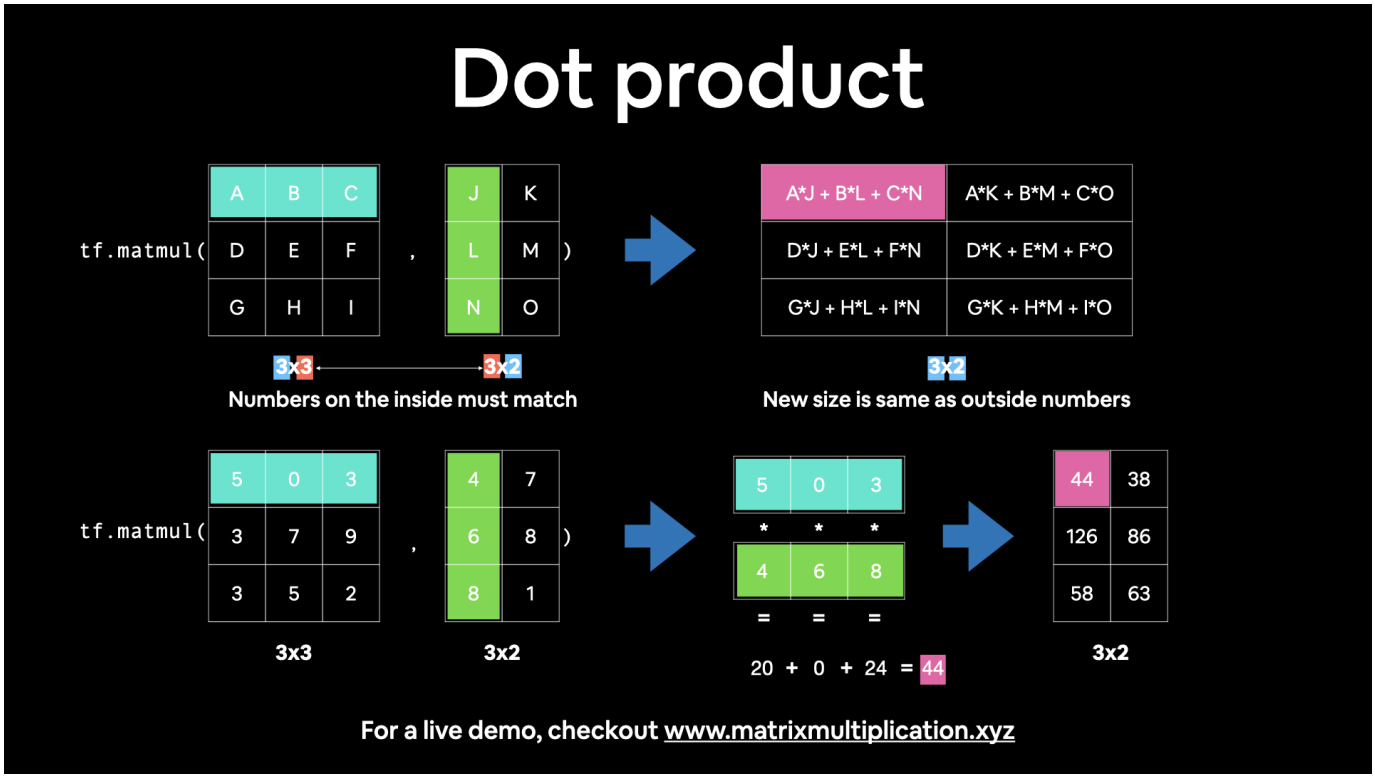
Trying to matrix multiply two tensors with the shape `(3, 2)` errors because the inner dimensions don't match.

We need to either:

- Reshape X to `(2, 3)` so it's `(2, 3) @ (3, 2)`.
- Reshape Y to `(3, 2)` so it's `(3, 2) @ (2, 3)`.

We can do this with either:

- `tf.reshape()` - allows us to reshape a tensor into a defined shape.
- `tf.transpose()` - switches the dimensions of a given tensor.



Let's try `tf.reshape()` first.

```
# Example of reshape (3, 2) -> (2, 3)
tf.reshape(Y, shape=(2, 3))
```

```
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[ 7,  8,  9],
       [10, 11, 12]], dtype=int32)>
```

```
# Try matrix multiplication with reshaped Y
X @ tf.reshape(Y, shape=(2, 3))
```

```
<tf.Tensor: shape=(3, 3), dtype=int32, numpy=
array([[ 27,  30,  33],
       [ 61,  68,  75],
       [ 95, 106, 117]], dtype=int32)>
```

It worked, let's try the same with a reshaped `X`, except this time we'll use [`tf.transpose()`] and `tf.matmul()`.

```
# Example of transpose (3, 2) -> (2, 3)
tf.transpose(X)
```

```
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[1, 3, 5],
       [2, 4, 6]], dtype=int32)>
```

```
# Try matrix multiplication
tf.matmul(tf.transpose(X), Y)
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 89,  98],
       [116, 128]], dtype=int32)>
```

```
# You can achieve the same result with parameters
tf.matmul(a=X, b=Y, transpose_a=True, transpose_b=False)
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 89,  98],
       [116, 128]], dtype=int32)>
```

Notice the difference in the resulting shapes when tranposing `X` or reshaping `Y`.

This is because of the 2nd rule mentioned above:

- `(3, 2) @ (2, 3) -> (3, 3)` done with `X @ tf.reshape(Y, shape=(2, 3))`
- `(2, 3) @ (3, 2) -> (2, 2)` done with `tf.matmul(tf.transpose(X), Y)`

This kind of data manipulation is a reminder: you'll spend a lot of your time in machine learning and working with neural networks reshaping data (in the form of tensors) to prepare it to be used with various operations (such as feeding it to a model).

The dot product

Multiplying matrices by eachother is also referred to as the dot product.

You can perform the `tf.matmul()` operation using `tf.tensordot()`.

```
# Perform the dot product on X and Y (requires X to be transposed)
tf.tensordot(tf.transpose(X), Y, axes=1)
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 89,  98],
       [116, 128]], dtype=int32)>
```

You might notice that although using both `reshape` and `tranpose` work, you get different results when using each.

Let's see an example, first with `tf.transpose()` then with `tf.reshape()`.

```
# Perform matrix multiplication between X and Y (transposed)
tf.matmul(X, tf.transpose(Y))
```

```
<tf.Tensor: shape=(3, 3), dtype=int32, numpy=
array([[ 23,  29,  35],
       [ 53,  67,  81],
       [ 83, 105, 127]], dtype=int32)>
```

```
# Perform matrix multiplication between X and Y (reshaped)
tf.matmul(X, tf.reshape(Y, (2, 3)))
```

```
<tf.Tensor: shape=(3, 3), dtype=int32, numpy=
array([[ 27,  30,  33],
       [ 61,  68,  75],
       [ 95, 106, 117]], dtype=int32)>
```

Hmm... they result in different values.

Which is strange because when dealing with `Y` (a `(3x2)` matrix), reshaping to `(2, 3)` and tranposing it result in the same shape.

```
# Check shapes of Y, reshaped Y and tranposed Y
Y.shape, tf.reshape(Y, (2, 3)).shape, tf.transpose(Y).shape
```

```
(TensorShape([3, 2]), TensorShape([2, 3]), TensorShape([2, 3]))
```

But calling `tf.reshape()` and `tf.transpose()` on `Y` don't necessarily result in the same values.

```
# Check values of Y, reshape Y and tranposed Y
print("Normal Y:")
print(Y, "\n") # "\n" for newline

print("Y reshaped to (2, 3):")
print(tf.reshape(Y, (2, 3)), "\n")

print("Y transposed:")
print(tf.transpose(Y))
```

```
Normal Y:
tf.Tensor(
[[ 7  8]
 [ 9 10]
 [11 12]], shape=(3, 2), dtype=int32)

Y reshaped to (2, 3):
tf.Tensor(
[[ 7  8  9]
 [10 11 12]], shape=(2, 3), dtype=int32)

Y transposed:
tf.Tensor(
[[ 7  9 11]
 [ 8 10 12]], shape=(2, 3), dtype=int32)
```

As you can see, the outputs of `tf.reshape()` and `tf.transpose()` when called on `Y`, even though they have the same shape, are different.

This can be explained by the default behaviour of each method:

- `tf.reshape()` - change the shape of the given tensor (first) and then insert values in order they appear (in our case, 7, 8, 9, 10, 11, 12).

- `tf.transpose()` - swap the order of the axes, by default the last axis becomes the first, however the order can be changed using the `perm` parameter.

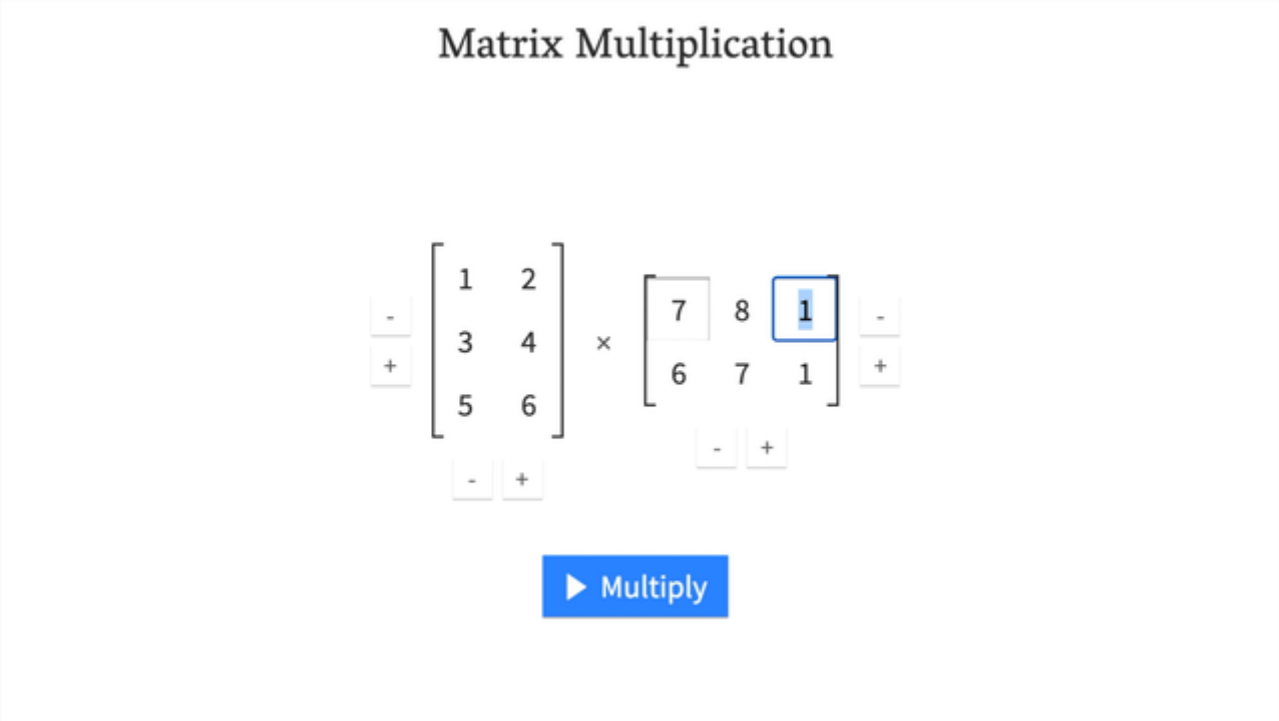
So which should you use?

Again, most of the time these operations (when they need to be run, such as during the training a neural network, will be implemented for you).

But generally, whenever performing a matrix multiplication and the shapes of two matrices don't line up, you will transpose (not reshape) one of them in order to line them up.

Matrix multiplication tidbits

- If we transposed `Y`, it would be represented as $\mathbf{Y}^T$ (note the capital T for tranpose).
- Get an illustrative view of matrix multiplication by Math is Fun.
- Try a hands-on demo of matrix multiplcation: http://matrixmultiplication.xyz/ (shown below).



Changing the datatype of a tensor

Sometimes you'll want to alter the default datatype of your tensor.

This is common when you want to compute using less precision (e.g. 16-bit floating point numbers vs. 32-bit floating point numbers).

Computing with less precision is useful on devices with less computing capacity such as mobile devices (because the less bits, the less space the computations require).

You can change the datatype of a tensor using `tf.cast()`.

```
# Create a new tensor with default datatype (float32)
B = tf.constant([1.7, 7.4])

# Create a new tensor with default datatype (int32)
C = tf.constant([1, 7])
B, C
```

```
(<tf.Tensor: shape=(2,), dtype=float32, numpy=array([1.7, 7.4], dtype=float32)>,
 <tf.Tensor: shape=(2,), dtype=int32, numpy=array([1, 7], dtype=int32)>)
```

```
# Change from float32 to float16 (reduced precision)
B = tf.cast(B, dtype=tf.float16)
B
```

```
<tf.Tensor: shape=(2,), dtype=float16, numpy=array([1.7, 7.4], dtype=float16)>
```

```
# Change from int32 to float32
C = tf.cast(C, dtype=tf.float32)
C
```

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([1., 7.], dtype=float32)>
```

Getting the absolute value

Sometimes you'll want the absolute values (all values are positive) of elements in your tensors.

To do so, you can use `tf.abs()`.

```
# Create tensor with negative values
D = tf.constant([-7, -10])
D
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([ -7, -10], dtype=int32)>
```

```
# Get the absolute values
tf.abs(D)
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([ 7, 10], dtype=int32)>
```

Finding the min, max, mean, sum (aggregation)

You can quickly aggregate (perform a calculation on a whole tensor) tensors to find things like the minimum value, maximum value, mean and sum of all the elements.

To do so, aggregation methods typically have the syntax `reduce()_[action]`, such as:

- `tf.reduce_min()` - find the minimum value in a tensor.
- `tf.reduce_max()` - find the maximum value in a tensor (helpful for when you want to find the highest prediction probability).

- `tf.reduce_mean()` - find the mean of all elements in a tensor.
- `tf.reduce_sum()` - find the sum of all elements in a tensor.
- **Note:** typically, each of these is under the `math` module, e.g. `tf.math.reduce_min()` but you can use the alias `tf.reduce_min()`.

Let's see them in action.

```
# Create a tensor with 50 random values between 0 and 100
import numpy as np
E = tf.constant(np.random.randint(low=0, high=70, size=50))
E
```

```
<tf.Tensor: shape=(50,), dtype=int64, numpy=
array([19, 54,  5, 22, 61, 59, 28, 65, 65, 18, 20, 45, 34, 21, 48, 26, 36,
       16, 63,  8, 15, 51, 59, 22, 44, 19,  5, 47, 51, 49, 44, 40, 42, 57,
       25, 68, 52, 14, 24,  3, 47, 41, 11, 63, 43, 37, 31, 36, 31, 57])>
```

```
# Find the minimum
tf.reduce_min(E)
```

```
<tf.Tensor: shape=(), dtype=int64, numpy=6>
```

```
# Find the maximum
tf.reduce_max(E)
```

```
<tf.Tensor: shape=(), dtype=int64, numpy=98>
```

```
# Find the mean
tf.reduce_mean(E)
```

```
<tf.Tensor: shape=(), dtype=int64, numpy=46>
```

```
# Find the sum
tf.reduce_sum(E)
```

```
<tf.Tensor: shape=(), dtype=int64, numpy=2320>
```

You can also find the standard deviation (`tf.reduce_std()`) and variance (`tf.reduce_variance()`) of elements in a tensor using similar methods.

Finding the positional maximum and minimum

How about finding the position a tensor where the maximum value occurs?

This is helpful when you want to line up your labels (say `['Green', 'Blue', 'Red']`) with your prediction probabilities tensor (e.g. `[0.98, 0.01, 0.01]`).

In this case, the predicted label (the one with the highest prediction probability) would be `'Green'`.

You can do the same for the minimum (if required) with the following:

- `tf.argmax()` - find the position of the maximum element in a given tensor.
- `tf.argmin()` - find the position of the minimum element in a given tensor.

```
# Create a tensor with 50 values between 0 and 1
F = tf.constant(np.random.random(50))
F
```

```
<tf.Tensor: shape=(50,), dtype=float64, numpy=
array([0.08892525, 0.94484011, 0.72484292, 0.11100388, 0.23637676,
       0.40758941, 0.78697704, 0.51382876, 0.67817528, 0.33801469,
       0.06847686, 0.43940259, 0.05384978, 0.90974966, 0.17755086,
       0.55687455, 0.22395101, 0.61040988, 0.19115218, 0.43669498,
       0.95362449, 0.65974345, 0.98141608, 0.72890794, 0.31333329,
       0.95735583, 0.80562309, 0.08673455, 0.5237697 , 0.9006758 ,
       0.07103048, 0.88667591, 0.70505817, 0.79932324, 0.28416341,
       0.50271115, 0.2614137 , 0.22194647, 0.96336433, 0.88853101,
       0.23221737, 0.92196873, 0.84103254, 0.01333408, 0.24513585,
       0.74766312, 0.8508123 , 0.94218343, 0.90917265, 0.78489794])>
```

```
# Find the maximum element position of F
tf.argmax(F)
```

```
<tf.Tensor: shape=(), dtype=int64, numpy=22>
```

```
# Find the minimum element position of F
tf.argmin(F)
```

```
<tf.Tensor: shape=(), dtype=int64, numpy=43>
```

```
# Find the maximum element position of F
print(f"The maximum value of F is at position: {tf.argmax(F).numpy()}")
print(f"The maximum value of F is: {tf.reduce_max(F).numpy()}")
print(f"Using tf.argmax() to index F, the maximum value of F is: {F[tf.argmax(F)].numpy()}
print(f"Are the two max values the same (they should be)? {F[tf.argmax(F)].numpy() == tf.r
```

```
The maximum value of F is at position: 22
The maximum value of F is: 0.9814160834311638
Using tf.argmax() to index F, the maximum value of F is: 0.9814160834311638
Are the two max values the same (they should be)? True
```

Squeezing a tensor (removing all single dimensions)

If you need to remove single-dimensions from a tensor (dimensions with size 1), you can use `tf.squeeze()`.

- `tf.squeeze()` - remove all dimensions of 1 from a tensor.

```
# Create a rank 5 (5 dimensions) tensor of 50 numbers between 0 and 100
G = tf.constant(np.random.randint(0, 100, 50), shape=(1, 1, 1, 1, 50))
```

```
G.shape, G.ndim
```

```
(TensorShape([1, 1, 1, 1, 50]), 5)
```

```
# Squeeze tensor G (remove all 1 dimensions)
G_squeezed = tf.squeeze(G)
G_squeezed.shape, G_squeezed.ndim
```

```
(TensorShape([50]), 1)
```

One-hot encoding

If you have a tensor of indicies and would like to one-hot encode it, you can use `tf.one_hot()`.

You should also specify the `depth` parameter (the level which you want to one-hot encode to).

```
# Create a list of indices
some_list = [0, 1, 2, 3]

# One hot encode them
tf.one_hot(some_list, depth=4)
```

```
<tf.Tensor: shape=(4, 4), dtype=float32, numpy=
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]], dtype=float32)>
```

You can also specify values for `on_value` and `off_value` instead of the default `0` and `1`.

```
# Specify custom values for on and off encoding
tf.one_hot(some_list, depth=4, on_value="We're live!", off_value="Offline")
```

```
<tf.Tensor: shape=(4, 4), dtype=string, numpy=
array([[b"We're live!", b'Offline', b'Offline', b'Offline'],
       [b'Offline', b"We're live!", b'Offline', b'Offline'],
       [b'Offline', b'Offline', b"We're live!", b'Offline'],
       [b'Offline', b'Offline', b'Offline', b"We're live!"]], dtype=object)>
```

Squaring, log, square root

Many other common mathematical operations you'd like to perform at some stage, probably exist.

Let's take a look at:

- `tf.square()` - get the square of every value in a tensor.
- `tf.sqrt()` - get the squareroot of every value in a tensor (**note:** the elements need to be floats or this will error).
- `tf.math.log()` - get the natural log of every value in a tensor (elements need to floats).

```
# Create a new tensor
H = tf.constant(np.arange(1, 10))
H
```

```
<tf.Tensor: shape=(9,), dtype=int64, numpy=array([1, 2, 3, 4, 5, 6, 7, 8, 9])>
```

```
# Square it
tf.square(H)
```

```
<tf.Tensor: shape=(9,), dtype=int64, numpy=array([ 1,  4,  9, 16, 25, 36, 49, 64,
81])>
```

```
# Find the squareroot (will error), needs to be non-integer
tf.sqrt(H)
```

```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-75-d7db039da8bb> in <cell line: 2>()
      1 # Find the squareroot (will error), needs to be non-integer
----> 2 tf.sqrt(H)

 1 frames 
/usr/local/lib/python3.9/dist-packages/tensorflow/python/framework/ops.py in
raise_from_not_ok_status(e, name)
   7260 def raise_from_not_ok_status(e, name):
   7261   e.message += (" name: " + name if name is not None else "")
-> 7262   raise core._status_to_exception(e) from None  # pylint: disable=protected-
access
   7263
   7264

InvalidArgumentError: Value for attr 'T' of int64 is not in the list of allowed
values: bfloat16, half, float, double, complex64, complex128
        ; NodeDef: {{node Sqrt}}; Op<name=Sqrt; signature=x:T -> y:T;
attr=T:type,allowed=[DT_BFLOAT16, DT_HALF, DT_FLOAT, DT_DOUBLE, DT_COMPLEX64,
```

```
# Change H to float32
H = tf.cast(H, dtype=tf.float32)
H
```

```
<tf.Tensor: shape=(9,), dtype=float32, numpy=array([1., 2., 3., 4., 5., 6., 7., 8.,
9.], dtype=float32)>
```

```
# Find the square root
tf.sqrt(H)
```

```
<tf.Tensor: shape=(9,), dtype=float32, numpy=
array([1.       , 1.4142135, 1.7320508, 2.       , 2.2360678, 2.4494896,
       2.6457512, 2.828427 , 3.       ], dtype=float32)>
```

```
# Find the log (input also needs to be float)
tf.math.log(H)
```

```
<tf.Tensor: shape=(9,), dtype=float32, numpy=
array([0.       , 0.6931472, 1.0986123, 1.3862944, 1.609438 , 1.7917595,
       1.9459102, 2.0794415, 2.1972246], dtype=float32)>
```

Manipulating `tf.Variable` tensors

Tensors created with `tf.Variable()` can be changed in place using methods such as:

- `.assign()` - assign a different value to a particular index of a variable tensor.
- `.add_assign()` - add to an existing value and reassign it at a particular index of a variable tensor.

```
# Create a variable tensor
I = tf.Variable(np.arange(0, 5))
I
```

```
<tf.Variable 'Variable:0' shape=(5,) dtype=int64, numpy=array([0, 1, 2, 3, 4])>
```

```
# Assign the final value a new value of 50
I.assign([0, 1, 2, 3, 50])
```

```
<tf.Variable 'UnreadVariable' shape=(5,) dtype=int64, numpy=array([ 0,  1,  2,  3,
50])>
```

```
# The change happens in place (the last value is now 50, not 4)
I
```

```
<tf.Variable 'Variable:0' shape=(5,) dtype=int64, numpy=array([ 0,  1,  2,  3, 50])>
```

```
# Add 10 to every element in I
I.assign_add([10, 10, 10, 10, 10])
```

```
<tf.Variable 'UnreadVariable' shape=(5,) dtype=int64, numpy=array([10, 11, 12, 13,
60])>
```

```
# Again, the change happens in place
I
```

```
<tf.Variable 'Variable:0' shape=(5,) dtype=int64, numpy=array([10, 11, 12, 13, 60])>
```

Tensors and NumPy

We've seen some examples of tensors interact with NumPy arrays, such as, using NumPy arrays to create tensors.

Tensors can also be converted to NumPy arrays using:

- `np.array()` - pass a tensor to convert to an ndarray (NumPy's main datatype).

- `tensor.numpy()` - call on a tensor to convert to an ndarray.

Doing this is helpful as it makes tensors iterable as well as allows us to use any of NumPy's methods on them.

```
# Create a tensor from a NumPy array
J = tf.constant(np.array([3., 7., 10.]))
J
```

```
<tf.Tensor: shape=(3,), dtype=float64, numpy=array([ 3.,  7., 10.])>
```

```
# Convert tensor J to NumPy with np.array()
np.array(J), type(np.array(J))
```

```
(array([ 3.,  7., 10.]), numpy.ndarray)
```

```
# Convert tensor J to NumPy with .numpy()
J.numpy(), type(J.numpy())
```

```
(array([ 3.,  7., 10.]), numpy.ndarray)
```

By default tensors have `dtype=float32`, where as NumPy arrays have `dtype=float64`.

This is because neural networks (which are usually built with TensorFlow) can generally work very well with less precision (32-bit rather than 64-bit).

