Introduction to Neural Networks

I thought I won't make a new article for this, Just as a introduction to Neural Networks. Neural networks needs a long mathematical explanation. In this article I'll not go much deep in the mathematics behind this. But just Enough to get you started with Neural Networks and `tensorflow`.

- LinkedIn
- YouTube
- gtihub
- Gmail
- discord

Introduction

`Neural Networks` are a set of `algorithms`, modeled loosely after the `human brain`, that are designed to `recognize patterns`. They `interpret sensory data` through a kind of `machine perception`, `labeling`, or `clustering` of raw input. The patterns they recognize are `numerical` and contained in `vectors`, into which all real-world data (images, sound, text, etc.) must be `translated`.

> Without any of the weird words, Neural Networks are a `set of algorithms` that help us to `model complex patterns` and `prediction problems` that are `non-linear` in nature.

Perceptron Model

To fully unserstand `neural networks`, we need to understand the `Perceptron Model`.

A `perceptron` is the most basic `building block` of an `artificial neural network`. Invented by Frank Rosenblatt in 1957.

It is a simplified mathematical model of a single biological neuron.

You can think of a `Perceptron` as a `single neuron` of a `Human Brain`. It takes `input`, processes it, and provides `output`.

If you think of a `Perceptron` as a `single neuron`, this is how it looks like:

It is the `simplest form` of a `neural network`.

Amazingly, even back almost `70 years ago`, `Frank Rosenblatt` was able to see the potential of `neural networks` and stated that...

> "The `Perceptron` may eventually be able to `learn`, `make decisions`, and `translate languages`."

Which is exactly what we are doing today with `Deep Learning` and `Neural Networks` and `Machine Learning` and `Artificial Intelligence` and in the base of all these, there is the `Perceptrons`.

And this is why it is important to understand the `Perceptron` model.

But it was not until the `1980s` that `Neural Networks` started to become popular again. Because in `1969`, `Marvin Minsky` and `Seymour Papert` published a book called `Perceptrons` which showed the `limitations` of `Perceptrons`. They showed that `Perceptrons` were `limited` to `linearly separable problems`. The book also suggested that `Perceptrons` were `severely limited` in what they could learn.

This Statement by `Marvin Minsky` and `Seymour Papert` led to the `first AI winter` where `funding` and `interest` in `Neural Networks` and `AI` dried up.

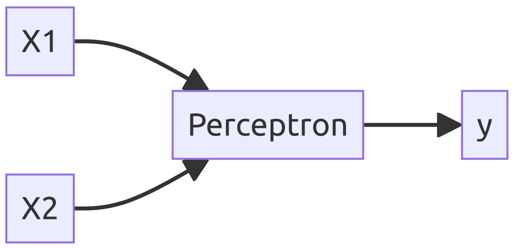
SO, back in the `1970s` There was not enough `computational power` to `train` Neural Networks and Perceptrons were limited to linearly separable problems.

But in the `1980s`, `Neural Networks` started to become popular again with the `discovery` of the `backpropagation algorithm` which allowed `Neural Networks` to learn `non-linear functions`.

Perceptrons

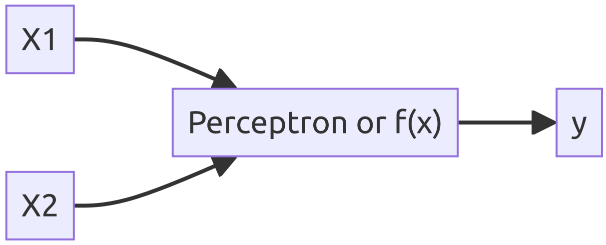
Now again think of the `screen` you are looking at right now. It is made up of `pixels`. Each `pixel` has a `color` and `intensity`. If you think of a `pixel` as a `feature`, then a `Perceptron` can be used to `classify` the `pixel` as `black` or `white`. SO, by analyzing the the screen we can see, almost always, we are taking `multiple inputs` and `processing` them to get `output`.

So, now suppose we have `two inputs` (X1, X2) and we want to `classify` them as `black` or `white` which we will represent as `y`. We can represent this as:



Now, the `Perceptron` will take `input` from `X1` and `X2`, process it, and provide `output` as `y`.

We can represent the `Perceptron` as a `mathematical model` or a `function`. So, the final `perceptron` will look like this:

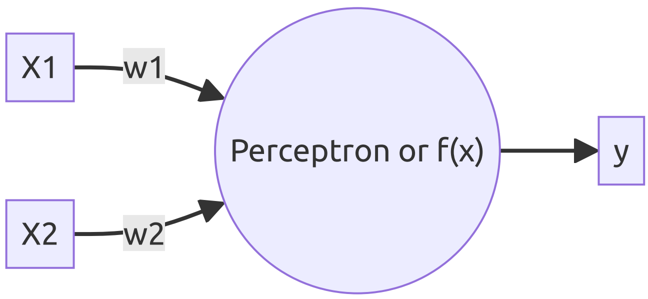


So, now if I say that the `f(x)` is a `sum` then,

$$f(x) = x_1 + x_2$$

Simple as that. This is the most simplistic form of a `Perceptron`. It takes `input`, `processes` it, and provides `output`.