```
# Create a tensor from NumPy and from an array
numpy_J = tf.constant(np.array([3., 7., 10.])) # will be float64 (due to NumPy)
tensor_J = tf.constant([3., 7., 10.]) # will be float32 (due to being TensorFlow default)
numpy_J.dtype, tensor_J.dtype
```

```
(tf.float64, tf.float32)
```

Using `@tf.function`

In your TensorFlow adventures, you might come across Python functions which have the decorator `@tf.function`.

If you aren't sure what Python decorators do, read RealPython's guide on them.

But in short, decorators modify a function in one way or another.

In the `@tf.function` decorator case, it turns a Python function into a callable TensorFlow graph. Which is a fancy way of saying, if you've written your own Python function, and you decorate it with `@tf.function`, when you export your code (to potentially run on another device), TensorFlow will attempt to convert it into a fast(er) version of itself (by making it part of a computation graph).

For more on this, read the Better performnace with tf.function guide.

```
# Create a simple function
def function(x, y):
  return x ** 2 + y

x = tf.constant(np.arange(0, 10))
y = tf.constant(np.arange(10, 20))
function(x, y)
```

```
<tf.Tensor: shape=(10,), dtype=int64, numpy=array([ 10,  12,  16,  22,  30,  40,
52,  66,  82, 100])>
```

```
# Create the same function and decorate it with tf.function
@tf.function
def tf_function(x, y):
  return x ** 2 + y

tf_function(x, y)
```

```
<tf.Tensor: shape=(10,), dtype=int64, numpy=array([ 10,  12,  16,  22,  30,  40,
52,  66,  82, 100])>
```

If you noticed no difference between the above two functions (the decorated one and the non-decorated one) you'd be right.

Much of the difference happens behind the scenes. One of the main ones being potential code speed-ups where possible.

Finding access to GPUs

We've mentioned GPUs plenty of times throughout this notebook.

So how do you check if you've got one available?

You can check if you've got access to a GPU using `tf.config.list_physical_devices()`.

```
print(tf.config.list_physical_devices('GPU'))
```

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

If the above outputs an empty array (or nothing), it means you don't have access to a GPU (or at least TensorFlow can't find it).

If you're running in Google Colab, you can access a GPU by going to *Runtime -> Change Runtime Type -> Select GPU* (**note:** after doing this your notebook will restart and any variables you've saved will be lost).

Once you've changed your runtime type, run the cell below.

```
import tensorflow as tf
print(tf.config.list_physical_devices('GPU'))
```

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

If you've got access to a GPU, the cell above should output something like:

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

You can also find information about your GPU using `!nvidia-smi`.

```
!nvidia-smi
```

```
Tue Apr 25 05:24:15 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.85.12    Driver Version: 525.85.12    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A100-SXM...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   32C    P0    51W / 400W |    693MiB / 40960MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

> 🔑 **Note:** If you have access to a GPU, TensorFlow will automatically use it whenever possible.

🛠 Exercises

1. Create a vector, scalar, matrix and tensor with values of your choosing using `tf.constant()`.
2. Find the shape, rank and size of the tensors you created in 1.
3. Create two tensors containing random values between 0 and 1 with shape `[5, 300]`.
4. Multiply the two tensors you created in 3 using matrix multiplication.
5. Multiply the two tensors you created in 3 using dot product.
6. Create a tensor with random values between 0 and 1 with shape `[224, 224, 3]`.
7. Find the min and max values of the tensor you created in 6.
8. Created a tensor with random values of shape `[1, 224, 224, 3]` then squeeze it to change the shape to `[224, 224, 3]`.

9. Create a tensor with shape `[10]` using your own choice of values, then find the index which has the maximum value.
10. One-hot encode the tensor you created in 9.

📖 Extra-curriculum

- Read through the list of TensorFlow Python APIs, pick one we haven't gone through in this notebook, reverse engineer it (write out the documentation code for yourself) and figure out what it does.
- Try to create a series of tensor functions to calculate your most recent grocery bill (it's okay if you don't use the names of the items, just the price in numerical form).
    - How would you calculate your grocery bill for the month and for the year using tensors?
- Go through the TensorFlow 2.x quick start for beginners tutorial (be sure to type out all of the code yourself, even if you don't understand it).
    - Are there any functions we used in here that match what's used in there? Which are the same? Which haven't you seen before?
- Watch the video "What's a tensor?" - a great visual introduction to many of the concepts we've covered in this notebook.

```
#questions 1
scalar1=tf.constant(5)
scalar1
vector1=tf.constant([5,7])
vector1
matrix1=tf.constant([[14,45],[45,25]])
matrix1

\
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[14, 45],
       [45, 25]], dtype=int32)>
```

```
#5) Multiply the two tensors you created in 3 using dot product.
# Create (3, 2) tensor
import tensorflow as tf
X = tf.constant([[1, 2],
                 [7, 4],
                 [8, 1]])

# Create another (3, 2) tensor
Y = tf.constant([[2, 8],
                 [12, 10],
                 [6, 12]])
```

```
X, Y
tf.transpose(X)
tf.tensordot(tf.transpose(X), Y, axes=1)
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[134, 174],
       [ 58,  68]], dtype=int32)>
```

```
#6)Create a tensor with random values between 0 and 1 with shape [224, 224, 3].
rank_4_tensor = tf.zeros([224, 224, 3])
rank_4_tensor
import numpy as np
numpy_A = np.arange(0, 1, dtype=np.float16) # create a NumPy array between 0and 1
A = tf.constant(numpy_A,
                shape=[224, 224, 3]) # note: the shape total (2*4*3) has to match the numb
numpy_A, A
```

```
(array([0.], dtype=float16),
 <tf.Tensor: shape=(224, 224, 3), dtype=float16, numpy=
 array([[[0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.],
         ...,
         [0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.]],

        [[0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.],
         ...,
         [0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.]],

        [[0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.],
         ...,
         [0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.]],

        ...,

        [[0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.],
         ...,
         [0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.]],

        [[0., 0., 0.],
         [0., 0., 0.],
         [0., 0., 0.],
         ...,
         [0., 0., 0.],
```

```
                [0., 0., 0.],
                [0., 0., 0.]],

               [[0., 0., 0.],
                [0., 0., 0.],
                [0., 0., 0.],
                ...,
                [0., 0., 0.],
                [0., 0., 0.],
                [0., 0., 0.]]], dtype=float16)>)
```

```
#7) find min max
print(tf.reduce_min(A))
```

```
     tf.Tensor(0.0, shape=(), dtype=float16)
```

```
print(tf.reduce_max(A))
```

```
     tf.Tensor(0.0, shape=(), dtype=float16)
```

```
#8) Created a tensor with random values of shape [1, 224, 224, 3] then squeeze it to chang
# Create a rank 5 (5 dimensions) tensor of 50 numbers between 0 and 100
G = tf.constant(np.random.randint(0, 100, 50), shape=(1, 224,224,3))
G.shape, G.ndim

# Squeeze tensor G (remove all 1 dimensions)
```

Derivatives of Functions with Known Dependency When we talk about the derivative of a function, we are essentially talking about the rate of change of that function with respect to its input. For example, if we have a function f(x) = x^2, the derivative of this function with respect to x is 2x. This means that if we change x by a small amount, the output of the function will change by a proportionally larger amount.

In TensorFlow, we can compute the derivative of a function using the GradientTape API. This API allows us to record the operations that occur during the execution of our function, and then compute the gradient of the function with respect to its input. In FOLLOWING example, we define a TensorFlow variable x with a value of 2.0. We then use a GradientTape context to record the operations that occur during the execution of our function y = x ** 2. Finally, we compute the gradient of y with respect to x using the tape.gradient() method.

```
import tensorflow as tf

x = tf.Variable(5.0)

with tf.GradientTape() as tape:
  #y = 1/(1-exp(-x))
  y=tf.sigmoid(x)

dy_dx = tape.gradient(y, x)

print(dy_dx) # Output: tf.Tensor(4.0, shape=(), dtype=float32)
```

```
tf.Tensor(0.006648033, shape=(), dtype=float32)
```

Derivatives of Functions with Unknown Dependency

Now let's consider the case where the function we want to compute the derivative of has an unknown dependency on the decision variable. In this case, we cannot simply use the GradientTape API to compute the derivative, as we do not know the exact relationship between the decision variable and the function.

For example, let's say we have a function f(x) = g(x) * x, where g(x) is some unknown function of x. We want to compute the derivative of this function with respect to x. In this case, we cannot simply use the GradientTape API, as we do not know the relationship between g(x) and x.

However, we can use a technique called automatic differentiation to compute the derivative of the function. Automatic differentiation is a technique that allows us to compute the derivative of a function by breaking it down into its constituent parts, and then computing the derivative of each part separately. In the example below, we define a TensorFlow variable x with a value of 2.0. We then use two GradientTape contexts to record the operations that occur during the execution of our function y = g(x) * x. The inner tape is used to record the operations that occur

during the execution of g(x), and the outer tape is used to record the operations that occur during the execution of y. Finally, we compute the second derivative of y with respect to x using the tape.gradient() method

```
import tensorflow as tf

x = tf.Variable(2.0)

with tf.GradientTape() as tape:
  with tf.GradientTape() as inner_tape:
    y = g(x) * x
  dy_dx = inner_tape.gradient(y, x)

d2y_dx2 = tape.gradient(dy_dx, x)

print(d2y_dx2) # Output: tf.Tensor(<unknown>, shape=(), dtype=float32)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-6ef188b01890> in <cell line: 5>()
      5 with tf.GradientTape() as tape:
      6   with tf.GradientTape() as inner_tape:
----> 7     y = g(x) * x
      8   dy_dx = inner_tape.gradient(y, x)
      9

NameError: name 'g' is not defined
```

SEARCH STACK OVERFLOW

Each neuron has a set of weights that need to be maintained. **One weight for each input connection and an additional weight for the bias**. We will need to store additional properties for a neuron during training, therefore we will use **a dictionary to represent each neuron and store properties by names such as 'weights' for the weights**.

A network is organized into layers. The input layer is really just a row from our training dataset. The first real layer is the hidden layer. This is followed by the output layer that has one neuron for each class value.

It is good practice to initialize the network weights to small random numbers. In this case, will we use random numbers in the range of 0 to 1.

Below is a function named initialize_network() that creates a new neural network ready for training. It accepts three parameters, the number of inputs, the number of neurons to have in the hidden layer and the number of outputs.

You can see that for the hidden layer we create n_hidden neurons and each neuron in the hidden layer has n_inputs + 1 weights, one for each input column in a dataset and an additional one for the bias.

You can also see that the output layer that connects to the hidden layer has n_outputs neurons, each with n_hidden + 1 weights. This means that each neuron in the output layer connects to (has a weight for) each neuron in the hidden layer.

```
# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    return network
```

```
network
```

```
[[{'weights': [-1.482313569067226, 1.8308790073202204, 1.078381922048799],
   'output': 0.020376679405025657},
  {'weights': [0.23244990332399884, 0.3621998343835864, 0.40289821191094327],
   'output': 0.9694529285797023}],
 [{'weights': [2.5001872433501404, 0.7887233511355132, -1.1026649757805829],
   'output': 0.428716603655823},
  {'weights': [-2.429350576245497, 0.8357651039198697, 1.0699217181280656],
   'output': 0.8618396836408665}]]
```

```
from random import seed
from random import random

seed(1)
network = initialize_network(2, 1, 2)
for layer in network:
    print(layer)
```

```
[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}]
[{'weights': [0.2550690257394217, 0.49543508709194095]}, {'weights': [0.4494910647887381, 0.651592972722763]}]
```

We can break forward propagation down into three parts:

Neuron Activation. Neuron Transfer. Forward Propagation.

```
#Neuron Activation
# Calculate neuron activation for an input
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation
```

Neuron Transfer

The sigmoid activation function looks like an S shape, it's also called the logistic function. It can take any input value and produce a number between 0 and 1 on an S-curve. It is also a function of which we can easily calculate the derivative (slope) that we will need later when backpropagating error.

```
# Transfer neuron activation
def transfer(activation):
  return 1.0 / (1.0 + exp(-activation))
```

Forward Propagation

We work through each layer of our network calculating the outputs for each neuron. All of the outputs from one layer become inputs to the neurons on the next layer.

```
# Forward propagate input to a network output
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
             activation = activate(neuron['weights'], inputs)
             neuron['output'] = transfer(activation)
             new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs
```

```
from math import exp
# test forward propagation
network = [[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}],
         [{'weights': [0.2550690257394217, 0.4954350870919409 5]}, {'weights': [0.4494910647887381, 0.651592972722763]}]]
row = [1, 0, None]
output = forward_propagate(network, row)
print(output)
```

```
     [0.6629970129852887, 0.7253160725279748]
```

Running the example propagates the input pattern [1, 0] and produces an output value that is printed. Because the output layer has two neurons, we get a list of two numbers as output.

The actual output values are just nonsense for now, but next, we will start to learn how to make the weights in the neurons more useful.

Back Propagate Error The backpropagation algorithm is named for the way in which weights are trained.

Error is calculated between the expected outputs and the outputs forward propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go.

This part is broken down into two sections.

Transfer Derivative.

Error Backpropagation.

Below is a function named backward_propagate_error() that implements this procedure.

You can see that the error signal calculated for each neuron is stored with the name ‘delta’. You can see that the layers of the network are iterated in reverse order, starting at the output and working backwards. This ensures that the neurons in the output layer have ‘delta’ values calculated first that neurons in the hidden layer can use in the subsequent iteration. I chose the name ‘delta’ to reflect the change the error implies on the neuron (e.g. the weight delta).

You can see that the error signal for neurons in the hidden layer is accumulated from neurons in the output layer where the hidden neuron number j is also the index of the neuron’s weight in the output layer neuron[‘weights’][j].

```
# Calculate the derivative of an neuron output
def transfer_derivative(output):
     return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
     for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network)-1:
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += (neuron['weights'][j] * neuron['delta'])
                errors.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(expected[j] - neuron['output'])
        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])


# test backpropagation of error
network = [[{'output': 0.7105668883115941, 'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}],
        [{'output': 0.6213859615555266, 'weights': [0.2550690257394217, 0.49543508709194095]}, {'output': 0.6573693455986976, 'weights':
expected = [0, 1]
backward_propagate_error(network, expected)
for layer in network:
    print(layer)
```

```
[{'output': 0.7105668883115941, 'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614], 'delta': -0.00053480480466
[{'output': 0.6213859615555266, 'weights': [0.2550690257394217, 0.49543508709194095], 'delta': -0.14619064683582808}, {'output': 0.
```

Train Network As mentioned, the network is updated using stochastic gradient descent.

This involves first looping for a fixed number of epochs and within each epoch updating the network for each row in the training dataset.

Because updates are made for each training pattern, this type of learning is called online learning. If errors were accumulated across an epoch before updating the weights, this is called batch learning or batch gradient descent.

Below is a function that implements the training of an already initialized neural network with a given training dataset, learning rate, fixed number of epochs and an expected number of output values.

The expected number of output values is used to transform class values in the training data into a one hot encoding. That is a binary vector with one column for each class value to match the output of the network. This is required to calculate the error for the output layer.

You can also see that the sum squared error between the expected output and the network output is accumulated each epoch and printed. This is helpful to create a trace of how much the network is learning and improving each epoch.

```
# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
    for epoch in range(n_epoch):
        sum_error = 0
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            sum_error += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))


# Update network weights with error
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] += l_rate * neuron['delta']

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
    for epoch in range(n_epoch):
        sum_error = 0
```

```
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            sum_error += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))

# Test training backprop algorithm
seed(1)
dataset = [[2.7810836,2.550537003,0],
    [1.465489372,2.362125076,0],
    [3.396561688,4.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [7.627531214,2.759262235,1],
    [5.332441248,2.088626775,1],
    [6.922596716,1.77106367,1],
    [8.675418651,-0.242068655,1],
    [7.673756466,3.508563011,1]]
n_inputs = len(dataset[0]) - 1
n_outputs = len(set([row[-1] for row in dataset]))
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
    print(layer)
```

```
>epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.221
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output': 0.029980305604426185, 'delta': -0.005954660416
[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output': 0.23648794202357587, 'delta': -0.042700592783
```

Predict

Making predictions with a trained neural network is easy enough.

```
# Make a prediction with a network
def predict(network, row):
  outputs = forward_propagate(network, row)
  return outputs.index(max(outputs))

# Test making predictions with the network
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
network = [[{'weights': [-1.482313569067226, 1.8308790073202204, 1.078381922048799]}, {'weights': [0.23244990332399884, 0.362199834383586
  [{'weights': [2.5001872433501404, 0.7887233511355132, -1.1026649757805829]}, {'weights': [-2.429350576245497, 0.8357651039198697, 1.069
for row in dataset:
  prediction = predict(network, row)
  print('Expected=%d, Got=%d' % (row[-1], prediction))
```

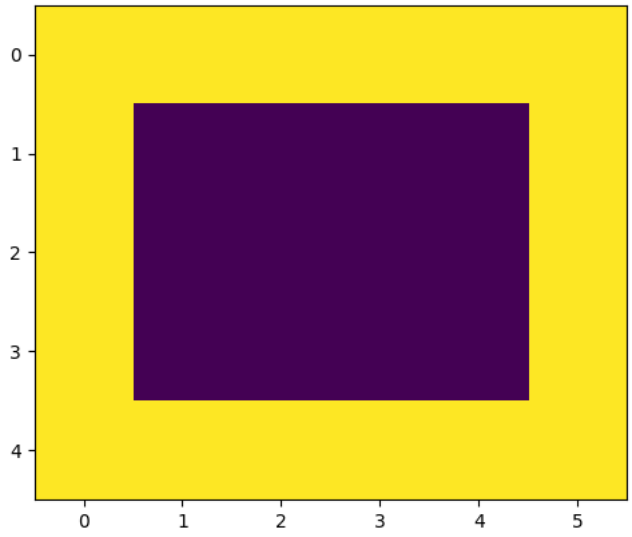
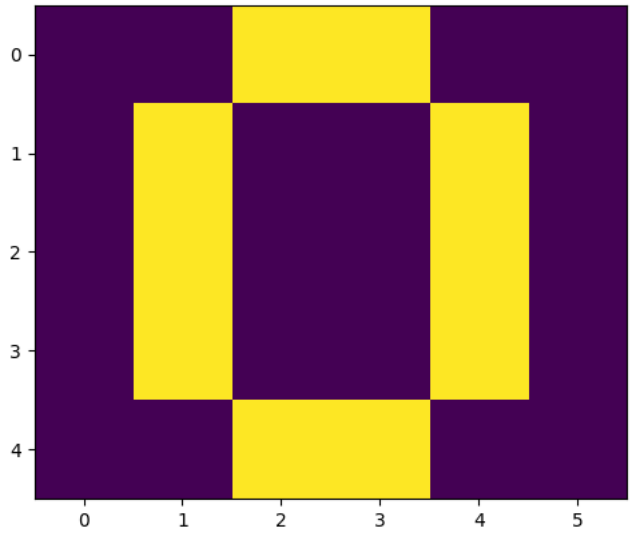
```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
```

```
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```

ANN FROM SCRACH

```
# Creating data set

# A square
a =[1, 1, 1, 1, 1,1 ,
   1, 0, 0, 0, 0, 1,
   1, 0, 0, 0, 0, 1,
   1, 0, 0, 0, 0, 1,
   1, 1, 1, 1, 1, 1]
# B circle
b =[0, 0, 1, 1, 0, 0,
   0, 1, 0, 0, 1, 0,
   0, 1, 0, 0, 1, 0,
   0, 1, 0, 0, 1, 0,
   0, 0, 1, 1, 0, 0]
# C  Triangle
c =[0, 0, 1, 1, 0, 0,
   0, 1, 1, 1, 1, 0,
   0, 1, 1, 1, 1, 0,
   1, 1, 1, 1, 1, 1,
   0, 0, 0, 0, 0, 0]
  #D dimond adding extra shape
d=[1, 1, 0, 0, 1,1,
   1, 0, 0, 0, 0,1,
   0, 0, 0, 0, 0,0,
   1, 0, 0, 0, 0,1,
   1, 1, 0, 0, 1,1]
# Creating labels
y =[[1, 0, 0,0],
   [0, 1, 0,0],
   [0, 0, 1,0],
    [0,0,0,1]]
```

```
import numpy as np
import matplotlib.pyplot as plt
# visualizing the data, plotting A.
plt.imshow(np.array(a).reshape(5, 6))
plt.show()
plt.imshow(np.array(b).reshape(5, 6))
plt.show()
plt.imshow(np.array(c).reshape(5, 6))
plt.show()
plt.imshow(np.array(d).reshape(5, 6))
plt.show()
```


```
# converting data and labels into numpy array

"""
Convert the matrix of 0 and 1 into one hot vector
so that we can directly feed it to the neural network,
these vectors are then stored in a list x.
"""

x =[np.array(a).reshape(1, 30), np.array(b).reshape(1, 30),
                                np.array(c).reshape(1, 30),np.array(d).reshape(1, 30)]


# Labels are also converted into NumPy array
y = np.array(y)


print(x, "\n\n", y)
```

```
     [array([[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0,
             0, 1, 1, 1, 1, 1, 1, 1]]), array([[0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0,
             1, 0, 0, 0, 1, 1, 0, 0]]), array([[0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1,
             1, 1, 0, 0, 0, 0, 0, 0]]), array([[1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
             0, 1, 1, 1, 0, 0, 1, 1]])]

      [[1 0 0 0]
```

```
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]]
```

1st layer: Input layer(1, 30) 2nd layer: Hidden layer (1, 5) 3rd layer: Output layer(3, 3)

```
# activation function

def sigmoid(x):
    return(1/(1 + np.exp(-x)))

# Creating the Feed forward neural network
# 1 Input layer(1, 30)
# 1 hidden layer (1, 5)
# 1 output layer(3, 3)

def f_forward(x, w1, w2):
    # hidden
    z1 = x.dot(w1)# input from layer 1
    a1 = sigmoid(z1)# out put of layer 2

    # Output layer
    z2 = a1.dot(w2)# input of out layer
    a2 = sigmoid(z2)# output of out layer
    return(a2)

# initializing the weights randomly
def generate_wt(x, y):
    l =[]
    for i in range(x * y):
        l.append(np.random.randn())
    return(np.array(l).reshape(x, y))

# for loss we will be using mean square error(MSE)
def loss(out, Y):
    s =(np.square(out-Y))
    s = np.sum(s)/len(y)
    return(s)

# Back propagation of error
def back_prop(x, y, w1, w2, alpha):

    # hidden layer
    z1 = x.dot(w1)# input from layer 1
    a1 = sigmoid(z1)# output of layer 2

    # Output layer
    z2 = a1.dot(w2)# input of out layer
    a2 = sigmoid(z2)# output of out layer
    # error in output layer
    d2 =(a2-y)
    d1 = np.multiply((w2.dot((d2.transpose()))).transpose(),
                                  (np.multiply(a1, 1-a1)))

    # Gradient for w1 and w2
    w1_adj = x.transpose().dot(d1)
    w2_adj = a1.transpose().dot(d2)

    # Updating parameters
    w1 = w1-(alpha*(w1_adj))
    w2 = w2-(alpha*(w2_adj))

    return(w1, w2)

def train(x, Y, w1, w2, alpha = 0.01, epoch = 10):
    acc =[]
    losss =[]
    for j in range(epoch):
        l =[]
        for i in range(len(x)):
            out = f_forward(x[i], w1, w2)
            l.append((loss(out, Y[i])))
            w1, w2 = back_prop(x[i], y[i], w1, w2, alpha)
        print("epochs:", j + 1, "======== acc:", (1-(sum(l)/len(x)))*100)
        acc.append((1-(sum(l)/len(x)))*100)
        losss.append(sum(l)/len(x))
    return(acc, losss, w1, w2)

def predict(x, w1, w2):
    Out = f_forward(x, w1, w2)
    maxm = 0
```

```
    k = 0
    for i in range(len(Out[0])):
        if(maxm<Out[0][i]):
            maxm = Out[0][i]
            k = i
    if(k == 0):
        print("Image is of square.")
    elif(k == 1):
        print("Image is of circle.")
    elif(k>=0.5):
        print("Image is of triangle.")

    else:
      print("Image is of dimond.")
    plt.imshow(x.reshape(5, 6))
    plt.show()


w1 = generate_wt(30, 5)
w2 = generate_wt(5, 3)
print(w1, "\n\n", w2)
```

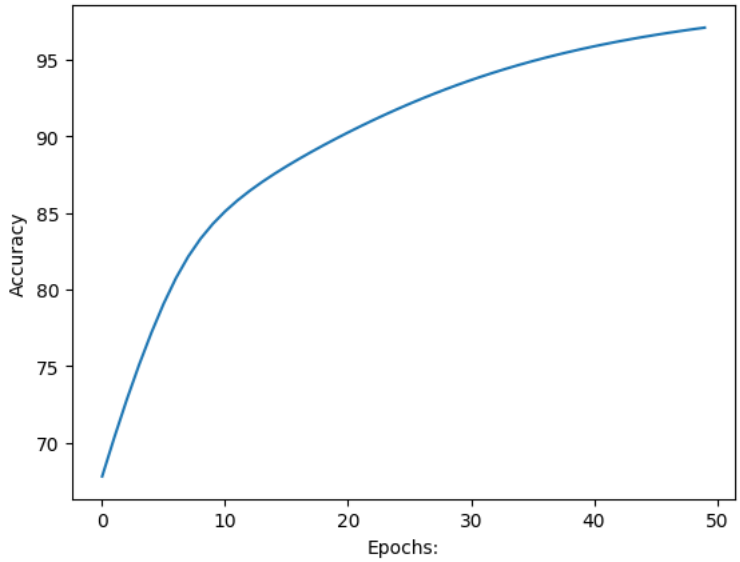
```
[[-0.86495353 -0.84829143  0.31751784 -1.23572017  1.30801979]
 [ 0.68689693  0.62538607 -0.23323806 -0.27862661 -0.33775247]
 [-0.70339688  2.92376131  1.30158613 -1.23865172 -0.72214532]
 [ 1.25039264 -0.50051996 -0.7445227  -1.40241077  0.43318186]
 [ 1.0082066  -0.29566707 -0.50546627 -0.24132458  1.93809415]
 [ 0.77913775  0.94102103  0.32355953 -0.00956481 -0.15609521]
 [ 1.33842068  0.65568849 -0.34249944  0.57989039  1.26672596]
 [ 0.1017321  -1.27411112  0.68210177 -0.05499193  0.73581596]
 [ 0.43049858  0.72793033  0.13408037  0.24473703  1.89495099]
 [-0.95569066 -1.47624416  1.26775469 -0.30666848  0.20804923]
 [ 1.43744684 -2.32341975  0.54526449 -0.17584259  0.21134674]
 [-0.83782574 -0.54578178 -0.21071799 -0.7695837  -0.87123538]
 [ 1.16578575  0.36653943 -1.13015282  0.2540849  -0.56731577]
 [ 0.37073985  1.22598488 -0.34228332 -0.05833151 -1.88445837]
 [ 0.73837738 -1.54348994 -0.01938104  3.12407205  1.22132272]
 [-0.31613349  0.04789013  0.31609038  0.02114192  0.09733676]
 [-0.3293131   0.69818086 -0.19829948 -1.45224913 -0.23137078]
 [-0.32460035  0.36805192  2.29232401  0.57735346 -0.03294765]
 [ 1.72993729  0.95282512 -0.43365488 -0.50513398 -0.73211354]
 [ 0.52246059 -0.94915522 -0.89073871 -2.35761061  1.47768721]
 [-2.1869994  -0.17609375 -0.14915876  1.81603722 -0.28989623]
 [-1.10650657 -1.12613    -0.47731727  0.10812379 -0.95300264]
 [ 1.02415535  0.97566215 -0.11063741  1.676416    1.0386056 ]
 [-1.06163095 -1.48095722  0.71143625  1.10433489  0.03782481]
 [-1.25456322 -0.07458237 -0.04697822 -1.2501741   0.50149769]
 [-1.22053789 -1.50058419 -0.77649216  0.45051558  1.4392072 ]
 [ 1.45403923 -0.43461769  0.52075743  0.23794673  2.08069555]
 [-0.71147899  1.16363517 -0.94618463 -1.34760056 -0.76995058]
 [ 0.02846693 -0.26993837  0.57525762  0.4628354   0.46330609]
 [-0.64850816 -0.92047866  1.68123018 -2.11668345  0.07908233]]

 [[-0.11619356  1.7298068   0.56383927]
 [ 0.80520538  0.38566027 -0.73094984]
 [ 1.39738823 -0.67822317  0.86511743]
 [ 0.80715502 -0.46618482  0.00348079]
 [-0.16728394  0.08944146  0.62477454]]
```

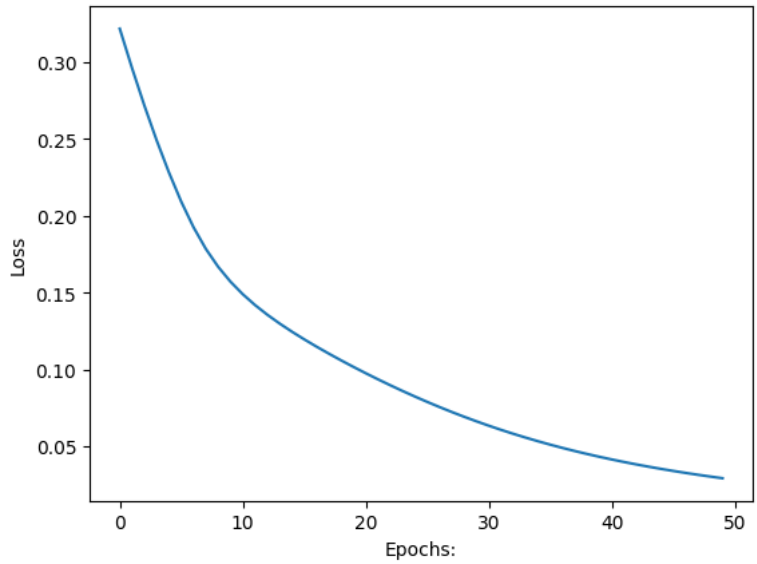
```
"""The arguments of train function are data set list x,
correct labels y, weights w1, w2, learning rate = 0.1,
no of epochs or iteration.The function will return the
matrix of accuracy and loss and also the matrix of
trained weights w1, w2"""

acc, losss, w1, w2 = train(x, y, w1, w2, 0.1, 50)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-47-10888ee515b5> in <cell line: 7>()
      5 trained weights w1, w2"""
      6
----> 7 acc, losss, w1, w2 = train(x, y, w1, w2, 0.1, 50)

                                 1 frames
<ipython-input-45-232f8e85d00f> in loss(out, Y)
     28 # for loss we will be using mean square error(MSE)
     29 def loss(out, Y):
---> 30     s =(np.square(out-Y))
     31     s = np.sum(s)/len(y)
     32     return(s)

ValueError: operands could not be broadcast together with shapes (1,3) (4,)
```

SEARCH STACK OVERFLOW

```
import matplotlib.pyplot as plt1

# plotting accuracy
plt1.plot(acc)
plt1.ylabel('Accuracy')
plt1.xlabel("Epochs:")
plt1.show()

# plotting Loss
plt1.plot(losss)
plt1.ylabel('Loss')
plt1.xlabel("Epochs:")
plt1.show()
```



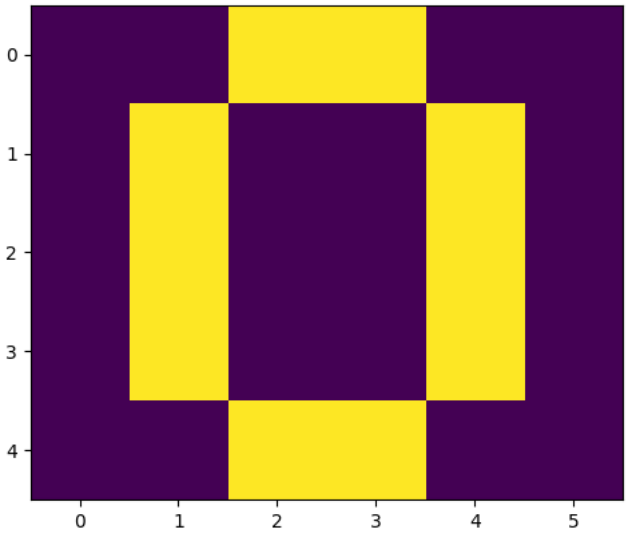



```
"""
The predict function will take the following arguments:
1) image matrix
2) w1 trained weights
3) w2 trained weights
"""
predict(x[2], w1, w2)
```

```
Image is of triangle.
```

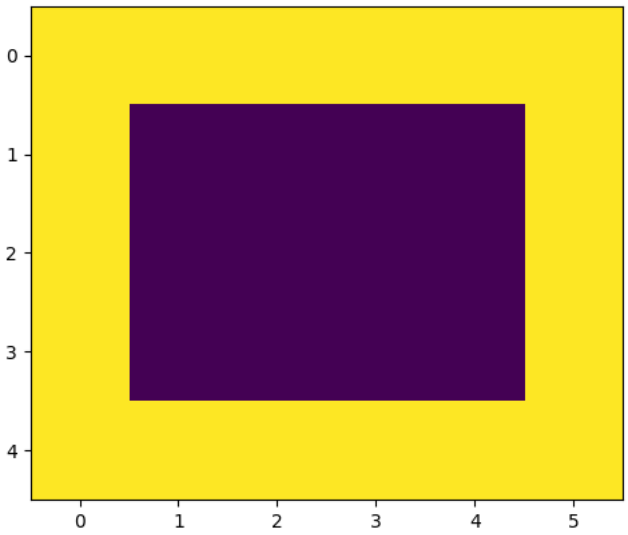
```
"""
The predict function will take the following arguments:
1) image matrix
2) w1 trained weights
3) w2 trained weights
"""
predict(x[1], w1, w2)
```

```
Image is of circle.
```



```
"""
The predict function will take the following arguments:
1) image matrix
2) w1 trained weights
3) w2 trained weights
"""
predict(x[0], w1, w2)
```

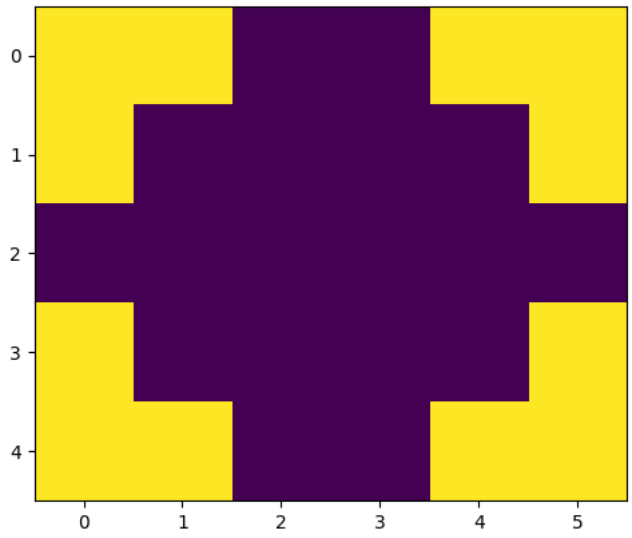
```
Image is of square.
```



```
"""
The predict function will take the following arguments:
1) image matrix
2) w1 trained weights
3) w2 trained weights
```

```
"""
predict(x[3], w1, w2)
```

```
Image is of triangle.
```



✓ 2s completed at 6:12 PM

```
# Modules used for data handling and linear algebra operations.
import pandas as pd
import numpy as np

# Modules used for data visualization
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style()

# Modules used for encoding the categorical variables.
from sklearn.preprocessing import OneHotEncoder


from google.colab import files


uploaded = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving exp3_data.csv to exp3_data.csv

```
df = pd.read_csv("exp3_data.csv",header=None )



df = df.replace({"?":None})
df = df.dropna()
df[1]=df[1].astype(float)
df[13]=df[13].astype(float)


df.head()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	b	30.83	0.000	u	g	w	v	1.25	t	t	1.0	f	g	202.0	0.0	+
1	a	58.67	4.460	u	g	q	h	3.04	t	t	6.0	f	g	43.0	560.0	+
2	a	24.50	0.500	u	g	q	h	1.50	t	f	0.0	f	g	280.0	824.0	+
3	b	27.83	1.540	u	g	w	v	3.75	t	t	5.0	t	g	100.0	3.0	+
4	b	20.17	5.625	u	g	w	v	1.71	t	f	0.0	f	s	120.0	0.0	+

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 653 entries, 0 to 689
Data columns (total 16 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       653 non-null    object 
 1   1       653 non-null    float64
 2   2       653 non-null    float64
 3   3       653 non-null    object 
 4   4       653 non-null    object 
 5   5       653 non-null    object 
 6   6       653 non-null    object 
 7   7       653 non-null    float64
 8   8       653 non-null    object 
 9   9       653 non-null    object 
 10  10      653 non-null    float64
 11  11      653 non-null    object 
 12  12      653 non-null    object 
 13  13      653 non-null    float64
 14  14      653 non-null    float64
 15  15      653 non-null    object 
dtypes: float64(6), object(10)
memory usage: 86.7+ KB
```

Exploratory Data Analysis

In Exploratory Data Analysis the following parts are included:

Seggragation of columns (into categorical and numerical) Analysis of Missing Values Target Variable Class Distribution Seggregating columns The columns with data type as Object are considered as categorical while others are considered as numerical.

```
cat_cols = []
num_cols = []

for i in df.columns:
    if df[i].dtype == "O":
        cat_cols.append(i)
    else:
        num_cols.append(i)
```

```
cat_cols
```

```
[0, 3, 4, 5, 6, 8, 9, 11, 12, 15]
```

```
num_cols
```

```
[1, 2, 7, 10, 13, 14]
```

```
df.shape
```

```
(653, 16)
```

Missing Values Analysis

Since, the UCI data repository mentions missing values as "?" instead of null values the analysis has to be carried out accordingly.

```
df = df.replace({"?":None})
df = df.dropna()
```

```
df.shape
```

```
(653, 17)
```

```
df
```

	index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	202.0	0.0	0.000	0.0	0.0	1.0	0.0	1.25	0.0	0.0	1.0	0.0	1.0	30.83	0.0	1.0
1	1	43.0	0.0	4.460	1.0	0.0	1.0	0.0	3.04	0.0	0.0	6.0	0.0	1.0	58.67	560.0	1.0
2	2	280.0	0.0	0.500	1.0	0.0	1.0	0.0	1.50	0.0	0.0	0.0	1.0	1.0	24.50	824.0	1.0
3	3	100.0	0.0	1.540	0.0	0.0	1.0	0.0	3.75	0.0	0.0	5.0	0.0	0.0	27.83	3.0	1.0
4	4	120.0	0.0	5.625	0.0	0.0	1.0	0.0	1.71	0.0	0.0	0.0	1.0	1.0	20.17	0.0	1.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
648	685	260.0	1.0	10.085	0.0	0.0	0.0	0.0	1.25	0.0	1.0	0.0	1.0	1.0	21.08	0.0	0.0
649	686	200.0	1.0	0.750	1.0	0.0	1.0	0.0	2.00	0.0	1.0	2.0	0.0	0.0	22.67	394.0	0.0
650	687	200.0	1.0	13.500	1.0	0.0	0.0	0.0	2.00	0.0	1.0	1.0	0.0	0.0	25.25	1.0	0.0
651	688	280.0	1.0	0.205	0.0	0.0	1.0	1.0	0.04	0.0	1.0	0.0	1.0	1.0	17.92	750.0	0.0
652	689	0.0	1.0	3.375	0.0	0.0	1.0	0.0	8.29	0.0	1.0	0.0	1.0	0.0	35.00	0.0	0.0

653 rows × 17 columns

Encoding the columns

```
encoder = OneHotEncoder(sparse=False)
for i in cat_cols:
    df[i] = encoder.fit_transform(df[i].values.reshape(-1,1))
print(df[3])
```

```
0      1.0
1      0.0
2      0.0
3      1.0
4      1.0
      ...
648    1.0
649    0.0
650    0.0
651    1.0
652    1.0
```

```
Name: 3, Length: 653, dtype: float64
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
```

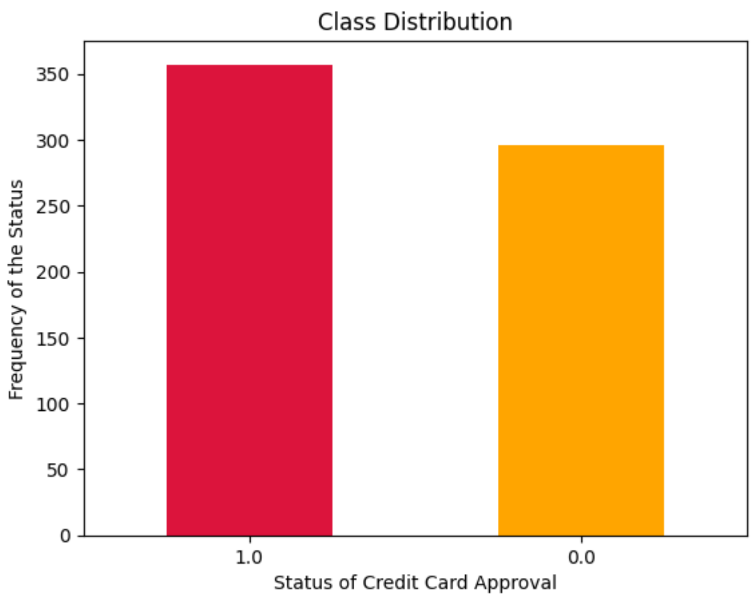
```
df = df.reset_index()
```

```
df
```

	level_0	index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0.0	0.0	0.000	1.0	1.0	0.0	1.0	1.25	1.0	1.0	1.0	1.0	0.0	30.83	0.0	0.0
1	1	1	0.0	0.0	4.460	0.0	1.0	0.0	1.0	3.04	1.0	1.0	6.0	1.0	0.0	58.67	560.0	0.0
2	2	2	0.0	0.0	0.500	0.0	1.0	0.0	1.0	1.50	1.0	1.0	0.0	0.0	0.0	24.50	824.0	0.0
3	3	3	0.0	0.0	1.540	1.0	1.0	0.0	1.0	3.75	1.0	1.0	5.0	1.0	1.0	27.83	3.0	0.0
4	4	4	0.0	0.0	5.625	1.0	1.0	0.0	1.0	1.71	1.0	1.0	0.0	0.0	0.0	20.17	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
648	648	685	0.0	1.0	10.085	1.0	1.0	1.0	1.0	1.25	1.0	0.0	0.0	0.0	0.0	21.08	0.0	1.0
649	649	686	0.0	1.0	0.750	0.0	1.0	0.0	1.0	2.00	1.0	0.0	2.0	1.0	1.0	22.67	394.0	1.0
650	650	687	0.0	1.0	13.500	0.0	1.0	1.0	1.0	2.00	1.0	0.0	1.0	1.0	1.0	25.25	1.0	1.0
651	651	688	0.0	1.0	0.205	1.0	1.0	0.0	0.0	0.04	1.0	0.0	0.0	0.0	0.0	17.92	750.0	1.0
652	652	689	1.0	1.0	3.375	1.0	1.0	0.0	1.0	8.29	1.0	0.0	0.0	0.0	1.0	35.00	0.0	1.0

653 rows × 18 columns

```
df[15].value_counts().plot(kind="bar",
                           title="Class Distribution",
                           xlabel="Status of Credit Card Approval",
                           ylabel="Frequency of the Status",
                           color=["crimson","orange"],
                           rot=0)
plt.show()
```



Test Train Split

```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(df,
    df[15],
    test_size = 0.10,
    train_size=0.90,
    random_state = 0
)
X_train.pop(15)
X_test.pop(15)
```

```
535    0.0
492    0.0
14     0.0
247    1.0
85     1.0
      ...
506    1.0
266    1.0
155    0.0
403    1.0
18     0.0
Name: 15, Length: 66, dtype: float64
```

```
X_train
```

	level_0	index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
436	436	459	0.0	1.0	0.290	1.0	1.0	1.0	1.0	1.500	1.0	0.0	0.0	0.0	1.0	25.67	0.0
75	75	76	0.0	1.0	6.500	1.0	1.0	0.0	0.0	0.125	1.0	1.0	0.0	0.0	1.0	34.08	0.0
55	55	55	0.0	0.0	11.625	1.0	1.0	1.0	1.0	0.835	1.0	1.0	0.0	0.0	1.0	23.33	300.0
49	49	49	0.0	0.0	0.665	1.0	1.0	0.0	1.0	0.165	1.0	0.0	0.0	0.0	0.0	23.92	0.0
589	589	623	0.0	1.0	0.000	1.0	1.0	0.0	1.0	0.665	1.0	0.0	0.0	0.0	0.0	18.83	1.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
9	9	9	0.0	0.0	4.915	1.0	1.0	1.0	1.0	3.165	1.0	1.0	0.0	0.0	1.0	42.50	1442.0
359	359	377	0.0	1.0	0.835	1.0	1.0	1.0	1.0	2.000	1.0	0.0	0.0	0.0	1.0	20.67	0.0
192	192	197	1.0	0.0	7.625	1.0	1.0	0.0	1.0	15.500	1.0	1.0	12.0	1.0	0.0	48.17	790.0
629	629	665	1.0	1.0	0.040	1.0	1.0	1.0	1.0	0.040	1.0	0.0	0.0	0.0	0.0	31.83	0.0
559	559	588	0.0	0.0	1.750	1.0	1.0	1.0	1.0	1.000	1.0	1.0	5.0	1.0	1.0	26.67	5777.0

587 rows × 17 columns

Perceptron

```
class Perceptron:

    # Initialising the required parameters for the perceptron.
    def __init__(self, X, y, learning_rate, epochs : int):
        self.X = X
        self.y = y
        self.learning_rate = learning_rate
        self.epochs = epochs

    # Activation function.
    def __activation_function(self,x):
        return 1.0 if (x > 0) else 0.0

    # The model training or fitting by updating weights.
    def fit(self):
        n_rows,n_cols = self.X.shape
        self.weights = np.zeros((n_cols + 1, 1))
        for epoch in range(self.epochs):
            for index, features in enumerate(self.X.values):
                feature_transposed = np.insert(features, 0, 1).reshape(-1,1)
                predicted_target = self.__activation_function(np.dot(feature_transposed.T, self.weights))
                flag = np.squeeze(predicted_target) - self.y[index]
                if flag != 0:
                    self.weights += self.learning_rate*((self.y[index] - predicted_target)*feature_transposed)
```

```
    # Predicting on a single instance.
    def predict(self, X_test):
        return self.__activation_function(np.dot(p.weights.reshape(1,-1)[0],X_test))

    # Predicting on a larger number of instances and returning accuracy.
    def test(self, test_data, y):
        x = []
        for i in range(len(test_data.values)):
            X_test = np.array(test_data.iloc[i])
            x.append(p.predict(np.insert(X_test,0,1)) == p.y[i])
        return sum(x)*100/len(test_data)
```

Initialising the Perceptron Parameters

```
y = np.array(pd.DataFrame(y_train).reset_index().drop(["index"],axis=1))
X = pd.DataFrame(X_train).reset_index().drop(["index"],axis=1)
p = Perceptron(X, y, 0.01, 100)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-49-d64ccdcea403> in <cell line: 2>()
      1 y = np.array(pd.DataFrame(y_train).reset_index().drop(["index"],axis=1))
----> 2 X = pd.DataFrame(X_train).reset_index().drop(["index"],axis=1)
      3 p = Perceptron(X, y, 0.01, 100)

                                  2 frames
/usr/local/lib/python3.10/dist-packages/pandas/core/frame.py in insert(self, loc, column, value,
allow_duplicates)
   4815         if not allow_duplicates and column in self.columns:
   4816             # Should this be a different kind of error??
-> 4817             raise ValueError(f"cannot insert {column}, already exists")
   4818         if not isinstance(loc, int):
   4819             raise TypeError("loc must be int")

ValueError: cannot insert level_0, already exists
```

SEARCH STACK OVERFLOW

Fitting the model

```
p.fit()
```

Test Accuracy

```
p.test(pd.DataFrame(X_test).reset_index().drop(["index"],axis=1),
     np.array(pd.DataFrame(y_test).reset_index().drop(["index"],axis=1)))
```

```
array([53.03030303])
```

Train Accuracy

```
train_acc =p.test(X,y)
train_acc
```

```
array([74.78705281])
```

Tuning the number of epochs

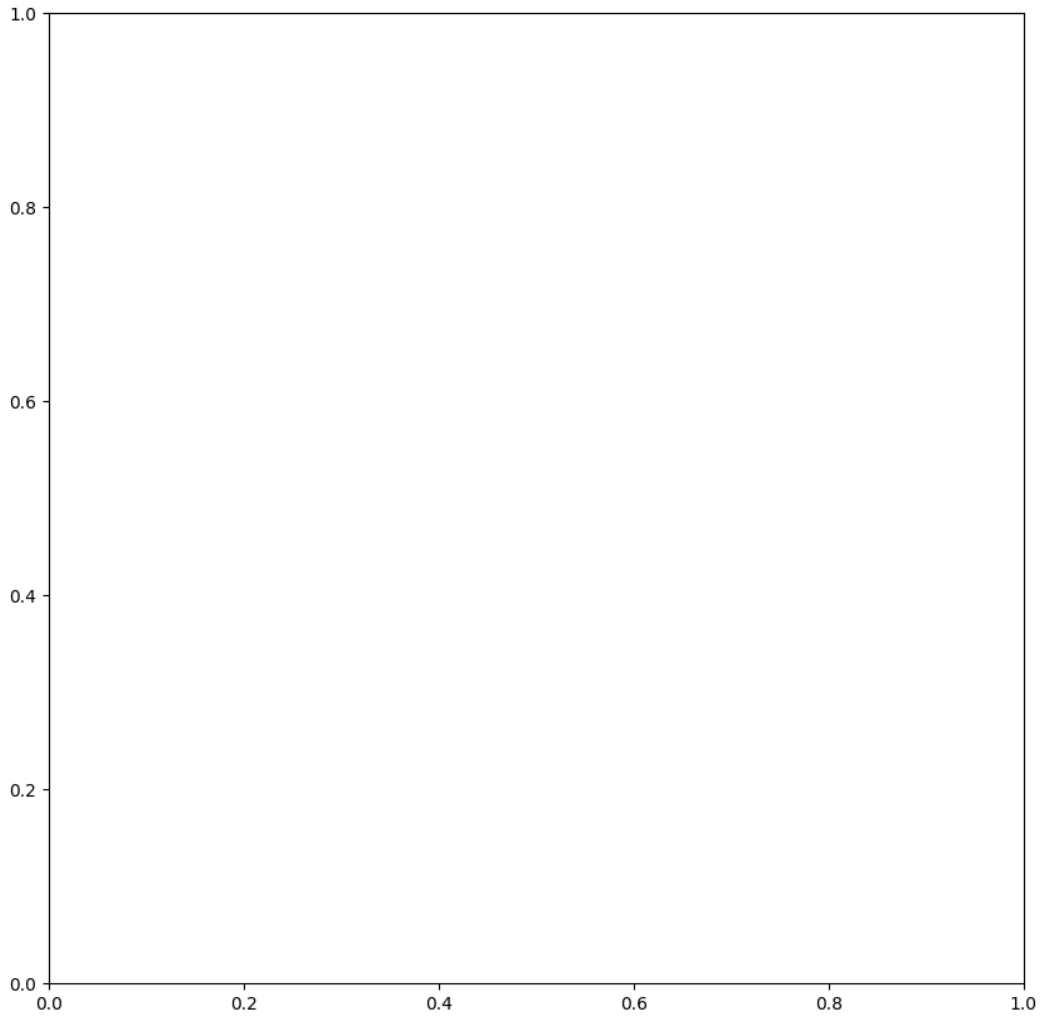
Train-Test Accuracies across different epochs

```
plt.figure(figsize=(10,10))
plt.plot(epochs,train_acc)
plt.plot(epochs,test_acc)
plt.xlabel("Number of Epochs")
plt.ylabel("Accuracy of the model")
```

```
plt.legend(['Train Accuracy',"Test Accuracy"])
plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-37-67f79ab2423f> in <cell line: 2>()
      1 plt.figure(figsize=(10,10))
----> 2 plt.plot(epochs,train_acc)
      3 plt.plot(epochs,test_acc)
      4 plt.xlabel("Number of Epochs")
      5 plt.ylabel("Accuracy of the model")

                                3 frames
/usr/local/lib/python3.10/dist-packages/matplotlib/axes/_base.py in _plot_args(self, tup, kwargs,
return_kwargs, ambiguous_fmt_datakey)
    502
    503         if x.shape[0] != y.shape[0]:
--> 504             raise ValueError(f"x and y must have same first dimension, but "
    505                              f"have shapes {x.shape} and {y.shape}")
    506         if x.ndim > 2 or y.ndim > 2:

ValueError: x and y must have same first dimension, but have shapes (20,) and (1,)
```

SEARCH STACK OVERFLOW



Neural network as per given specifications

```
import keras
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(16, input_dim=15, activation='relu'))
model.add(Dense(4, activation='relu'))

model.add(Dense(1, activation='sigmoid'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 16)                256

 dense_1 (Dense)             (None, 4)                 68

 dense_2 (Dense)             (None, 1)                 5

=================================================================
Total params: 329
Trainable params: 329
Non-trainable params: 0
_________________________________________________________________
```

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
X_test
```

	index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
535	564	0.0	0.0	5.04	0.0	0.0	1.0	0.0	12.750	0.0	0.0	0.0	1.0	0.0	1.0	0.0
492	519	0.0	0.0	1.71	0.0	0.0	1.0	0.0	0.125	0.0	0.0	5.0	0.0	0.0	1.0	0.0
14	14	0.0	1.0	10.50	0.0	0.0	1.0	0.0	5.000	0.0	0.0	7.0	0.0	0.0	1.0	0.0
247	257	1.0	0.0	0.00	0.0	0.0	1.0	0.0	0.500	0.0	1.0	0.0	1.0	1.0	1.0	0.0
85	88	1.0	0.0	4.50	0.0	0.0	1.0	1.0	1.000	0.0	0.0	0.0	1.0	0.0	1.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
506	534	1.0	0.0	2.50	0.0	0.0	1.0	1.0	7.500	0.0	0.0	0.0	1.0	0.0	1.0	0.0
266	277	1.0	1.0	10.00	0.0	0.0	0.0	0.0	0.165	0.0	1.0	0.0	1.0	1.0	1.0	0.0
155	160	0.0	0.0	2.00	0.0	0.0	1.0	0.0	1.000	0.0	0.0	4.0	0.0	1.0	1.0	7544.0
403	422	1.0	0.0	1.25	0.0	0.0	1.0	0.0	1.750	0.0	1.0	0.0	1.0	1.0	1.0	0.0
18	18	0.0	0.0	0.25	0.0	0.0	1.0	0.0	0.665	0.0	0.0	0.0	1.0	0.0	1.0	0.0

66 rows × 16 columns

```
X_test=X_test.drop(['index'], axis=1)
```

```
X_train=X_train.drop(['index'], axis=1)
```

```
y_train
```

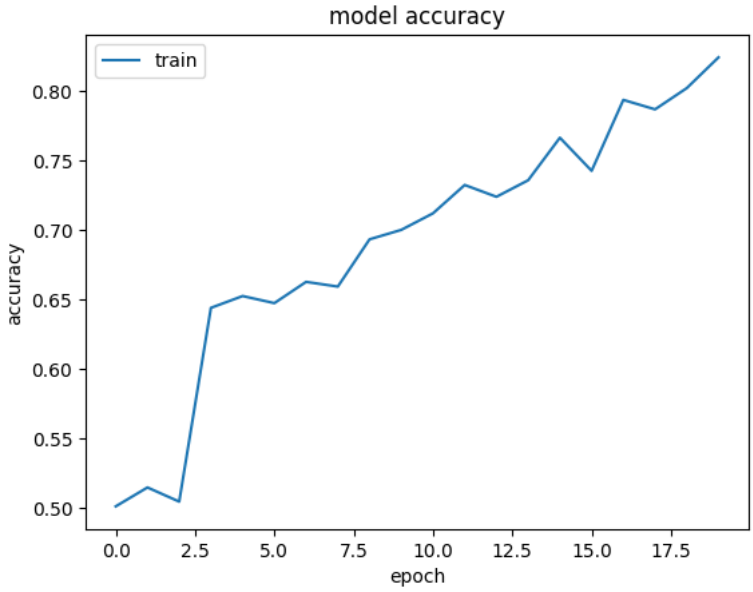
```
436    0.0
75     0.0
55     1.0
49     1.0
589    0.0
      ...
9      1.0
359    0.0
192    1.0
629    0.0
559    1.0
Name: 15, Length: 587, dtype: float64
```

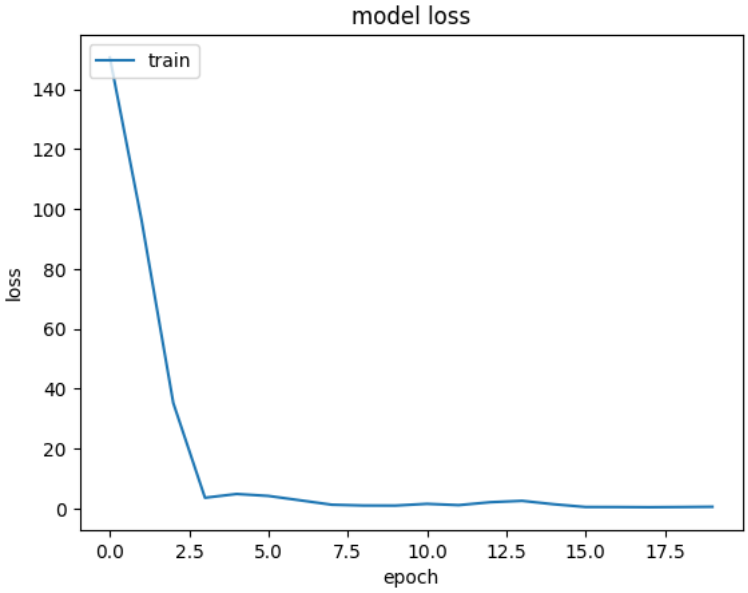
```
history = model.fit(X_train , y_train, epochs=20, batch_size=50)
```

```
import matplotlib.pyplot as plt
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.title('model loss')
plt.ylabel('loss')
```

```
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
dict_keys(['loss', 'accuracy'])
```





```
import numpy as np

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
```

```
from google.colab import files


uploaded = files.upload()
```

Choose Files Iris.csv

- **Iris.csv**(text/csv) - 5107 bytes, last modified: 8/11/2023 - 100% done

```
Saving Iris.csv to Iris.csv
```

```
import pandas as pd
data = pd.read_csv('Iris.csv')
```

```
print(data.dtypes)
data.head()
```

Saved successfully!

```
SepalLengthCm    float64
SepalWidthCm     float64
PetalLengthCm    float64
PetalWidthCm     float64
Species           object
dtype: object
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

```
iris_data = load_iris()
x = iris_data.data
y_ = iris_data.target.reshape(-1, 1) # Convert data to a single column



# One Hot encode the class labels
encoder = OneHotEncoder(sparse=False)
y = encoder.fit_transform(y_)
#print(y)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: Futur
  warnings.warn(
```

```
# Split the data for training and testing
train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.20)



# Build the model

model = Sequential()

model.add(Dense(10, input_shape=(4,), activation='relu', name='fc1'))
model.add(Dense(10, activation='relu', name='fc2'))
model.add(Dense(3, activation='softmax', name='output'))



# Adam optimizer with learning rate of 0.001
optimizer = Adam(lr=0.001)
model.compile(optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

Saved successfully!

```
')
```

```
Neural Network Model Summary:
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 fc1 (Dense)                 (None, 10)                50

 fc2 (Dense)                 (None, 10)                110

 output (Dense)              (None, 3)                 33

=================================================================
Total params: 193
Trainable params: 193
Non-trainable params: 0
```

```
_________________________________________________________________
None
/usr/local/lib/python3.10/dist-packages/keras/optimizers/legacy/adam.py:117: UserWarn
  super().__init__(name, **kwargs)
```

```
# Train the model
history = model.fit(train_x, train_y, verbose=2, batch_size=5, epochs=30)
```

```
Epoch 2/30
24/24 - 0s - loss: 1.0832 - accuracy: 0.2917 - 39ms/epoch - 2ms/step
Epoch 3/30
24/24 - 0s - loss: 0.9497 - accuracy: 0.3500 - 44ms/epoch - 2ms/step
Epoch 4/30
24/24 - 0s - loss: 0.8168 - accuracy: 0.7000 - 47ms/epoch - 2ms/step
Epoch 5/30
24/24 - 0s - loss: 0.7013 - accuracy: 0.7833 - 44ms/epoch - 2ms/step
Epoch 6/30
24/24 - 0s - loss: 0.6017 - accuracy: 0.8333 - 42ms/epoch - 2ms/step
Epoch 7/30
24/24 - 0s - loss: 0.5285 - accuracy: 0.8750 - 50ms/epoch - 2ms/step
Epoch 8/30
24/24 - 0s - loss: 0.4690 - accuracy: 0.8917 - 49ms/epoch - 2ms/step
Epoch 9/30
24/24 - 0s - loss: 0.4293 - accuracy: 0.9417 - 43ms/epoch - 2ms/step
Epoch 10/30
24/24 - 0s - loss: 0.3989 - accuracy: 0.9417 - 42ms/epoch - 2ms/step
Epoch 11/30
24/24 - 0s - loss: 0.3771 - accuracy: 0.9500 - 44ms/epoch - 2ms/step
Epoch 12/30
24/24 - 0s - loss: 0.3587 - accuracy: 0.9583 - 47ms/epoch - 2ms/step
Epoch 13/30
24/24 - 0s - loss: 0.3420 - accuracy: 0.9500 - 62ms/epoch - 3ms/step
Epoch 14/30
24/24 - 0s - loss: 0.3279 - accuracy: 0.9583 - 43ms/epoch - 2ms/step
Epoch 15/30
24/24 - 0s - loss: 0.3160 - accuracy: 0.9583 - 49ms/epoch - 2ms/step
Epoch 16/30
24/24 - 0s - loss: 0.3060 - accuracy: 0.9750 - 44ms/epoch - 2ms/step
Epoch 17/30
24/24 - 0s - loss: 0.2998 - accuracy: 0.9417 - 46ms/epoch - 2ms/step
Epoch 18/30
24/24 - 0s - loss: 0.2856 - accuracy: 0.9500 - 95ms/epoch - 4ms/step
Epoch 19/30
24/24 - 0s - loss: 0.2703 - accuracy: 0.9667 - 89ms/epoch - 4ms/step
Epoch 20/30
24/24 - 0s - loss: 0.2639 - accuracy: 0.9667 - 98ms/epoch - 4ms/step
Epoch 21/30
24/24 - 0s - loss: 0.2558 - accuracy: 0.9583 - 69ms/epoch - 3ms/step
Epoch 22/30
24/24 - 0s - loss: 0.2485 - accuracy: 0.9583 - 81ms/epoch - 3ms/step
Epoch 23/30
24/24 - 0s - loss: 0.2385 - accuracy: 0.9667 - 78ms/epoch - 3ms/step
Epoch 24/30
```

Saved successfully!

```
24/24 - 0s - loss: 0.2253 - accuracy: 0.9667 - 64ms/epoch - 3ms/step
Epoch 26/30
24/24 - 0s - loss: 0.2194 - accuracy: 0.9833 - 57ms/epoch - 2ms/step
Epoch 27/30
24/24 - 0s - loss: 0.2201 - accuracy: 0.9417 - 90ms/epoch - 4ms/step
Epoch 28/30
24/24 - 0s - loss: 0.2107 - accuracy: 0.9667 - 79ms/epoch - 3ms/step
Epoch 29/30
24/24 - 0s - loss: 0.2080 - accuracy: 0.9667 - 82ms/epoch - 3ms/step
Epoch 30/30
24/24 - 0s - loss: 0.1967 - accuracy: 0.9750 - 70ms/epoch - 3ms/step
```

```
# Test on unseen data

results = model.evaluate(test_x, test_y)

print('Final test set loss: {:4f}'.format(results[0]))
print('Final test set accuracy: {:4f}'.format(results[1]))
```

```
1/1 [==============================] - 0s 289ms/step - loss: 0.1605 - accuracy: 1.000
Final test set loss: 0.160539
Final test set accuracy: 1.000000
```

Saved successfully!

Colab paid products - Cancel contracts here

✓ 0s completed at 9:52 PM

Linear Regression using Gradient Descent

Linear Regression by implementing of gradient descent algorithm

The linear regression line is defined as

$$y = \theta_0 + \theta_1 x$$

The parameters θ_0 and θ_1 can be computed using gradeint descent algorithm.

Gradeint descent algorithm is given as:

repeat until convergence {

$$\theta_1 = \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) * x^{(i)}$$

$$\theta_0 = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)$$

}

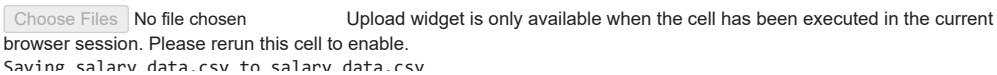
Reference: https://towardsdatascience.com/

Step 1: Import libraries and dataset

```
## Import all the necessary libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

%matplotlib inline
```

```
from google.colab import files
```

```
uploaded = files.upload()
```

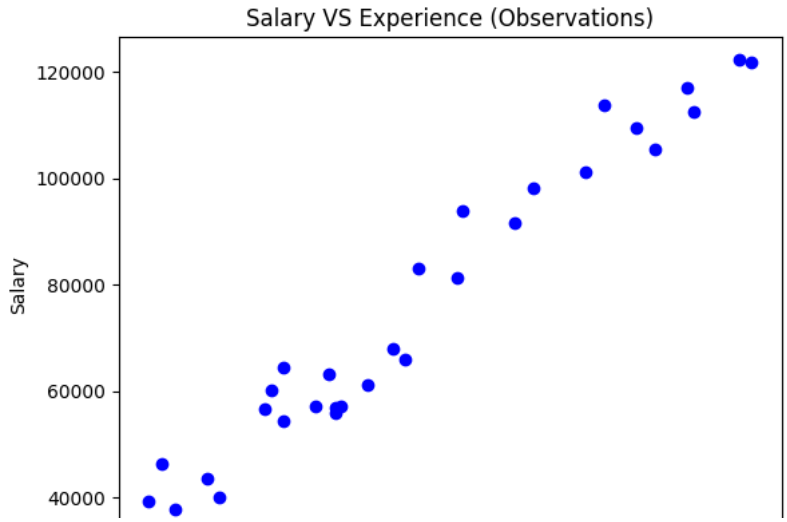



```
Saving salary_data.csv to salary_data.csv
```

```
## Import the dataset

data = pd.read_csv('salary_data.csv')
# change the path as necessary

X = data.iloc[:, 0]
Y = data.iloc[:, 1]

# View the size of the arrays X and Y
print(X.shape)
print(Y.shape)
```

```
(30,)
(30,)
```

```
# Visualize the dataset

plt.scatter(X, Y, color='blue')
plt.title('Salary VS Experience (Observations)')
plt.xlabel('Year of Experience')
plt.ylabel('Salary')
plt.show()
```



Step 2 is skipped.

Year of Experience

Step 3: Create and train the machine learning model

```
# Build the model

theta1 = 0
theta0 = 0

alpha = 0.01  # Learning Rate
epochs = 1000  # Number of iterations to perform gradient descent

m = float(len(X)) # Number of elements in X

cost_history = []
# Performing Gradient Descent
for i in range(epochs):
    Y_pred = theta1 * X + theta0
    temp1 = (-1/m) * sum(X * (Y - Y_pred))
    temp0 = (-1/m) * sum(Y - Y_pred)
    theta1 = theta1 - alpha * temp1
    theta0 = theta0 - alpha * temp0
    cost = (1/2*m) * sum((Y - Y_pred)**2)
    cost_history.append(cost)
```

Step 4: Visualize the results

```
# The coefficients

# print the parameter theta1
print('Theta1 = ', theta1)
# print the parameter theta0
print('Theta0 = ', theta0)
```

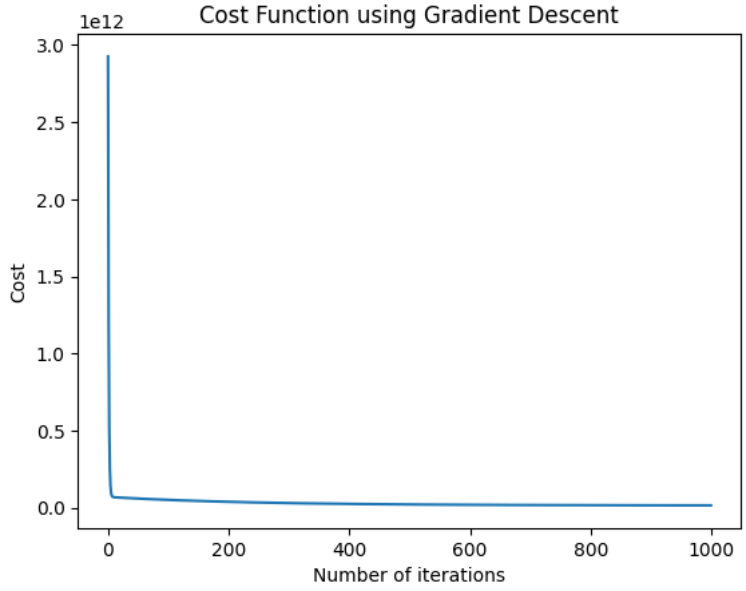
```
Theta1 =  9876.112752879602
Theta0 =  22920.48554852225
```

Step 5: Prediction

```
# Predict the values for the given X
Y_pred = theta1 * X + theta0



# Visualize the dataset and plot the residuals

fig, ax = plt.subplots()

ax.scatter(X, Y, color='blue')       # observed values
ax.scatter(X, Y_pred, color='green') # predicted values
ax.vlines(X,Y, Y_pred, color='red')  # residual lines
plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='black')  # regression line
```

```
plt.title('Salary VS Experience')
plt.xlabel('Year of Experience')
plt.ylabel('Salary')
plt.show()
```


```
# plot the cost function

plt.plot(cost_history)
plt.title('Cost Function using Gradient Descent')
plt.xlabel("Number of iterations")
plt.ylabel("Cost")
plt.show()
```



Step 6: Performance measures

```
from sklearn.metrics import mean_squared_error, r2_score

# The mean squared error
print("Mean squared error = %.2f" % mean_squared_error(Y, Y_pred))

# Explained variance score: 1 is perfect prediction
print('Variance score = %.2f' % r2_score(Y, Y_pred))
```

```
Mean squared error = 33053746.31
Variance score = 0.95
```

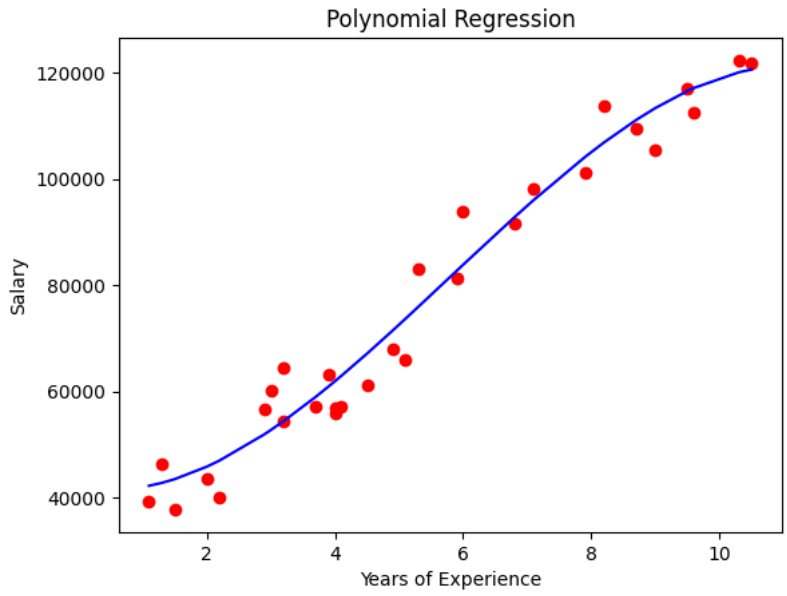
Polynomial Regression Overfitting

Case study: Higher order plynomial regression is prone to overfitting, revisit Exercise 1, let's try a degree 6 polynomial regression

```
## Import the dataset

data = pd.read_csv('salary_data.csv')
# change the path as necessary

X = data.iloc[:, :-1].values # copy all columns excluding last column
Y = data.iloc[:, 1].values   # copy the last column only


# Splitting the dataset into the Training set and Test set
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=1/3, random_state=0)
```

Step 2: Train degree 6 polynomial regression¶

```
from sklearn.preprocessing import PolynomialFeatures

poly_feature = PolynomialFeatures(degree=3)

Xtrain_poly_feature = poly_feature.fit_transform(X_train)
Xtest_poly_feature = poly_feature.transform(X_test)

pol_reg = LinearRegression()
pol_reg.fit(Xtrain_poly_feature, Y_train)


Y_predict = pol_reg.predict(Xtest_poly_feature)



plt.scatter(X, Y, color='red')
plt.plot(X,pol_reg.predict(poly_feature.transform(X)), color='blue')
plt.title('Polynomial Regression')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```



Polynomial regression line seems to be overfitting

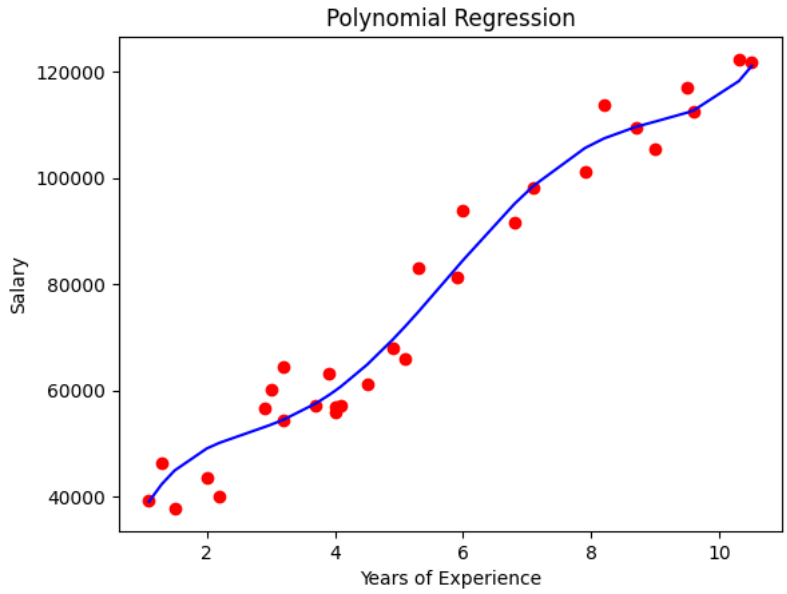
```
from sklearn.metrics import mean_squared_error, r2_score

# The mean squared error
#Mean squared error regression loss

print("Mean squared error = %.2f" % mean_squared_error(Y_test, Y_predict))

# Explained variance score: 1 is perfect prediction.
# R^2 (coefficient of determination) regression score function.
# Best possible score is 1.0, a
```

```
# a constant model that always predicts the expected value of y, disregarding the input features,
#would get a R^2 score of 0.0.

print('Variance score = %.2f' % r2_score(Y_test, Y_predict))
```

```
     Mean squared error = 19744068.28
     Variance score = 0.98
```

Polynomial regression has indded overfitted MSE has increased and R2 score decreased

```
from sklearn.preprocessing import PolynomialFeatures

poly_feature = PolynomialFeatures(degree=5)

Xtrain_poly_feature = poly_feature.fit_transform(X_train)
Xtest_poly_feature = poly_feature.transform(X_test)

pol_reg = LinearRegression()
pol_reg.fit(Xtrain_poly_feature, Y_train)


Y_predict = pol_reg.predict(Xtest_poly_feature)



plt.scatter(X, Y, color='red')
plt.plot(X,pol_reg.predict(poly_feature.transform(X)), color='blue')
plt.title('Polynomial Regression')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```

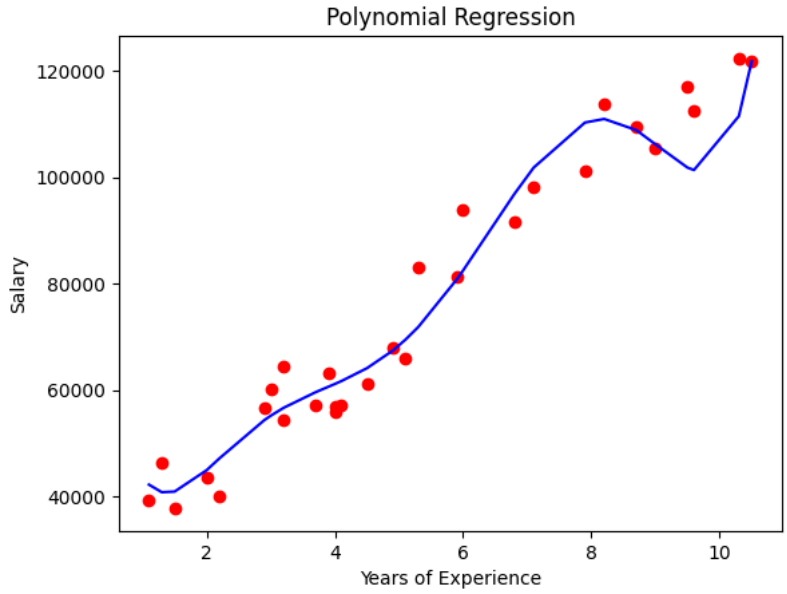



```
from sklearn.metrics import mean_squared_error, r2_score

# The mean squared error
#Mean squared error regression loss

print("Mean squared error = %.2f" % mean_squared_error(Y_test, Y_predict))

# Explained variance score: 1 is perfect prediction.
# R^2 (coefficient of determination) regression score function.
# Best possible score is 1.0, a
# a constant model that always predicts the expected value of y, disregarding the input features,
#would get a R^2 score of 0.0.

print('Variance score = %.2f' % r2_score(Y_test, Y_predict))
```

```
     Mean squared error = 22482651.58
     Variance score = 0.97
```

```
from sklearn.preprocessing import PolynomialFeatures

poly_feature = PolynomialFeatures(degree=6)

Xtrain_poly_feature = poly_feature.fit_transform(X_train)
```

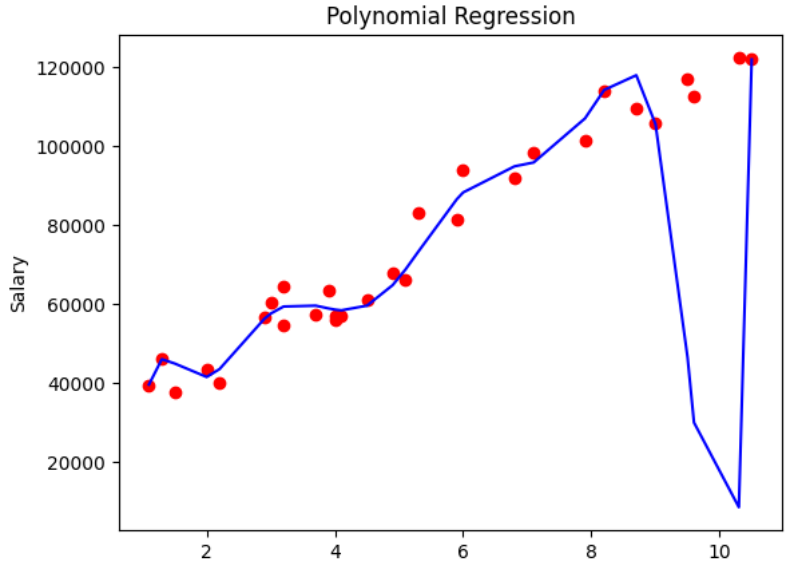
```
Xtest_poly_feature = poly_feature.transform(X_test)

pol_reg = LinearRegression()
pol_reg.fit(Xtrain_poly_feature, Y_train)


Y_predict = pol_reg.predict(Xtest_poly_feature)


plt.scatter(X, Y, color='red')
plt.plot(X,pol_reg.predict(poly_feature.transform(X)), color='blue')
plt.title('Polynomial Regression')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```




```
from sklearn.metrics import mean_squared_error, r2_score

# The mean squared error
#Mean squared error regression loss

print("Mean squared error = %.2f" % mean_squared_error(Y_test, Y_predict))

# Explained variance score: 1 is perfect prediction.
# R^2 (coefficient of determination) regression score function.
# Best possible score is 1.0, a
# a constant model that always predicts the expected value of y, disregarding the input features,
#would get a R^2 score of 0.0.

print('Variance score = %.2f' % r2_score(Y_test, Y_predict))
```

```
Mean squared error = 74756936.74
Variance score = 0.91
```

```
from sklearn.preprocessing import PolynomialFeatures

poly_feature = PolynomialFeatures(degree=9)

Xtrain_poly_feature = poly_feature.fit_transform(X_train)
Xtest_poly_feature = poly_feature.transform(X_test)

pol_reg = LinearRegression()
pol_reg.fit(Xtrain_poly_feature, Y_train)


Y_predict = pol_reg.predict(Xtest_poly_feature)


plt.scatter(X, Y, color='red')
plt.plot(X,pol_reg.predict(poly_feature.transform(X)), color='blue')
plt.title('Polynomial Regression')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```




```
from sklearn.metrics import mean_squared_error, r2_score

# The mean squared error
#Mean squared error regression loss

print("Mean squared error = %.2f" % mean_squared_error(Y_test, Y_predict))

# Explained variance score: 1 is perfect prediction.
# R^2 (coefficient of determination) regression score function.
# Best possible score is 1.0, a
# a constant model that always predicts the expected value of y, disregarding the input features,
#would get a R^2 score of 0.0.

print('Variance score = %.2f' % r2_score(Y_test, Y_predict))
```

```
Mean squared error = 2508742791.30
Variance score = -1.99
```

```
import pandas as pd

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras import regularizers

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.datasets import load_iris


from google.colab import files


uploaded = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to er

```
Saving Iris.csv to Iris.csv
```

```
iris = load_iris()
X = iris.data

y = iris.target


# Split the dataset into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)



# Scale the features using StandardScaler

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)


# Define the model

model = Sequential([Dense(10, input_shape=(4,), activation='relu',kernel_initializer='normal', kernel_regularizer=regularizers.l1(1e-6)),

                    Dense(3, activation='softmax')])



# Compile the model

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])



# Fit the model to the training data

model.fit(X_train, y_train, epochs=100, batch_size=32)



# Evaluate the model on the test data

loss, accuracy = model.evaluate(X_test, y_test)

print("Test Loss:", loss)

print("Test Accuracy:", accuracy)
```

```
Epoch 1/100
4/4 [==============================] - 1s 6ms/step - loss: 1.1069 - accuracy: 0.3833
Epoch 2/100
4/4 [==============================] - 0s 4ms/step - loss: 1.0997 - accuracy: 0.3833
Epoch 3/100
4/4 [==============================] - 0s 4ms/step - loss: 1.0932 - accuracy: 0.3750
Epoch 4/100
4/4 [==============================] - 0s 5ms/step - loss: 1.0859 - accuracy: 0.4083
Epoch 5/100
```

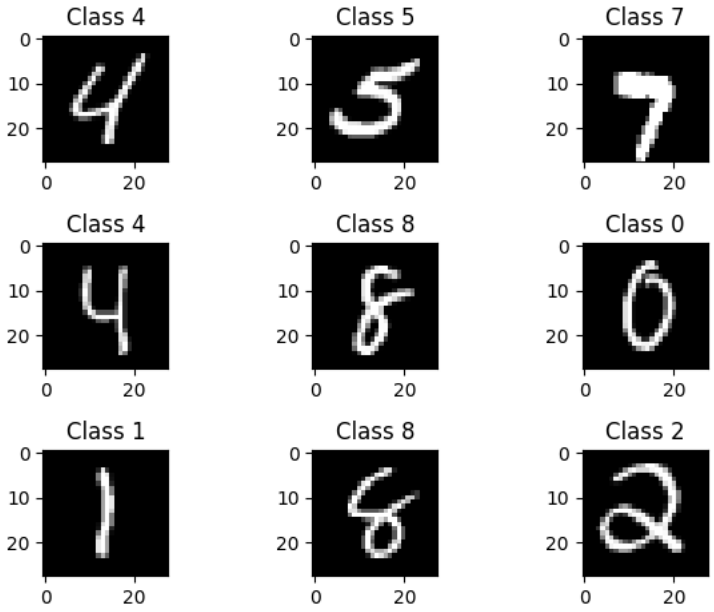
```
4/4 [==============================] - 0s 4ms/step - loss: 1.0793 - accuracy: 0.3917
Epoch 6/100
4/4 [==============================] - 0s 5ms/step - loss: 1.0710 - accuracy: 0.5500
Epoch 7/100
4/4 [==============================] - 0s 5ms/step - loss: 1.0626 - accuracy: 0.5750
Epoch 8/100
4/4 [==============================] - 0s 6ms/step - loss: 1.0540 - accuracy: 0.6000
Epoch 9/100
4/4 [==============================] - 0s 5ms/step - loss: 1.0453 - accuracy: 0.6083
Epoch 10/100
4/4 [==============================] - 0s 4ms/step - loss: 1.0359 - accuracy: 0.6333
Epoch 11/100
4/4 [==============================] - 0s 4ms/step - loss: 1.0263 - accuracy: 0.6583
Epoch 12/100
4/4 [==============================] - 0s 5ms/step - loss: 1.0167 - accuracy: 0.6750
Epoch 13/100
4/4 [==============================] - 0s 4ms/step - loss: 1.0065 - accuracy: 0.6917
Epoch 14/100
4/4 [==============================] - 0s 3ms/step - loss: 0.9964 - accuracy: 0.7000
Epoch 15/100
4/4 [==============================] - 0s 5ms/step - loss: 0.9855 - accuracy: 0.7250
Epoch 16/100
4/4 [==============================] - 0s 4ms/step - loss: 0.9746 - accuracy: 0.7417
Epoch 17/100
4/4 [==============================] - 0s 4ms/step - loss: 0.9634 - accuracy: 0.7417
Epoch 18/100
4/4 [==============================] - 0s 4ms/step - loss: 0.9518 - accuracy: 0.7417
Epoch 19/100
4/4 [==============================] - 0s 4ms/step - loss: 0.9400 - accuracy: 0.7500
Epoch 20/100
4/4 [==============================] - 0s 4ms/step - loss: 0.9277 - accuracy: 0.7583
Epoch 21/100
4/4 [==============================] - 0s 4ms/step - loss: 0.9153 - accuracy: 0.7833
Epoch 22/100
4/4 [==============================] - 0s 4ms/step - loss: 0.9022 - accuracy: 0.7667
Epoch 23/100
4/4 [==============================] - 0s 3ms/step - loss: 0.8896 - accuracy: 0.7750
Epoch 24/100
4/4 [==============================] - 0s 4ms/step - loss: 0.8765 - accuracy: 0.7750
Epoch 25/100
4/4 [==============================] - 0s 4ms/step - loss: 0.8630 - accuracy: 0.7833
Epoch 26/100
4/4 [==============================] - 0s 4ms/step - loss: 0.8492 - accuracy: 0.7833
Epoch 27/100
4/4 [==============================] - 0s 4ms/step - loss: 0.8364 - accuracy: 0.7833
Epoch 28/100
4/4 [==============================] - 0s 4ms/step - loss: 0.8222 - accuracy: 0.7917
Epoch 29/100
4/4 [==============================] - 0s 6ms/step - loss: 0.8086 - accuracy: 0.7917
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random

from keras.datasets import mnist
from keras.models import Sequential

from keras.layers.core import Dense, Dropout, Activation
from keras.utils import np_utils


(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("X_train shape", X_train.shape)
print("y_train shape", y_train.shape)
print("X_test shape", X_test.shape)
print("y_test shape", y_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
X_train shape (60000, 28, 28)
y_train shape (60000,)
X_test shape (10000, 28, 28)
y_test shape (10000,)
```

We have imported the MNIST dataset which is pre-loaded in Keras. We will use the Sequential Model and import the basic layers and util tools.

```
for i in range(9):
    plt.subplot(3,3,i+1)
    num = random.randint(0, len(X_train))
    plt.imshow(X_train[num], cmap='gray', interpolation='none')
    plt.title("Class {}".format(y_train[num]))

plt.tight_layout()
```



Here we are randomly selecting 9 images from the dataset and plotting them to get an idea of the handwritten digits and their respective classes.

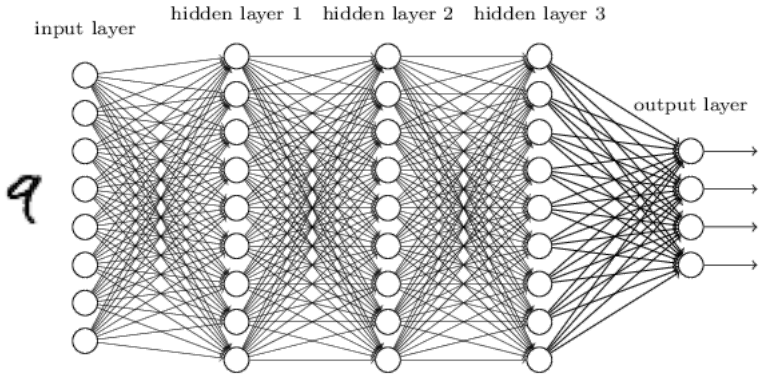
```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

X_train /= 255
X_test /= 255

print("Training matrix shape", X_train.shape)
print("Testing matrix shape", X_test.shape)
```

```
Training matrix shape (60000, 784)
Testing matrix shape (10000, 784)
```

Data Preprocessing Instead of a 28 x 28 matrix, we build our network to accept a 784-length vector. Pixel values range from 0 to 255 where 0 is black and 255 is pure white. We will normalize these values by dividing them by 255 so that we get the output pixel values between [0,1] in the same magnitude.

Note that we are working with grayscale images of dimension 28 x 28 pixels. If we have color images, we have 3 channels for RGB, i.e. 28 x 28 x 3, each with pixel value in the range 0 to 255.

```
no_classes = 10

Y_train = np_utils.to_categorical(y_train, no_classes)
Y_test = np_utils.to_categorical(y_test, no_classes)
```



First Hidden Layer

```
model = Sequential()
```

Double-click (or enter) to edit

```
model.add(Dense(512, input_shape=(784,)))
```

The first hidden layer has 512 nodes (neurons) whose input is a vector of size 784. Each node will receive an element from each input vector and apply some weight and bias to it.

```
model.add(Activation('relu'))
```

In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. ReLU stands for rectified linear unit, and is a type of activation function. ReLU:f(x)=max(0,x)

Double-click (or enter) to edit

```
model.add(Dropout(0.2))
```

Dropout randomly selects a few nodes and nullifies their output (deactivates the node). This helps in ensuring that the model is not overfitted to the training dataset.

```
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))
```

Second Hidden Layer The second hidden layer also has 512 nodes and it takes input from the 512 nodes in the previous layer and gives its output to the next subsequent layer.

```
model.add(Dense(10))
model.add(Activation('softmax'))
```

Final Output Layer The final layer of 10 neurons in fully-connected to the previous 512-node layer. The final layer should be equal to the number of desired output classes. The Softmax Activation represents a probability distribution over n different possible outcomes. Its values are all

non-negative and sum to 1. For example, if the final output is: [0, 0.94, 0, 0, 0, 0, 0, 0.06, 0, 0] then it is most probable that the image is that of the digit 1

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               401920

 activation (Activation)     (None, 512)               0

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 512)               262656

 activation_1 (Activation)   (None, 512)               0

 dropout_1 (Dropout)         (None, 512)               0

 dense_2 (Dense)             (None, 10)                5130

 activation_2 (Activation)   (None, 10)                0

=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

The loss function we'll use here is called categorical cross-entropy and is a loss function well-suited to comparing two probability distributions. The cross-entropy is a measure of how different your predicted distribution is from the target distribution.

Optimizers are algorithms or methods used to change the attributes of the neural network such as weights and learning rate to reduce the losses. Optimizers are used to solve optimization problems by minimizing the loss function. In our case, we use the Adam Optimizer.

```
history = model.fit(X_train, Y_train,
          batch_size=128, epochs=10,
          verbose=1)
```

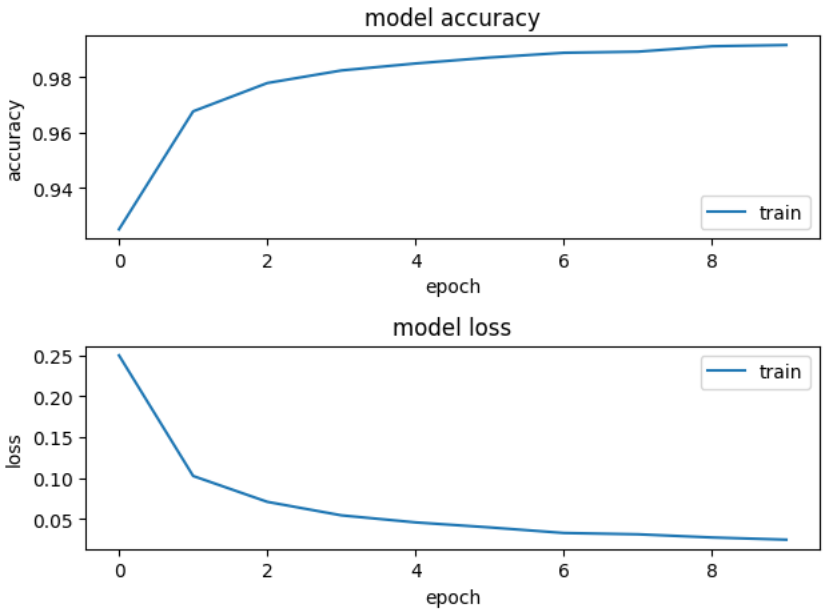
```
Epoch 1/10
469/469 [==============================] - 10s 19ms/step - loss: 0.2498 - accuracy: 0.9251
Epoch 2/10
469/469 [==============================] - 9s 19ms/step - loss: 0.1025 - accuracy: 0.9676
Epoch 3/10
469/469 [==============================] - 8s 16ms/step - loss: 0.0710 - accuracy: 0.9778
Epoch 4/10
469/469 [==============================] - 10s 22ms/step - loss: 0.0545 - accuracy: 0.9823
Epoch 5/10
469/469 [==============================] - 14s 29ms/step - loss: 0.0459 - accuracy: 0.9848
Epoch 6/10
469/469 [==============================] - 9s 19ms/step - loss: 0.0397 - accuracy: 0.9869
Epoch 7/10
469/469 [==============================] - 8s 16ms/step - loss: 0.0329 - accuracy: 0.9887
Epoch 8/10
469/469 [==============================] - 9s 19ms/step - loss: 0.0313 - accuracy: 0.9891
Epoch 9/10
469/469 [==============================] - 9s 19ms/step - loss: 0.0275 - accuracy: 0.9910
Epoch 10/10
469/469 [==============================] - 8s 17ms/step - loss: 0.0248 - accuracy: 0.9915
```

```
score = model.evaluate(X_test, Y_test)
print('Test accuracy:', score[1])
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.0720 - accuracy: 0.9811
Test accuracy: 0.9811000227928162
```

```
fig = plt.figure()
plt.subplot(2,1,1)
plt.plot(history.history['accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')

plt.subplot(2,1,2)
plt.plot(history.history['loss'])
plt.title('model loss')
plt.ylabel('loss')
```

```
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')

plt.tight_layout()
#https://www.kaggle.com/code/williamkempson/dog-cat-classification/notebook
```



Convolutional Neural Network with Dropout

```
# Import Libraries
# - Tensorflow
# - Keras
# - numpy and random
from tensorflow import keras
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras import layers

import random
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
random.seed(42)         # Initialize the random number generator.
np.random.seed(42)      # With the seed reset, the same set of numbers will appear every time.
tf.set_random_seed(42)  # sets the graph-level random seed
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-2-b5b7b33c24c6> in <cell line: 3>()
      1 random.seed(42)         # Initialize the random number generator.
      2 np.random.seed(42)      # With the seed reset, the same set of numbers will appear every time.
----> 3 tf.set_random_seed(42)  # sets the graph-level random seed

AttributeError: module 'tensorflow' has no attribute 'set_random_seed'
```

SEARCH STACK OVERFLOW

Dataset - MNIST

```
# Use the MNIST dataset  of Keras.

mnist = tf.keras.datasets.mnist

(Xtrain, Ytrain), (Xtest,Ytest) = mnist.load_data()

# Display size of dataset
Xtrain = Xtrain.reshape((60000,28,28,1))
Xtrain = Xtrain.astype('float32')/255

Xtest = Xtest.reshape((10000,28,28,1))
Xtest = Xtest.astype('float32')/255

Ytrain = tf.keras.utils.to_categorical(Ytrain)
Ytest = tf.keras.utils.to_categorical(Ytest)

print(Xtrain.shape, Xtest.shape)
print(Ytrain.shape, Ytest.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 28, 28, 1) (10000, 28, 28, 1)
(60000, 10) (10000, 10)
```

Create a CNN Model

```
# Create a Sequential model object
cnnModel = models.Sequential()

# Add layers Conv2D for CNN and sepcify MaxPooling

# Layer 1 = input layer
cnnModel.add(layers.Conv2D(32, (3,3), activation="relu", input_shape=(28,28,1) ))

cnnModel.add(layers.MaxPooling2D((2,2)))


# Layer 2
cnnModel.add(layers.Conv2D(64, (3,3), activation="relu"))
```

```
cnnModel.add(layers.MaxPooling2D((2,2)))


# Add dropout of 50% to layer 2
cnnModel.add(layers.Dropout(0.5))



# Layer 3
cnnModel.add(layers.Conv2D(64, (3,3), activation="relu" ))


# Add dropout of 50% to layer 3
cnnModel.add(layers.Dropout(0.5))


cnnModel.add(layers.Flatten())


# Add Dense layers or fully connected layers
# Layer 4
cnnModel.add(layers.Dense(64, activation="relu" ))

# Layer 5
cnnModel.add(layers.Dense(32, activation="relu" ))

# Layer 6
cnnModel.add(layers.Dense(10, activation="softmax" ))

cnnModel.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 64)          0
 g2D)

 dropout (Dropout)           (None, 5, 5, 64)          0

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928

 dropout_1 (Dropout)         (None, 3, 3, 64)          0

 flatten (Flatten)           (None, 576)               0

 dense (Dense)               (None, 64)                36928

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 10)                330

=================================================================
Total params: 95082 (371.41 KB)
Trainable params: 95082 (371.41 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
# Configure  the model for training, by using appropriate optimizers and regularizations
# Available optimizer: adam, rmsprop, adagrad, sgd
# loss:  objective that the model will try to minimize.
# Available loss: categorical_crossentropy, binary_crossentropy, mean_squared_error
# metrics: List of metrics to be evaluated by the model during training and testing.

cnnModel.compile(optimizer = "adam", loss = "categorical_crossentropy", metrics = ["accuracy"])



# train the model

history = cnnModel.fit(Xtrain, Ytrain, epochs = 5, batch_size = 64, validation_split = 0.1)
```

```
Epoch 1/5
844/844 [==============================] - 45s 53ms/step - loss: 0.0955 - accuracy: 0.9697 - val_loss: 0.0425 - val_accuracy: 0.988
Epoch 2/5
844/844 [==============================] - 41s 49ms/step - loss: 0.0809 - accuracy: 0.9749 - val_loss: 0.0394 - val_accuracy: 0.989
Epoch 3/5
```

```
844/844 [==============================] - 42s 49ms/step - loss: 0.0698 - accuracy: 0.9780 - val_loss: 0.0332 - val_accuracy: 0.9893
Epoch 4/5
844/844 [==============================] - 41s 49ms/step - loss: 0.0641 - accuracy: 0.9798 - val_loss: 0.0302 - val_accuracy: 0.9913
Epoch 5/5
844/844 [==============================] - 43s 51ms/step - loss: 0.0603 - accuracy: 0.9811 - val_loss: 0.0318 - val_accuracy: 0.9906
```

```
print('Final training loss \t', history.history['loss'][-1])
print('Final training accuracy ', history.history['accuracy'][-1])
```

```
Final training loss      0.06032975763082504
Final training accuracy  0.9811481237411499
```

Results and Outputs

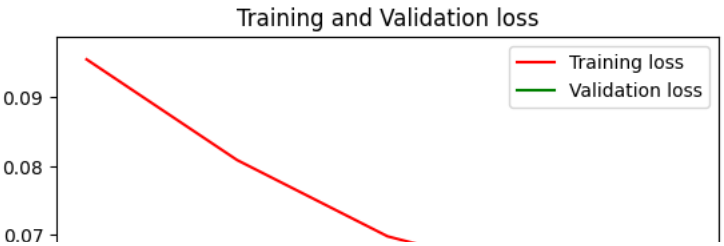
```
# testing the model

testLoss, testAccuracy = cnnModel.evaluate( Xtest, Ytest)
```

```
313/313 [==============================] - 2s 7ms/step - loss: 0.0248 - accuracy: 0.9926
```

```
print('Testing loss \t', testLoss)
print('Testing accuracy ', testAccuracy)
```

```
Testing loss     0.02480687014758587
Testing accuracy  0.9926000237464905
```

```
# plotting training and validation loss

loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, color='red', label='Training loss')
plt.plot(epochs, val_loss, color='green', label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# plotting training and validation accuracy

acc = history.history['accuracy']
val_acc = history.history['val_acc']
plt.plot(epochs, acc, color='red', label='Training acc')
plt.plot(epochs, val_acc, color='green', label='Validation acc')
plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Confusion Matrix generation

Prediction for a specific testing data generte confusion matrix

```
Y_prediction = cnnModel.predict(Xtest)

# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_prediction, axis = 1)

# Convert validation observations to one hot vectors
Y_true = np.argmax(Ytest,axis = 1)
```

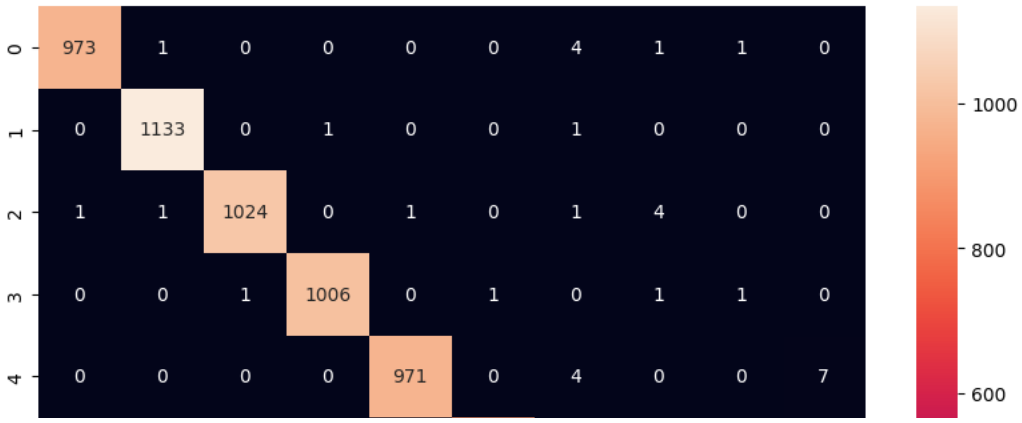
```
313/313 [==============================] - 3s 8ms/step
```

```
18 plt.plot(epochs, acc, color='red', label='Training acc')
```

```
# Classification Report

from sklearn.metrics import classification_report

print(classification_report(Y_true, Y_pred_classes))
```

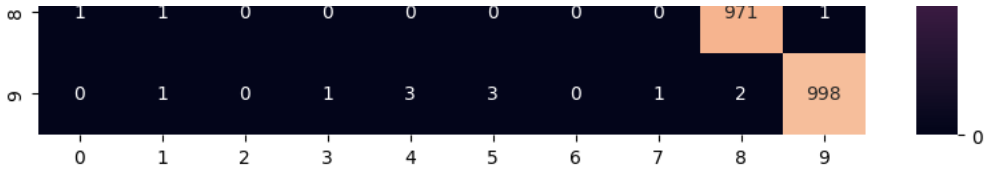
	precision	recall	f1-score	support
0	1.00	0.99	0.99	980
1	0.99	1.00	0.99	1135
2	0.99	0.99	0.99	1032
3	0.99	1.00	1.00	1010
4	0.99	0.99	0.99	982
5	0.99	0.99	0.99	892
6	0.99	0.99	0.99	958
7	0.99	0.99	0.99	1028
8	1.00	1.00	1.00	974
9	0.99	0.99	0.99	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

```
# confusion matrix
from sklearn.metrics import confusion_matrix
import seaborn as sns

# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)

plt.figure(figsize=(10,8))
sns.heatmap(confusion_mtx, annot=True, fmt="d");
```



Modify the code to get a better testing accuracy.

- Change the number of hidden units
- Increase the number of hidden layers
- Use a different optimizer
- Train for more epochs for better graphs
- Try using CIFAR dataset



Convolutional Neural Network with Dropout

```
# Import Libraries
# - Tensorflow
# - Keras
# - numpy and random
from tensorflow import keras
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras import layers

import random
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline


random.seed(42)         # Initialize the random number generator.
np.random.seed(42)      # With the seed reset, the same set of numbers will appear every time.
tf.random.set_seed(42)  # sets the graph-level random seed
```

Dataset - MNIST

```
# Use the MNIST dataset  of Keras.

mnist = tf.keras.datasets.cifar10

(Xtrain, Ytrain), (Xtest,Ytest) = mnist.load_data()

# Display size of dataset
#Xtrain = Xtrain.reshape((60000,28,28,1))
Xtrain = Xtrain.astype('float32')/255

#Xtest = Xtest.reshape((10000,28,28,1))
Xtest = Xtest.astype('float32')/255

Ytrain = tf.keras.utils.to_categorical(Ytrain)
Ytest = tf.keras.utils.to_categorical(Ytest)

print(Xtrain.shape, Xtest.shape)
print(Ytrain.shape, Ytest.shape)
```

```
(50000, 32, 32, 3) (10000, 32, 32, 3)
(50000, 10) (10000, 10)
```

Create a CNN Model

```
# Create a Sequential model object
cnnModel = models.Sequential()

# Add layers Conv2D for CNN and sepcify MaxPooling

# Layer 1 = input layer
cnnModel.add(layers.Conv2D(32, (3,3), activation="relu", input_shape=(32,32,3) ))

#cnnModel.add(layers.MaxPooling2D((2,2)))


# Layer 2
cnnModel.add(layers.Conv2D(64, (3,3), activation="relu"))

#cnnModel.add(layers.MaxPooling2D((2,2)))


# Add dropout of 50% to layer 2
cnnModel.add(layers.Dropout(0.5))



# Layer 3
cnnModel.add(layers.Conv2D(64, (3,3), activation="relu" ))


# Add dropout of 50% to layer 3
```

```
cnnModel.add(layers.Dropout(0.5))


cnnModel.add(layers.Flatten())


# Add Dense layers or fully connected layers
# Layer 4
cnnModel.add(layers.Dense(64, activation="relu" ))

# Layer 5
cnnModel.add(layers.Dense(32, activation="relu" ))

# Layer 6
cnnModel.add(layers.Dense(10, activation="softmax" ))

cnnModel.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_9 (Conv2D)           (None, 30, 30, 32)        896

 conv2d_10 (Conv2D)          (None, 28, 28, 64)        18496

 dropout_6 (Dropout)         (None, 28, 28, 64)        0

 conv2d_11 (Conv2D)          (None, 26, 26, 64)        36928

 dropout_7 (Dropout)         (None, 26, 26, 64)        0

 flatten_3 (Flatten)         (None, 43264)             0

 dense_9 (Dense)             (None, 64)                2768960

 dense_10 (Dense)            (None, 32)                2080

 dense_11 (Dense)            (None, 10)                330

=================================================================
Total params: 2827690 (10.79 MB)
Trainable params: 2827690 (10.79 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
# Configure  the model for training, by using appropriate optimizers and regularizations
# Available optimizer: adam, rmsprop, adagrad, sgd
# loss:  objective that the model will try to minimize.
# Available loss: categorical_crossentropy, binary_crossentropy, mean_squared_error
# metrics: List of metrics to be evaluated by the model during training and testing.

cnnModel.compile(optimizer = "adam", loss = "categorical_crossentropy", metrics = ["accuracy"])



# train the model

history = cnnModel.fit(Xtrain, Ytrain, epochs = 5, batch_size = 64, validation_split = 0.1)
```

```
Epoch 1/5
704/704 [==============================] - 21s 13ms/step - loss: 1.5563 - accuracy: 0.4350 - val_loss: 1.2683 - val_accuracy: 0.5446
Epoch 2/5
704/704 [==============================] - 9s 13ms/step - loss: 1.1930 - accuracy: 0.5716 - val_loss: 1.0580 - val_accuracy: 0.6216
Epoch 3/5
704/704 [==============================] - 8s 12ms/step - loss: 1.0298 - accuracy: 0.6359 - val_loss: 0.9797 - val_accuracy: 0.6608
Epoch 4/5
704/704 [==============================] - 9s 13ms/step - loss: 0.9309 - accuracy: 0.6700 - val_loss: 0.9425 - val_accuracy: 0.6750
Epoch 5/5
704/704 [==============================] - 9s 12ms/step - loss: 0.8440 - accuracy: 0.7021 - val_loss: 0.9089 - val_accuracy: 0.6912
```

```
print('Final training loss \t', history.history['loss'][-1])
print('Final training accuracy ', history.history['accuracy'][-1])
```

```
Final training loss      0.8440192937850952
Final training accuracy  0.7021111249923706
```

Results and Outputs

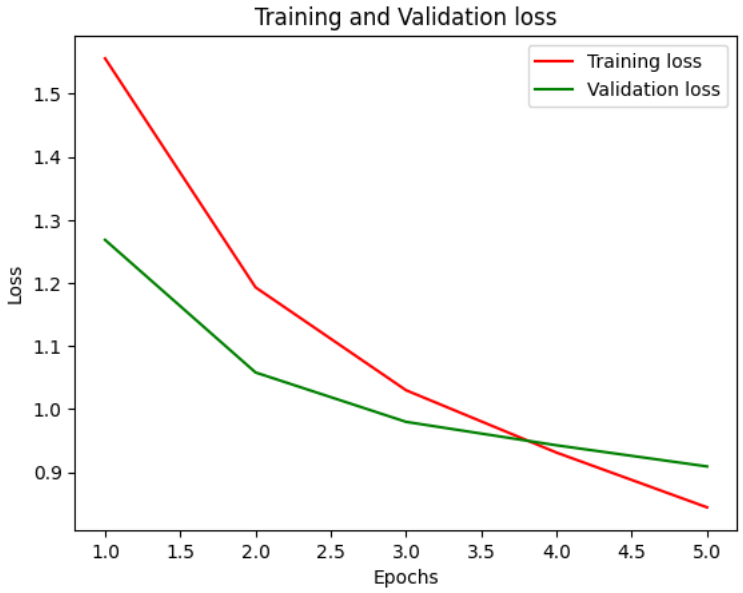
```
# testing the model

testLoss, testAccuracy = cnnModel.evaluate( Xtest, Ytest)
```

```
313/313 [==============================] - 1s 4ms/step - loss: 0.9357 - accuracy: 0.6733
```

```
print('Testing loss \t', testLoss)
print('Testing accuracy ', testAccuracy)
```

```
Testing loss     0.9356647729873657
Testing accuracy  0.67330002784729
```

```
# plotting training and validation loss

loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, color='red', label='Training loss')
plt.plot(epochs, val_loss, color='green', label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# plotting training and validation accuracy

acc = history.history['accuracy']
val_acc = history.history['val_acc']
plt.plot(epochs, acc, color='red', label='Training acc')
plt.plot(epochs, val_acc, color='green', label='Validation acc')
plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```




```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-18-566ec5548cc6> in <cell line: 17>()
     15
     16 acc = history.history['accuracy']
---> 17 val_acc = history.history['val_acc']
     18 plt.plot(epochs, acc, color='red', label='Training acc')
     19 plt.plot(epochs, val_acc, color='green', label='Validation acc')

KeyError: 'val_acc'
```

SEARCH STACK OVERFLOW

Confusion Matrix generation

Prediction for a specific testing data generte confusion matrix

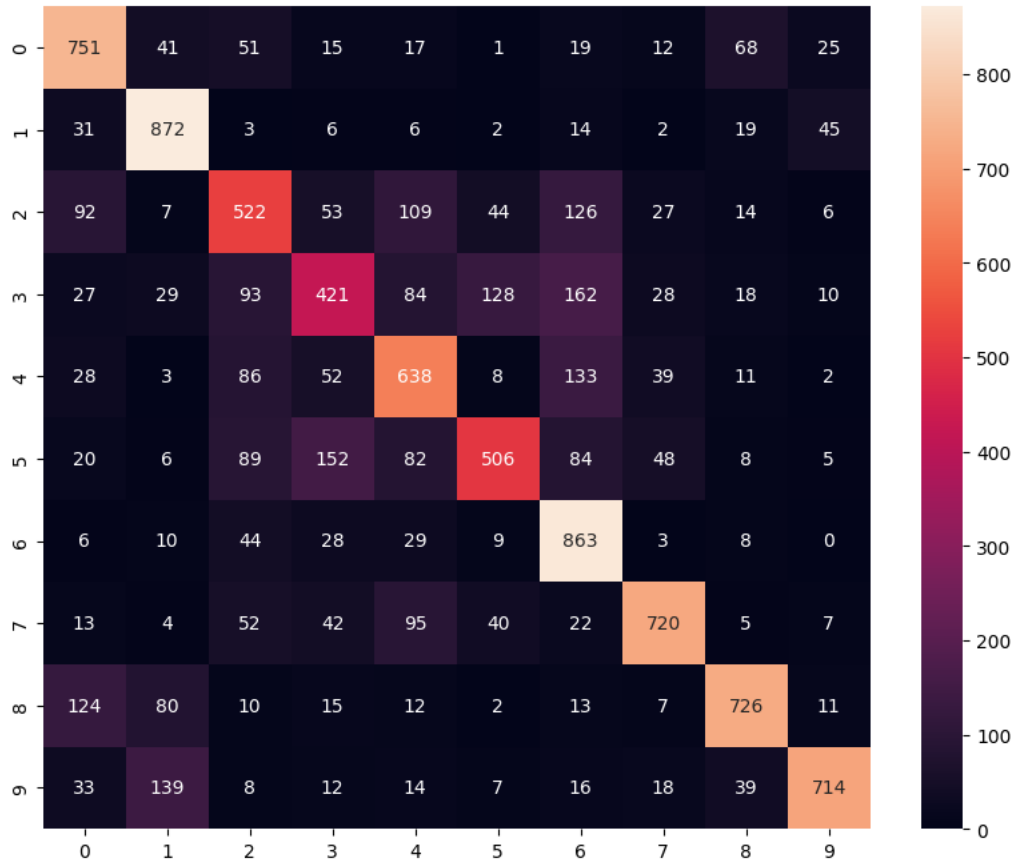
```
Y_prediction = cnnModel.predict(Xtest)

# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_prediction, axis = 1)

# Convert validation observations to one hot vectors
Y_true = np.argmax(Ytest,axis = 1)
```

```
313/313 [==============================] - 1s 2ms/step
```

```
# Classification Report

from sklearn.metrics import classification_report

print(classification_report(Y_true, Y_pred_classes))
```

```
              precision    recall  f1-score   support

           0       0.67      0.75      0.71      1000
           1       0.73      0.87      0.80      1000
           2       0.54      0.52      0.53      1000
           3       0.53      0.42      0.47      1000
           4       0.59      0.64      0.61      1000
           5       0.68      0.51      0.58      1000
           6       0.59      0.86      0.70      1000
           7       0.80      0.72      0.76      1000
           8       0.79      0.73      0.76      1000
           9       0.87      0.71      0.78      1000

    accuracy                           0.67     10000
   macro avg       0.68      0.67      0.67     10000
weighted avg       0.68      0.67      0.67     10000
```

```
# confusion matrix
from sklearn.metrics import confusion_matrix
import seaborn as sns

# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)

plt.figure(figsize=(10,8))
sns.heatmap(confusion_mtx, annot=True, fmt="d");
```



Modify the code to get a better testing accuracy.

- Change the number of hidden units
- Increase the number of hidden layers
- Use a different optimizer
- Train for more epochs for better graphs
- Try using CIFAR dataset

Convolutional Neural Network with Early Stopping

```
# Import Libraries
# - Tensorflow
# - Keras
# - numpy and random
from tensorflow import keras
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras import layers

from tensorflow.keras.callbacks import EarlyStopping

import random
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline


random.seed(42)         # Initialize the random number generator.
np.random.seed(42)      # With the seed reset, the same set of numbers will appear every time.
tf.random.set_seed(42)  # sets the graph-level random seed
```

Dataset - MNIST

```
# Use the MNIST dataset  of Keras.

mnist = tf.keras.datasets.mnist

(Xtrain, Ytrain), (Xtest,Ytest) = mnist.load_data()

# Display size of dataset
Xtrain = Xtrain.reshape((60000,28,28,1))
Xtrain = Xtrain.astype('float32')/255

Xtest = Xtest.reshape((10000,28,28,1))
Xtest = Xtest.astype('float32')/255

Ytrain = tf.keras.utils.to_categorical(Ytrain)
Ytest = tf.keras.utils.to_categorical(Ytest)

print(Xtrain.shape, Xtest.shape)
print(Ytrain.shape, Ytest.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 28, 28, 1) (10000, 28, 28, 1)
(60000, 10) (10000, 10)
```

Create a CNN Model

```
# Create a Sequential model object
cnnModel = models.Sequential()

# Add layers Conv2D for CNN and sepcify MaxPooling

# Layer 1 = input layer
cnnModel.add(layers.Conv2D(32, (3,3), activation="relu", input_shape=(28,28,1) ))

cnnModel.add(layers.MaxPooling2D((2,2)))


# Layer 2
cnnModel.add(layers.Conv2D(64, (3,3), activation="relu"))

cnnModel.add(layers.MaxPooling2D((2,2)))


# Layer 3
cnnModel.add(layers.Conv2D(64, (3,3), activation="relu" ))


cnnModel.add(layers.Flatten())
```

```
# Add Dense layers or fully connected layers
# Layer 4
cnnModel.add(layers.Dense(64, activation="relu" ))

# Layer 5
cnnModel.add(layers.Dense(32, activation="relu" ))

# Layer 6
cnnModel.add(layers.Dense(10, activation="softmax" ))

cnnModel.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928

 flatten (Flatten)           (None, 576)               0

 dense (Dense)               (None, 64)                36928

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 10)                330

=================================================================
Total params: 95082 (371.41 KB)
Trainable params: 95082 (371.41 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
# Configure  the model for training, by using appropriate optimizers and regularizations
# Available optimizer: adam, rmsprop, adagrad, sgd
# loss:  objective that the model will try to minimize.
# Available loss: categorical_crossentropy, binary_crossentropy, mean_squared_error
# metrics: List of metrics to be evaluated by the model during training and testing.

cnnModel.compile(optimizer = "adam", loss = "categorical_crossentropy", metrics = ["accuracy"])
```

```
# train the model
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1)

# fit model
history = cnnModel.fit(Xtrain, Ytrain, epochs = 25, batch_size = 64,
                       validation_split = 0.1, callbacks=[es])
```

```
Epoch 1/25
844/844 [==============================] - 16s 5ms/step - loss: 0.2245 - accuracy: 0.9313 - val_loss: 0.0620 - val_accuracy: 0.9815
Epoch 2/25
844/844 [==============================] - 5s 6ms/step - loss: 0.0575 - accuracy: 0.9821 - val_loss: 0.0627 - val_accuracy: 0.9818
Epoch 2: early stopping
```

```
print('Final training loss \t', history.history['loss'][-1])
print('Final training accuracy ', history.history['accuracy'][-1])
```

```
Final training loss      0.0575222410261631
Final training accuracy  0.9820740818977356
```

Results and Outputs

```
# testing the model

testLoss, testAccuracy = cnnModel.evaluate( Xtest, Ytest)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.0552 - accuracy: 0.9812
```

```
print('Testing loss \t', testLoss)
print('Testing accuracy ', testAccuracy)
```

```
Testing loss     0.05519125238060951
Testing accuracy  0.9811999797821045
```

```
# plotting training and validation loss

loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, color='red', label='Training loss')
plt.plot(epochs, val_loss, color='green', label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# plotting training and validation accuracy

acc = history.history['accuracy']
val_acc = history.history['val_acc']
plt.plot(epochs, acc, color='red', label='Training acc')
plt.plot(epochs, val_acc, color='green', label='Validation acc')
plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```


```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-15-566ec5548cc6> in <cell line: 17>()
     15
     16 acc = history.history['accuracy']
---> 17 val_acc = history.history['val_acc']
     18 plt.plot(epochs, acc, color='red', label='Training acc')
     19 plt.plot(epochs, val_acc, color='green', label='Validation acc')

KeyError: 'val_acc'
```

SEARCH STACK OVERFLOW

Confusion Matrix generation

Prediction for a specific testing data generte confusion matrix

```
Y_prediction = cnnModel.predict(Xtest)

# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_prediction, axis = 1)

# Convert validation observations to one hot vectors
```

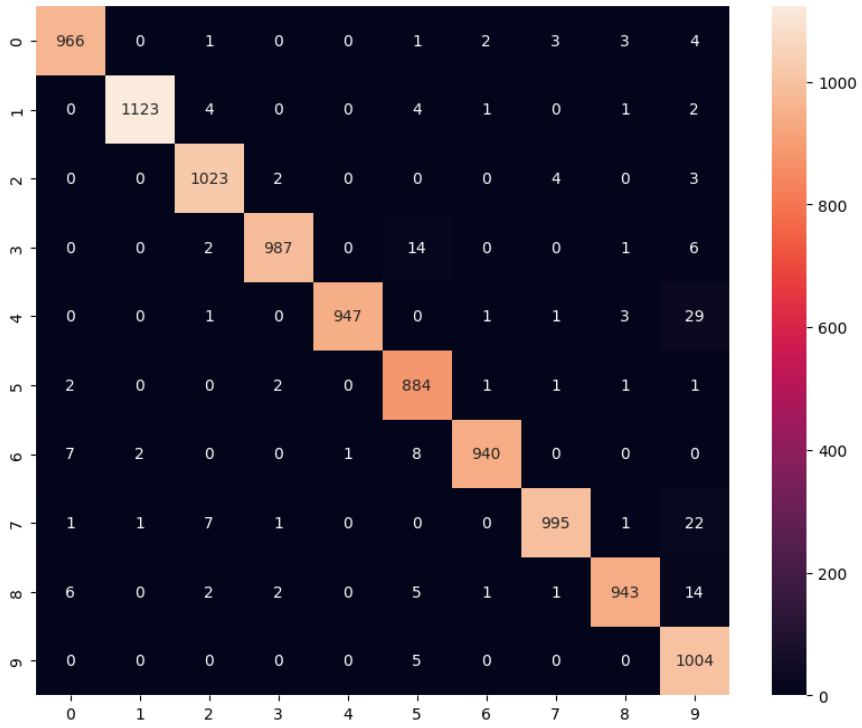
```
Y_true = np.argmax(Ytest,axis = 1)
```

```
313/313 [==============================] - 1s 2ms/step
```

```
# Classification Report

from sklearn.metrics import classification_report

print(classification_report(Y_true, Y_pred_classes))
```

```
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       980
           1       1.00      0.99      0.99      1135
           2       0.98      0.99      0.99      1032
           3       0.99      0.98      0.99      1010
           4       1.00      0.96      0.98       982
           5       0.96      0.99      0.98       892
           6       0.99      0.98      0.99       958
           7       0.99      0.97      0.98      1028
           8       0.99      0.97      0.98       974
           9       0.93      1.00      0.96      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

```
# confusion matrix
from sklearn.metrics import confusion_matrix
import seaborn as sns

# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)

plt.figure(figsize=(10,8))
sns.heatmap(confusion_mtx, annot=True, fmt="d");
```



Modify the code to get a better testing accuracy.

- Change the number of hidden units
- Increase the number of hidden layers
- Use a different optimizer

- Train for more epochs for better graphs
- Try using CIFAR dataset

Open in Colab

I. Introduction

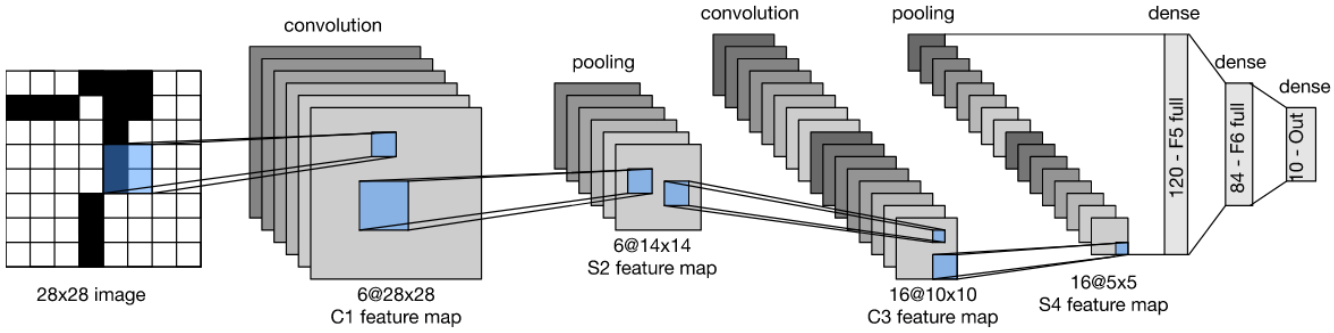
LeNet (or LeNet-5) is a convolutional neural network structure proposed by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner in 1989. The first purpose of this network is to recognize handwritten digits in images. It was successfully applied for identifying handwritten zip code numbers provided by the US Postal Service [1].

II. Architechture

LeNet consists of 2 parts:

- The first part includes two convolutional layers and two pooling layers which are placed alternatively.
- The second part consists of three fully connected layers.

The architecture of LeNet is described by the following figure:



In the figure above, Cx, Sx, Fx are corresponding to the convolutional layer, sub-sampling layer (a.k.a pooling layer), and fully connected layer, respectively, where x denotes the layer index.

- The input is images of size 28 × 28
- C1 is the first convolutional layer with 6 convolution kernels of size 5× 5.
- S2 is the pooling layer that outputs 6 channels of 14 × 14 images. The pooling window size, in this case, is a square matrix of size 2 × 2.
- C3 is a convolutional layer with 16 convolution kernels of size 5 × 5. Hence, the output of this layer is 16 feature images of size 10 × 10.
- S4 is a pooling layer with a pooling window of size 2 × 2. Hence, the dimension of images through this layer is halved, it outputs 16 feature images of size 5 × 5.
- C5 is the convolutional layer with 120 convolution kernels of size 5 × 5. Since the inputs of this layer have the same size as the kernel, then the output size of this layer is 1 × 1. The number of channels in output equals the channel number of kernels, which is 120. Hence the output of this layer is 120 feature images of size 1 × 1.
- F6 is a fully connected layer with 84 neurons which are all connected to the output of C5.
- The output layer consists of 10 neurons corresponding to the number of classes (numbers from 0 to 9).

III. Application for recognizing MNIST hanwritten digit images

In this section, we apply LeNet for recognizing MNIST handwritten digit images. This network is constructed in Keras platform:

```
from google.colab import drive
drive.mount('/content/drive/')
```

```
Mounted at /content/drive/
```

```
import os
os.chdir('/content/drive/MyDrive/Deep learning/')
```

1. Loading MNIST dataset

```
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

```
print('X_train shape', X_train.shape, 'X_test shape', X_test.shape)
```

```
print( X_train shape , X_train.shape,  X_test shape , X_test.shape)
```

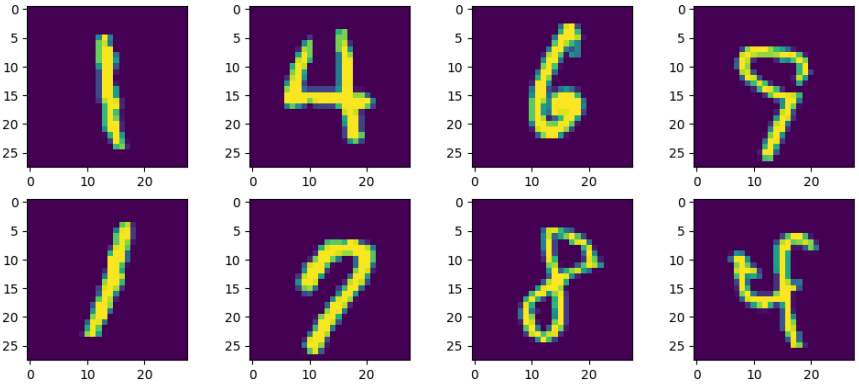
```
X_train shape (60000, 28, 28) X_test shape (10000, 28, 28)
```

```
type(X_train)
```

```
numpy.ndarray
```

Visualizing randomly some images in the training set:

```
import matplotlib.pyplot as plt
import random

plt.figure(figsize = (12,5))
for i in range(8):
  ind = random.randint(0, len(X_train))
  plt.subplot(240+1+i)
  plt.imshow(X_train[ind])
```



2. Preprocessing data

This task includes the following steps:

- Reshape images into required size of Keras
- Convert integer values into float values
- Normalize data
- One-hot encodeing labels

```
from keras.utils import to_categorical

def preprocess_data(X_train, y_train, X_test, y_test):
  # reshape images to the the required size by Keras
  X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], X_train.shape[2], 1)
  X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], X_test.shape[2], 1)
  # convert from integers to floats
  X_train = X_train.astype('float32')
  X_test = X_test.astype('float32')
  # normalize to range 0-1
  X_train = X_train/255.0
  X_test_norm = X_test/255.0
  # One-hot encoding label
  y_train = to_categorical(y_train)
  y_test = to_categorical(y_test)
  return X_train, y_train, X_test, y_test
```

3. Buiding the LeNet model

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten
from keras.optimizers import SGD
```

```
# metrics
from keras.metrics import categorical_crossentropy
# optimization method
from keras.optimizers import SGD


def LeNet():
  model = Sequential()
  model.add(Conv2D(filters = 6, kernel_size = (5,5), padding = 'same', activation = 'relu', input_shape = (28,28,1)))
  model.add(MaxPooling2D(pool_size = (2,2)))
  model.add(Conv2D(filters = 16, kernel_size = (5,5), activation = 'relu'))
  model.add(MaxPooling2D(pool_size = (2,2)))
  model.add(Flatten())
  model.add(Dense(120, activation = 'relu'))
  model.add(Dense(10, activation = 'softmax'))
  # compile the model with a loss function, a metric and an optimizer function
  opt = SGD(lr = 0.01)
  model.compile(loss = categorical_crossentropy,
                optimizer = opt,
                metrics = ['accuracy'])
  return model


LeNet_model = LeNet()
LeNet_model.summary()
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate` or use the legacy optimizer, e.g.,tf.keras.optimizers
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 max_pooling2d (MaxPooling2  (None, 14, 14, 6)         0
 D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 16)          0
 g2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 10)                1210

=================================================================
Total params: 51902 (202.74 KB)
Trainable params: 51902 (202.74 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

4. Training the model

Define the training function

```
def train_model(model, X_train, y_train, X_test, y_test, epochs = 50, batch_size = 128):
  # Rescaling all training and testing data
  X_train, y_train, X_test, y_test = preprocess_data(X_train, y_train, X_test, y_test)
  # Fitting the model
  history = model.fit(X_train, y_train, epochs = epochs, batch_size = batch_size, steps_per_epoch = X_train.shape[0]//batch_size, validat
  # evaluate the model
  _, acc = model.evaluate(X_test, y_test, verbose = 1)
  print('%.3f' % (acc * 100.0))
  summary_history(history)


def summary_history(history):
  plt.figure(figsize = (10,6))
  plt.plot(history.history['accuracy'], color = 'blue', label = 'train')
  plt.plot(history.history['val_accuracy'], color = 'red', label = 'val')
  plt.legend()
  plt.title('Accuracy')
  plt.show()


train_model(LeNet_model, X_train, y_train, X_test, y_test)
```

```
Epoch 1/50
468/468 [==============================] - 12s 5ms/step - loss: 0.9909 - accuracy: 0.7112 - val_loss: 31.6331 - val_accuracy: 0.9188
Epoch 2/50
468/468 [==============================] - 3s 6ms/step - loss: 0.2638 - accuracy: 0.9205 - val_loss: 25.7502 - val_accuracy: 0.9405
Epoch 3/50
468/468 [==============================] - 2s 5ms/step - loss: 0.1893 - accuracy: 0.9438 - val_loss: 19.2988 - val_accuracy: 0.9545
Epoch 4/50
468/468 [==============================] - 2s 5ms/step - loss: 0.1525 - accuracy: 0.9533 - val_loss: 16.5439 - val_accuracy: 0.9608
Epoch 5/50
468/468 [==============================] - 2s 5ms/step - loss: 0.1296 - accuracy: 0.9613 - val_loss: 14.0119 - val_accuracy: 0.9647
Epoch 6/50
468/468 [==============================] - 2s 5ms/step - loss: 0.1159 - accuracy: 0.9656 - val_loss: 12.4414 - val_accuracy: 0.9717
Epoch 7/50
468/468 [==============================] - 2s 5ms/step - loss: 0.1045 - accuracy: 0.9685 - val_loss: 11.8338 - val_accuracy: 0.9737
Epoch 8/50
468/468 [==============================] - 3s 5ms/step - loss: 0.0955 - accuracy: 0.9715 - val_loss: 10.1226 - val_accuracy: 0.9743
Epoch 9/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0892 - accuracy: 0.9732 - val_loss: 10.1150 - val_accuracy: 0.9766
Epoch 10/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0838 - accuracy: 0.9746 - val_loss: 10.1209 - val_accuracy: 0.9756
Epoch 11/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0782 - accuracy: 0.9763 - val_loss: 9.1431 - val_accuracy: 0.9785
Epoch 12/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0747 - accuracy: 0.9771 - val_loss: 8.7905 - val_accuracy: 0.9792
Epoch 13/50
468/468 [==============================] - 3s 6ms/step - loss: 0.0702 - accuracy: 0.9790 - val_loss: 8.9365 - val_accuracy: 0.9799
Epoch 14/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0682 - accuracy: 0.9788 - val_loss: 8.3987 - val_accuracy: 0.9802
Epoch 15/50
468/468 [==============================] - 2s 4ms/step - loss: 0.0640 - accuracy: 0.9803 - val_loss: 6.8029 - val_accuracy: 0.9827
Epoch 16/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0626 - accuracy: 0.9813 - val_loss: 7.3919 - val_accuracy: 0.9831
Epoch 17/50
468/468 [==============================] - 2s 4ms/step - loss: 0.0590 - accuracy: 0.9824 - val_loss: 7.8722 - val_accuracy: 0.9810
Epoch 18/50
468/468 [==============================] - 3s 6ms/step - loss: 0.0571 - accuracy: 0.9827 - val_loss: 7.2372 - val_accuracy: 0.9825
Epoch 19/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0559 - accuracy: 0.9828 - val_loss: 6.5507 - val_accuracy: 0.9840
Epoch 20/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0532 - accuracy: 0.9840 - val_loss: 6.8523 - val_accuracy: 0.9838
Epoch 21/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0515 - accuracy: 0.9845 - val_loss: 6.5120 - val_accuracy: 0.9836
Epoch 22/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0500 - accuracy: 0.9851 - val_loss: 6.9858 - val_accuracy: 0.9848
Epoch 23/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0485 - accuracy: 0.9855 - val_loss: 8.1122 - val_accuracy: 0.9811
Epoch 24/50
468/468 [==============================] - 3s 6ms/step - loss: 0.0466 - accuracy: 0.9856 - val_loss: 6.6527 - val_accuracy: 0.9848
Epoch 25/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0458 - accuracy: 0.9863 - val_loss: 6.3094 - val_accuracy: 0.9841
Epoch 26/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0430 - accuracy: 0.9868 - val_loss: 6.7836 - val_accuracy: 0.9853
Epoch 27/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0434 - accuracy: 0.9868 - val_loss: 5.8459 - val_accuracy: 0.9864
Epoch 28/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0419 - accuracy: 0.9872 - val_loss: 5.8540 - val_accuracy: 0.9863
Epoch 29/50
468/468 [==============================] - 3s 6ms/step - loss: 0.0406 - accuracy: 0.9877 - val_loss: 7.7690 - val_accuracy: 0.9837
Epoch 30/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0393 - accuracy: 0.9879 - val_loss: 7.3689 - val_accuracy: 0.9847
Epoch 31/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0377 - accuracy: 0.9881 - val_loss: 5.7369 - val_accuracy: 0.9862
Epoch 32/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0383 - accuracy: 0.9877 - val_loss: 6.2868 - val_accuracy: 0.9846
Epoch 33/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0362 - accuracy: 0.9887 - val_loss: 5.7517 - val_accuracy: 0.9863
Epoch 34/50
468/468 [==============================] - 3s 6ms/step - loss: 0.0366 - accuracy: 0.9888 - val_loss: 5.9492 - val_accuracy: 0.9860
Epoch 35/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0347 - accuracy: 0.9892 - val_loss: 6.8869 - val_accuracy: 0.9854
Epoch 36/50
468/468 [==============================] - 2s 5ms/step - loss: 0.0340 - accuracy: 0.9895 - val_loss: 5.7426 - val_accuracy: 0.9868
```

5. Prediction

```
import numpy as np


# predict labels for the test set
y_test_pred = []
for i in range(len(X_test)):
  img = X_test[i]
  img = img.reshape(1,28,28,1)
  img = img.astype('float32')
  img = img/255.0
  # one-hot vector output
  vec_p = LeNet_model.predict(img)
  # determine the lable corresponding to vec_p
  y_p = np.argmax(vec_p)
  y_test_pred.append(y_p)
```

```
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 18ms/step
```

```
import keras,os
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPool2D , Flatten
from keras.preprocessing.image import ImageDataGenerator
import numpy as np
```

We'll import ImageDataGenerator from keras.preprocessing. The objective of ImageDataGenerator is to make it easier to import data with labels into the model. It's a useful class, as it comes with a variety of functions that allow you to rescale, rotate, zoom and flip, etc. The most useful thing about this class is that it doesn't affect the data stored on the disk. This class alters the data on the go while passing it to the model.

```
trdata = ImageDataGenerator()
traindata = trdata.flow_from_directory(directory=r"C:\Users\mpstme.student\Downloads\train", target_size=(224,224))
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-3-228a0f76a63e> in <cell line: 2>()
      1 trdata = ImageDataGenerator()
----> 2 traindata =
trdata.flow_from_directory(directory=r"C:\Users\mpstme.student\Downloads\train",
target_size=(224,224))
      3

 1 frames
/usr/local/lib/python3.10/dist-packages/keras/src/preprocessing/image.py in
__init__(self, directory, image_data_generator, target_size, color_mode, classes,
class_mode, batch_size, shuffle, seed, data_format, save_to_dir, save_prefix,
save_format, follow_links, subset, interpolation, keep_aspect_ratio, dtype)
    561         if not classes:
    562             classes = []
--> 563             for subdir in sorted(os.listdir(directory)):
    564                 if os.path.isdir(os.path.join(directory, subdir)):
    565                     classes.append(subdir)

FileNotFoundError: [Errno 2] No such file or directory:
```

```
model = Sequential()
model.add(Conv2D(input_shape=(224,224,3),filters=64,kernel_size=(3,3),padding="same", activation="relu"))
model.add(Conv2D(filters=64,kernel_size=(3,3),padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
```

```
model.add(Flatten())
model.add(Dense(units=4096,activation="relu"))
model.add(Dense(units=4096,activation="relu"))
model.add(Dense(units=2, activation="softmax"))
```

```
from keras.optimizers import Adam
opt = Adam(lr=0.001)
model.compile(optimizer=opt, loss=keras.losses.categorical_crossentropy, metrics=['accuracy'])
model.summary()
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate` or use the legacy optimizer, e.g.,tf.keras.optimiz
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 224, 224, 64)      1792

 conv2d_1 (Conv2D)           (None, 224, 224, 64)      36928

 max_pooling2d (MaxPooling2  (None, 112, 112, 64)      0
```

```
D)

conv2d_2 (Conv2D)           (None, 112, 112, 128)     73856

conv2d_3 (Conv2D)           (None, 112, 112, 128)     147584

max_pooling2d_1 (MaxPoolin  (None, 56, 56, 128)       0
g2D)

conv2d_4 (Conv2D)           (None, 56, 56, 256)       295168

conv2d_5 (Conv2D)           (None, 56, 56, 256)       590080

conv2d_6 (Conv2D)           (None, 56, 56, 256)       590080

max_pooling2d_2 (MaxPoolin  (None, 28, 28, 256)       0
g2D)

conv2d_7 (Conv2D)           (None, 28, 28, 512)       1180160

conv2d_8 (Conv2D)           (None, 28, 28, 512)       2359808

conv2d_9 (Conv2D)           (None, 28, 28, 512)       2359808

max_pooling2d_3 (MaxPoolin  (None, 14, 14, 512)       0
g2D)

conv2d_10 (Conv2D)          (None, 14, 14, 512)       2359808

conv2d_11 (Conv2D)          (None, 14, 14, 512)       2359808

conv2d_12 (Conv2D)          (None, 14, 14, 512)       2359808

max_pooling2d_4 (MaxPoolin  (None, 7, 7, 512)         0
g2D)

flatten (Flatten)           (None, 25088)             0

dense (Dense)               (None, 4096)              102764544

dense_1 (Dense)             (None, 4096)              16781312

dense_2 (Dense)             (None, 2)                 8194

=================================================================
Total params: 134268738 (512.19 MB)
```

```
from keras.callbacks import ModelCheckpoint, EarlyStopping
checkpoint = ModelCheckpoint("vgg16_1.h5", monitor='val_acc', verbose=1, save_best_only=True , save_weights_only=False, mode='auto', peri
early = EarlyStopping(monitor='val_acc', min_delta=0, patience=20, verbose=1, mode='auto')
hist = model.fit_generator(steps_per_epoch=100,generator=traindata, validation_data= testdata, validation_steps=10,epochs=100,callbacks=[
```

```
WARNING:tensorflow:`period` argument is deprecated. Please use `save_freq` to specify the frequency in number of batches seen.
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-0c0afea1e1c9> in <cell line: 4>()
      2 checkpoint = ModelCheckpoint("vgg16_1.h5", monitor='val_acc', verbose=1, save_best_only=True, save_weights_only=False, mode=
      3 early = EarlyStopping(monitor='val_acc', min_delta=0, patience=20, verbose=1, mode='auto')
----> 4 hist = model.fit_generator(steps_per_epoch=100,generator=traindata, validation_data= testdata, validation_steps=10,epochs=10

NameError: name 'traindata' is not defined
```

SEARCH STACK OVERFLOW

VISUALIZE THE TRAINING/VALIDATION DATA

```
import matplotlib.pyplot as plt
plt.plot(hist.history["acc"])
plt.plot(hist.history['val_acc'])
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title("model accuracy")
plt.ylabel("Accuracy")
plt.xlabel("Epoch")
plt.legend(["Accuracy","Validation Accuracy","loss","Validation Loss"])
plt.show()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-0901f352c6e2> in <cell line: 2>()
      1 import matplotlib.pyplot as plt
----> 2 plt.plot(hist.history["acc"])
      3 plt.plot(hist.history['val_acc'])
      4 plt.plot(hist.history['loss'])
```

```
from keras.preprocessing import image
img = image.load_img("image.jpeg",target_size=(224,224))
img = np.asarray(img)
plt.imshow(img)
img = np.expand_dims(img, axis=0)
from keras.models import load_model
saved_model = load_model("vgg16_1.h5")
output = saved_model.predict(img)
if output[0][0] > output[0][1]:
    print("cat")
else:
    print('dog')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-9-a50b9d8b7260> in <cell line: 2>()
      1 from keras.preprocessing import image
----> 2 img = image.load_img("image.jpeg",target_size=(224,224))
      3 img = np.asarray(img)
      4 plt.imshow(img)
      5 img = np.expand_dims(img, axis=0)

/usr/local/lib/python3.10/dist-packages/keras/src/utils/image_utils.py in load_img(path, grayscale, color_mode, target_size, interpo
    420         if isinstance(path, pathlib.Path):
    421             path = str(path.resolve())
--> 422         with open(path, "rb") as f:
    423             img = pil_image.open(io.BytesIO(f.read()))
    424     else:

FileNotFoundError: [Errno 2] No such file or directory: 'image.jpeg'
```

SEARCH STACK OVERFLOW

Time Series Forecasting using RNN

```
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

import pandas as pd

import tensorflow as tf
from tensorflow import keras
```

Functions used for processing/preparing the data

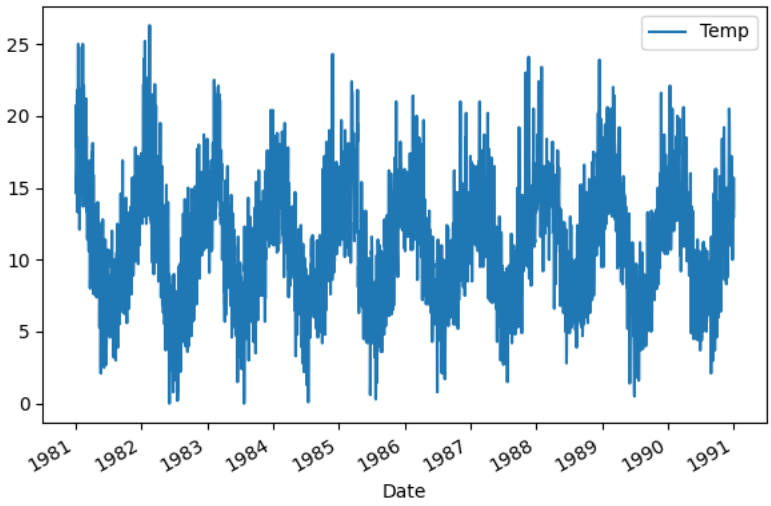
New Section

Dataset - Daily minimum temperatures in Melbourne from 1981 to 1990

```
dailyTemps = pd.read_csv("/content/sample_data/daily-min-temperatures.csv",
                         parse_dates=[0], index_col=[0])
dailyTemps.head()
```

Date	Temp
1981-01-01	20.7
1981-01-02	17.9
1981-01-03	18.8
1981-01-04	14.6
1981-01-05	15.8

```
dailyTemps.plot(figsize=(6,4))
plt.tight_layout()
plt.show()
```



Based on previous 7 days minimum temperature, predict the minimum temperature after two days

```
def createDelayedColumns(series, times):
    cols = []
    column_index = []
    for time in times:
        cols.append(series.shift(-time))
        lag_fmt = "t+{time}" if time > 0 else "t{time}" if time < 0 else "t"
        column_index += [(lag_fmt.format(time=time), col_name)
```

```
                        for col_name in series.columns]
    df = pd.concat(cols, axis=1)
    df.columns = pd.MultiIndex.from_tuples(column_index)
    return df

def convert3D(df):
    shape = [-1] + [len(level) for level in df.columns.remove_unused_levels().levels]
    return df.values.reshape(shape)




previousDays = 7
afterDays = 2

X = createDelayedColumns(dailyTemps,
        times=range(-previousDays+1,1)).iloc[previousDays:-afterDays]
y = createDelayedColumns(dailyTemps,
        times=[afterDays]).iloc[previousDays:-afterDays]

print(X.head())
print(y.head())

print(X.shape)
print(y.shape)
```

```
                 t-6   t-5   t-4   t-3   t-2   t-1     t
                Temp  Temp  Temp  Temp  Temp  Temp  Temp
     Date
     1981-01-08  17.9  18.8  14.6  15.8  15.8  15.8  17.4
     1981-01-09  18.8  14.6  15.8  15.8  15.8  17.4  21.8
     1981-01-10  14.6  15.8  15.8  15.8  17.4  21.8  20.0
     1981-01-11  15.8  15.8  15.8  17.4  21.8  20.0  16.2
     1981-01-12  15.8  15.8  17.4  21.8  20.0  16.2  13.3
                  t+2
                 Temp
     Date
     1981-01-08  20.0
     1981-01-09  16.2
     1981-01-10  13.3
     1981-01-11  16.7
     1981-01-12  21.5
     (3641, 7)
     (3641, 1)
```

Now you have X and y, slice them into training and test dataset.

```
train_slice = slice(None, '1988-12-28')
test_slice = slice('1989-01-01', None)

Xtrain, Ytrain = X.loc[train_slice], y.loc[train_slice]
Xtest, Ytest = X.loc[test_slice], y.loc[test_slice]

print(Xtrain.shape)
print(Xtest.shape)
```

```
     (2911, 7)
     (728, 7)
```

RNN needs 3D input

```
Xtrain3D = convert3D(Xtrain)
Xtest3D = convert3D(Xtest)

print(Xtrain3D.shape)
print(Xtest3D.shape)

Xtest3D
```

```
     (2911, 7, 1)
     (728, 7, 1)
     array([[[15.8],
             [ 9.5],
             [12.9],
             ...,
             [14.8],
             [14.1],
             [14.3]],

            [[ 9.5],
             [12.9],
```

```
       [12.9],
       ...,
       [14.1],
       [14.3],
       [17.4]],

      [[12.9],
       [12.9],
       [14.8],
       ...,
       [14.3],
       [17.4],
       [18.5]],

      ...,

      [[13.1],
       [13.2],
       [13.9],
       ...,
       [12.9],
       [14.6],
       [14. ]],

      [[13.2],
       [13.9],
       [10. ],
       ...,
       [14.6],
       [14. ],
       [13.6]],

      [[13.9],
       [10. ],
       [12.9],
       ...,
       [14. ],
       [13.6],
       [13.5]]])
```

Vanilla - Simple RNN

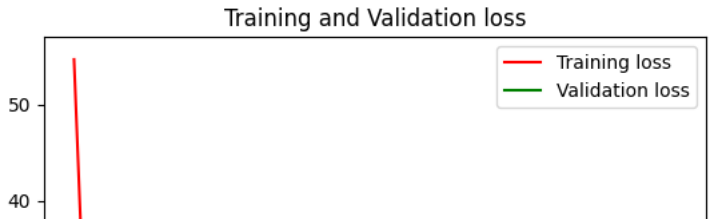
```
rnnModel = keras.models.Sequential()

rnnModel.add(keras.layers.SimpleRNN(50, return_sequences=True,
                                    input_shape=(7,1)))

rnnModel.add(keras.layers.SimpleRNN(25))

rnnModel.add(keras.layers.Dense(1))

rnnModel.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn (SimpleRNN)      (None, 7, 50)             2600

 simple_rnn_1 (SimpleRNN)    (None, 25)                1900

 dense (Dense)               (None, 1)                 26

=================================================================
Total params: 4526 (17.68 KB)
Trainable params: 4526 (17.68 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
# Configure  the model for training, by using appropriate optimizers and regularizations
# Available optimizer: adam, rmsprop, adagrad, sgd
# loss:  objective that the model will try to minimize.
# Available loss: categorical_crossentropy, binary_crossentropy, mean_squared_error
# metrics: List of metrics to be evaluated by the model during training and testing.

rnnModel.compile(loss='mse', optimizer='adam', metrics=['mae'])


# train the model

history = rnnModel.fit(Xtrain3D, Ytrain, epochs = 50, batch_size=20, validation_split=0.1, verbose=1 )
```

```
Epoch 23/50
131/131 [==============================] - 1s 6ms/step - loss: 7.5533 - mae: 2.1364 - val_loss: 8.2676 - val_mae: 2.2430
Epoch 24/50
131/131 [==============================] - 1s 7ms/step - loss: 7.5729 - mae: 2.1343 - val_loss: 8.4466 - val_mae: 2.2838
Epoch 25/50
131/131 [==============================] - 1s 7ms/step - loss: 7.4905 - mae: 2.1253 - val_loss: 8.9766 - val_mae: 2.3643
Epoch 26/50
131/131 [==============================] - 1s 6ms/step - loss: 7.5111 - mae: 2.1289 - val_loss: 8.5829 - val_mae: 2.2829
Epoch 27/50
131/131 [==============================] - 1s 7ms/step - loss: 7.4783 - mae: 2.1256 - val_loss: 8.5495 - val_mae: 2.3068
Epoch 28/50
131/131 [==============================] - 1s 10ms/step - loss: 7.5486 - mae: 2.1288 - val_loss: 8.9349 - val_mae: 2.3713
Epoch 29/50
131/131 [==============================] - 1s 10ms/step - loss: 7.4569 - mae: 2.1211 - val_loss: 8.8395 - val_mae: 2.3454
Epoch 30/50
131/131 [==============================] - 1s 7ms/step - loss: 7.3973 - mae: 2.1096 - val_loss: 8.5603 - val_mae: 2.2700
Epoch 31/50
131/131 [==============================] - 1s 8ms/step - loss: 7.4438 - mae: 2.1150 - val_loss: 8.8902 - val_mae: 2.3769
Epoch 32/50
131/131 [==============================] - 2s 14ms/step - loss: 7.3885 - mae: 2.1164 - val_loss: 8.7864 - val_mae: 2.3342
Epoch 33/50
131/131 [==============================] - 2s 14ms/step - loss: 7.4713 - mae: 2.1180 - val_loss: 8.6630 - val_mae: 2.2933
Epoch 34/50
131/131 [==============================] - 1s 11ms/step - loss: 7.3229 - mae: 2.1012 - val_loss: 8.6509 - val_mae: 2.3068
Epoch 35/50
131/131 [==============================] - 1s 9ms/step - loss: 7.3935 - mae: 2.1125 - val_loss: 8.8262 - val_mae: 2.3245
Epoch 36/50
131/131 [==============================] - 1s 9ms/step - loss: 7.3128 - mae: 2.1067 - val_loss: 8.7826 - val_mae: 2.3197
Epoch 37/50
131/131 [==============================] - 1s 6ms/step - loss: 7.3340 - mae: 2.1067 - val_loss: 8.9958 - val_mae: 2.3499
Epoch 38/50
131/131 [==============================] - 1s 9ms/step - loss: 7.3077 - mae: 2.1003 - val_loss: 8.9681 - val_mae: 2.3252
Epoch 39/50
131/131 [==============================] - 1s 9ms/step - loss: 7.3371 - mae: 2.1058 - val_loss: 8.9630 - val_mae: 2.3704
Epoch 40/50
131/131 [==============================] - 1s 7ms/step - loss: 7.2595 - mae: 2.0948 - val_loss: 8.9984 - val_mae: 2.3435
Epoch 41/50
131/131 [==============================] - 1s 6ms/step - loss: 7.2468 - mae: 2.0809 - val_loss: 9.0291 - val_mae: 2.3682
Epoch 42/50
131/131 [==============================] - 1s 6ms/step - loss: 7.1869 - mae: 2.0847 - val_loss: 8.7664 - val_mae: 2.3210
Epoch 43/50
131/131 [==============================] - 1s 6ms/step - loss: 7.2644 - mae: 2.0909 - val_loss: 8.9014 - val_mae: 2.3219
Epoch 44/50
131/131 [==============================] - 1s 6ms/step - loss: 7.2310 - mae: 2.0971 - val_loss: 9.0814 - val_mae: 2.3695
Epoch 45/50
131/131 [==============================] - 1s 6ms/step - loss: 7.1883 - mae: 2.0837 - val_loss: 8.8115 - val_mae: 2.3256
Epoch 46/50
131/131 [==============================] - 1s 6ms/step - loss: 7.1698 - mae: 2.0765 - val_loss: 8.7554 - val_mae: 2.3394
Epoch 47/50
131/131 [==============================] - 1s 6ms/step - loss: 7.1337 - mae: 2.0746 - val_loss: 8.9238 - val_mae: 2.3486
Epoch 48/50
131/131 [==============================] - 1s 6ms/step - loss: 7.2174 - mae: 2.0915 - val_loss: 9.0808 - val_mae: 2.3675
Epoch 49/50
131/131 [==============================] - 1s 6ms/step - loss: 7.1237 - mae: 2.0731 - val_loss: 9.3229 - val_mae: 2.3914
Epoch 50/50
131/131 [==============================] - 1s 6ms/step - loss: 7.0922 - mae: 2.0617 - val_loss: 8.6459 - val_mae: 2.2971
```

```
# plotting training and validation loss

loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, color='red', label='Training loss')
plt.plot(epochs, val_loss, color='green', label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```




```
testResult = rnnModel.evaluate(Xtest3D, Ytest)
print(testResult)
```

```
23/23 [==============================] - 0s 3ms/step - loss: 7.1639 - mae: 2.1034
[7.163897514343262, 2.1033895015716553]
```

```
rnnModel.predict(Xtest3D)
```

```
[15.181013 ],
[16.092846 ],
[15.11408  ],
[15.308973 ],
[13.718303 ],
[13.743988 ],
[13.624321 ],
[14.805035 ],
[15.428326 ],
[15.064596 ],
[15.406561 ],
[14.783416 ],
[14.938082 ],
[13.716433 ],
[11.741401 ],
[11.047916 ],
[11.333168 ],
[10.386427 ],
[12.848482 ],
[10.693253 ],
[13.615856 ],
[13.939246 ],
[13.196203 ],
[14.542439 ],
[12.144136 ],
[14.121059 ],
[13.819738 ],
[14.427272 ],
[13.99911  ],
[14.542582 ],
[14.189804 ],
[14.393771 ],
[14.324692 ],
[11.755526 ],
[12.337224 ],
[10.971712 ],
[13.275301 ],
[11.840857 ],
[10.998472 ],
[12.097724 ],
[10.41459  ],
[ 9.102376 ],
[ 9.401342 ],
[ 9.943217 ],
[ 8.358591 ],
[10.708942 ],
[12.577156 ],
[11.904816 ],
[12.178363 ],
[14.685263 ],
[14.933663 ],
[13.923521 ],
[14.550435 ],
[13.552047 ],
[11.646476 ],
[11.851775 ],
[11.61521  ],
[10.606009 ],
```

```
#testing by inputing custome epoch count
# train the model

history = rnnModel.fit(Xtrain3D, Ytrain, epochs = 20, batch_size=30, validation_split=0.1, verbose=1 )
```

```
Epoch 1/20
88/88 [==============================] - 1s 10ms/step - loss: 6.9318 - mae: 2.0448 - val_loss: 8.8408 - val_mae: 2.3124
Epoch 2/20
88/88 [==============================] - 1s 10ms/step - loss: 6.8936 - mae: 2.0398 - val_loss: 9.5239 - val_mae: 2.4231
Epoch 3/20
```

```
88/88 [==============================] - 1s 9ms/step - loss: 6.9228 - mae: 2.0513 - val_loss: 9.0181 - val_mae: 2.3324
Epoch 4/20
88/88 [==============================] - 1s 7ms/step - loss: 6.9478 - mae: 2.0556 - val_loss: 9.0896 - val_mae: 2.3654
Epoch 5/20
88/88 [==============================] - 1s 7ms/step - loss: 6.9068 - mae: 2.0462 - val_loss: 8.8282 - val_mae: 2.2915
Epoch 6/20
88/88 [==============================] - 1s 7ms/step - loss: 6.8833 - mae: 2.0407 - val_loss: 8.9437 - val_mae: 2.3358
Epoch 7/20
88/88 [==============================] - 1s 6ms/step - loss: 6.8873 - mae: 2.0421 - val_loss: 8.7372 - val_mae: 2.3001
Epoch 8/20
88/88 [==============================] - 1s 6ms/step - loss: 6.9071 - mae: 2.0413 - val_loss: 9.3191 - val_mae: 2.3904
Epoch 9/20
88/88 [==============================] - 1s 7ms/step - loss: 6.8040 - mae: 2.0276 - val_loss: 9.0630 - val_mae: 2.3461
Epoch 10/20
88/88 [==============================] - 1s 7ms/step - loss: 6.7395 - mae: 2.0204 - val_loss: 8.8100 - val_mae: 2.3234
Epoch 11/20
88/88 [==============================] - 1s 6ms/step - loss: 6.7999 - mae: 2.0308 - val_loss: 8.9976 - val_mae: 2.3366
Epoch 12/20
88/88 [==============================] - 1s 6ms/step - loss: 6.7569 - mae: 2.0201 - val_loss: 9.0420 - val_mae: 2.3454
Epoch 13/20
88/88 [==============================] - 1s 6ms/step - loss: 6.7612 - mae: 2.0160 - val_loss: 8.9388 - val_mae: 2.3302
Epoch 14/20
88/88 [==============================] - 1s 6ms/step - loss: 6.7523 - mae: 2.0216 - val_loss: 8.9935 - val_mae: 2.3238
Epoch 15/20
88/88 [==============================] - 1s 6ms/step - loss: 6.7183 - mae: 2.0103 - val_loss: 9.1064 - val_mae: 2.3306
Epoch 16/20
88/88 [==============================] - 1s 6ms/step - loss: 6.7434 - mae: 2.0210 - val_loss: 9.1993 - val_mae: 2.3673
Epoch 17/20
88/88 [==============================] - 1s 6ms/step - loss: 6.7357 - mae: 2.0182 - val_loss: 9.3651 - val_mae: 2.3761
Epoch 18/20
88/88 [==============================] - 1s 7ms/step - loss: 6.7238 - mae: 2.0165 - val_loss: 9.5393 - val_mae: 2.4059
Epoch 19/20
88/88 [==============================] - 1s 7ms/step - loss: 6.6859 - mae: 2.0108 - val_loss: 9.3018 - val_mae: 2.4042
Epoch 20/20
88/88 [==============================] - 1s 6ms/step - loss: 6.6536 - mae: 2.0051 - val_loss: 9.1392 - val_mae: 2.3306
```

chnaging the previous days and after days

```
#changing the parameters and givng previous days.
previousDays = 10
afterDays = 2

X = createDelayedColumns(dailyTemps,
        times=range(-previousDays+1,1)).iloc[previousDays:-afterDays]
y = createDelayedColumns(dailyTemps,
        times=[afterDays]).iloc[previousDays:-afterDays]

print(X.head())
print(y.head())

print(X.shape)
print(y.shape)
```

```
             t-9   t-8   t-7   t-6   t-5   t-4   t-3   t-2   t-1     t
            Temp  Temp  Temp  Temp  Temp  Temp  Temp  Temp  Temp  Temp
Date
1981-01-11  17.9  18.8  14.6  15.8  15.8  15.8  17.4  21.8  20.0  16.2
1981-01-12  18.8  14.6  15.8  15.8  15.8  17.4  21.8  20.0  16.2  13.3
1981-01-13  14.6  15.8  15.8  15.8  17.4  21.8  20.0  16.2  13.3  16.7
1981-01-14  15.8  15.8  15.8  17.4  21.8  20.0  16.2  13.3  16.7  21.5
1981-01-15  15.8  15.8  17.4  21.8  20.0  16.2  13.3  16.7  21.5  25.0
             t+2
            Temp
Date
1981-01-11  16.7
1981-01-12  21.5
1981-01-13  25.0
1981-01-14  20.7
1981-01-15  20.6
(3638, 10)
(3638, 1)
```

```
train_slice = slice(None, '1988-12-28')
test_slice = slice('1989-01-01', None)

Xtrain, Ytrain = X.loc[train_slice], y.loc[train_slice]
Xtest, Ytest = X.loc[test_slice], y.loc[test_slice]

print(Xtrain.shape)
print(Xtest.shape)
```

```
(2908, 10)
(728, 10)
```

```
Xtrain3D = convert3D(Xtrain)
Xtest3D = convert3D(Xtest)

print(Xtrain3D.shape)
print(Xtest3D.shape)
```

```
(2908, 10, 1)
(728, 10, 1)
```

```
rnnModel = keras.models.Sequential()

rnnModel.add(keras.layers.SimpleRNN(50, return_sequences=True,
                                    input_shape=(10,1)))

rnnModel.add(keras.layers.SimpleRNN(25))

rnnModel.add(keras.layers.Dense(1))

rnnModel.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_4 (SimpleRNN)    (None, 10, 50)            2600

 simple_rnn_5 (SimpleRNN)    (None, 25)                1900

 dense_2 (Dense)             (None, 1)                 26

=================================================================
Total params: 4526 (17.68 KB)
Trainable params: 4526 (17.68 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
# Configure  the model for training, by using appropriate optimizers and regularizations
# Available optimizer: adam, rmsprop, adagrad, sgd
# loss:  objective that the model will try to minimize.
# Available loss: categorical_crossentropy, binary_crossentropy, mean_squared_error
# metrics: List of metrics to be evaluated by the model during training and testing.

rnnModel.compile(loss='mse', optimizer='adam', metrics=['mae'])
```

```
# train the model

history = rnnModel.fit(Xtrain3D, Ytrain, epochs = 50, batch_size=20, validation_split=0.1, verbose=1 )
```

```
Epoch 39/50
131/131 [==============================] - 1s 9ms/step - loss: 7.9715 - mae: 2.2088 - val_loss: 8.2141 - val_mae: 2.1921
Epoch 40/50
131/131 [==============================] - 1s 7ms/step - loss: 8.0038 - mae: 2.2129 - val_loss: 7.8737 - val_mae: 2.1738
Epoch 41/50
131/131 [==============================] - 1s 7ms/step - loss: 7.9538 - mae: 2.2041 - val_loss: 8.2144 - val_mae: 2.2039
Epoch 42/50
131/131 [==============================] - 1s 7ms/step - loss: 7.9787 - mae: 2.2044 - val_loss: 8.1702 - val_mae: 2.1781
Epoch 43/50
131/131 [==============================] - 1s 7ms/step - loss: 7.8962 - mae: 2.2045 - val_loss: 8.1952 - val_mae: 2.2326
Epoch 44/50
131/131 [==============================] - 1s 7ms/step - loss: 7.9177 - mae: 2.2132 - val_loss: 8.7562 - val_mae: 2.2524
Epoch 45/50
131/131 [==============================] - 1s 7ms/step - loss: 7.9870 - mae: 2.2106 - val_loss: 7.9635 - val_mae: 2.1856
Epoch 46/50
131/131 [==============================] - 1s 8ms/step - loss: 7.8959 - mae: 2.1913 - val_loss: 7.8438 - val_mae: 2.1533
Epoch 47/50
131/131 [==============================] - 1s 7ms/step - loss: 7.8720 - mae: 2.1973 - val_loss: 8.1316 - val_mae: 2.1942
Epoch 48/50
131/131 [==============================] - 1s 8ms/step - loss: 7.8271 - mae: 2.1937 - val_loss: 8.1309 - val_mae: 2.1946
Epoch 49/50
```

Learning Work Embeddings

```
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

import pandas as pd

import tensorflow as tf
from tensorflow import keras


tf.set_random_seed(42)
```

Dataset - IMDB

```
imdb = keras.datasets.imdb

max_features = 20000

(Xtrain, Ytrain), (Xtest, Ytest) = imdb.load_data(num_words = max_features)


# Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
# 17465344/17464789 [==============================] - 8s 0us/step


print(len(Xtrain), len(Ytrain))
print(len(Xtest), len(Ytest))

Xtrain = keras.preprocessing.sequence.pad_sequences(Xtrain, maxlen =25)
Xtest = keras.preprocessing.sequence.pad_sequences(Xtest, maxlen =25)

print(len(Xtrain), len(Ytrain))
print(len(Xtest), len(Ytest))
```

```
25000 25000
25000 25000
25000 25000
25000 25000
```

LSTM

```
lstmModel = keras.models.Sequential()

lstmModel.add(keras.layers.Embedding(input_dim = max_features, output_dim = 128))

lstmModel.add(keras.layers.LSTM(128, dropout=0.2)) #, recurrent_dropout=0.2

lstmModel.add(keras.layers.Dense(1, activation = 'sigmoid'))

lstmModel.summary()
```

```
WARNING:tensorflow:From C:\Anaconda3\lib\site-packages\tensorflow\python\keras\initializers.py:119: calling RandomUniform.__init__
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From C:\Anaconda3\lib\site-packages\tensorflow\python\ops\init_ops.py:1251: calling VarianceScaling.__init__ (fro
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 128)         2560000
_________________________________________________________________
lstm (LSTM)                  (None, 128)               131584
_________________________________________________________________
dense (Dense)                (None, 1)                 129
=================================================================
Total params: 2,691,713
Trainable params: 2,691,713
Non-trainable params: 0
_________________________________________________________________
```

```
# Configure  the model for training, by using appropriate optimizers and regularizations
# Available optimizer: adam, rmsprop, adagrad, sgd
# loss:  objective that the model will try to minimize.
# Available loss: categorical_crossentropy, binary_crossentropy, mean_squared_error
# metrics: List of metrics to be evaluated by the model during training and testing.

lstmModel.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

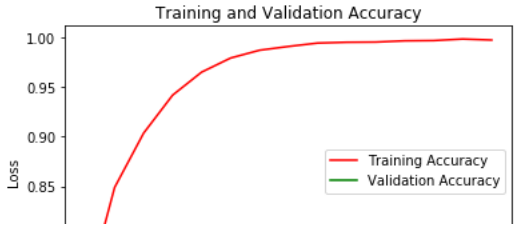
```
WARNING:tensorflow:From C:\Anaconda3\lib\site-packages\tensorflow\python\ops\nn_impl.py:180: add_dispatch_support.<locals>.wrapper
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
```

```
# train the model

history = lstmModel.fit(Xtrain, Ytrain, epochs = 15, batch_size=16, validation_split=0.2, verbose=1 )
```

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/15
20000/20000 [==============================] - 112s 6ms/sample - loss: 0.5103 - acc: 0.7393 - val_loss: 0.4768 - val_acc: 0.7608
Epoch 2/15
20000/20000 [==============================] - 110s 6ms/sample - loss: 0.3454 - acc: 0.8489 - val_loss: 0.5016 - val_acc: 0.7602
Epoch 3/15
20000/20000 [==============================] - 120s 6ms/sample - loss: 0.2330 - acc: 0.9035 - val_loss: 0.5999 - val_acc: 0.7616
Epoch 4/15
20000/20000 [==============================] - 133s 7ms/sample - loss: 0.1496 - acc: 0.9417 - val_loss: 0.7581 - val_acc: 0.7544
Epoch 5/15
20000/20000 [==============================] - 142s 7ms/sample - loss: 0.0917 - acc: 0.9649 - val_loss: 0.8958 - val_acc: 0.7588
Epoch 6/15
20000/20000 [==============================] - 150s 7ms/sample - loss: 0.0599 - acc: 0.9791 - val_loss: 1.1558 - val_acc: 0.7448
Epoch 7/15
20000/20000 [==============================] - 167s 8ms/sample - loss: 0.0394 - acc: 0.9869 - val_loss: 1.1964 - val_acc: 0.7416
Epoch 8/15
20000/20000 [==============================] - 174s 9ms/sample - loss: 0.0282 - acc: 0.9908 - val_loss: 1.3123 - val_acc: 0.7470
Epoch 9/15
20000/20000 [==============================] - 184s 9ms/sample - loss: 0.0180 - acc: 0.9941 - val_loss: 1.4391 - val_acc: 0.7364
Epoch 10/15
20000/20000 [==============================] - 203s 10ms/sample - loss: 0.0147 - acc: 0.9949 - val_loss: 1.7328 - val_acc: 0.7364
Epoch 11/15
20000/20000 [==============================] - 221s 11ms/sample - loss: 0.0142 - acc: 0.9952 - val_loss: 1.6748 - val_acc: 0.7406s
Epoch 12/15
20000/20000 [==============================] - 229s 11ms/sample - loss: 0.0111 - acc: 0.9964 - val_loss: 1.6143 - val_acc: 0.742600
Epoch 13/15
20000/20000 [==============================] - 240s 12ms/sample - loss: 0.0096 - acc: 0.9966 - val_loss: 1.8079 - val_acc: 0.737411
Epoch 14/15
20000/20000 [==============================] - 252s 13ms/sample - loss: 0.0055 - acc: 0.9982 - val_loss: 1.7984 - val_acc: 0.7328
Epoch 15/15
20000/20000 [==============================] - 266s 13ms/sample - loss: 0.0089 - acc: 0.9972 - val_loss: 1.7902 - val_acc: 0.7332
```

```
# plotting training and validation loss

loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, color='red', label='Training loss')
plt.plot(epochs, val_loss, color='green', label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
# plotting training and validation Accuracy

acc = history.history['acc']
val_acc = history.history['val_acc']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, acc, color='red', label='Training Accuracy')
plt.plot(epochs, val_acc, color='green', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```




```
testLoss, testAccuracy = lstmModel.evaluate(Xtest, Ytest)
print(testLoss, testAccuracy)
```

```
25000/25000 [==============================] - 18s 704us/sample - loss: 1.7381 - acc: 0.7378
1.7381113950538636 0.73784
```