Now Realistically, a we want to be able to adjust some `parameters` of the `Perceptron` to make it more `flexible` and `powerful`. In the above model there is no flexibility for the Perceptron to learn from the data. So, what we can do is add a `weight` to the `input` which will help the `Perceptron` to `learn` from the `data` more `efficiently`. So, we will add the `weights` to the `input` as a `multiplication` factor. So, the `Perceptron` will look like this:



So, the `Perceptron` will take `input` from `X1` and `X2`, `multiply` it with `weights` `w1` and `w2`, `process` it, and provide `output` as `y`.

$$y = w_1x_1 + w_2x_2$$

So, now we can `adjust` the `weights` to make the Perceptron more accurate and select the `weights` that `minimize` the `error`. This will give us more `flexibility` to the `Perceptron` to `learn` from the `data`.

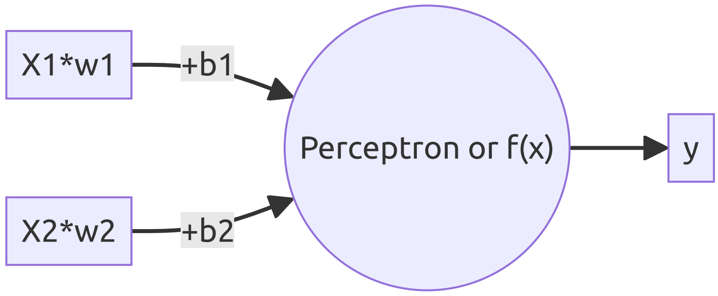
> We can think of the `weights` as `parameters` of the Perceptron that we can adjust to make the Perceptron more accurate.

Now here's a brain teaser for you. What if the actual `input` is `X1` or `X2` is `0`? The `weights` will not have any effect on the `output`. How do we fix this?

The answer is to add a `bias` to the `Perceptron`.

> A `bias` is a constant that helps the `Perceptrons` input to have a `non-zero` value.

Bias is a `constant` That we can add to the input to make sure that the input is never `0`. So, the weights can have an effect on the output. We can represent the `bias` as `b`. And the `Perceptron` will look like this:



So, the `Perceptron` will take `input` from `X1` and `X2`, `multiply` it with `weights` `w1` and `w2`, `add` the `bias` `b`, `process` it, and provide `output` as `y`.

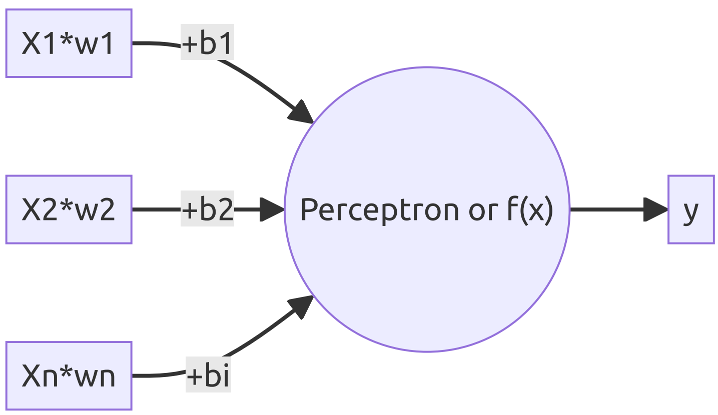
$$y = (w_1x_1 + b_1) + (w_2x_2 + b_2)$$

So, now we can `adjust` the `weights` and `bias` to make the `Perceptrons` more `flexible` even with the `input` as `0`.

We can `generalize` the `Perceptron` as:

$$y = w_1x_1 + w_2x_2 + b$$

Now think of `n` number of `inputs`. We can represent the `Perceptron` as:



Well, here's is the `mathematical model` of the `Perceptron` with `n` number of `inputs` and we can finally generalize the `Perceptron` as:

$$y = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

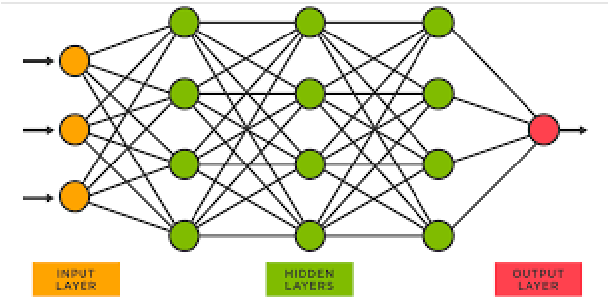
$$y = \sum_{i=1}^{n} w_ix_i + b$$

> b is the total bias of all the inputs.

With this we can have modeled a `biological neuron` as a `simple perceptron`. Later in this article, we will see how we can `expand` the `input` be a `tensor` of information(n-dimensional array) and how we can `expand` the `Perceptron` to a `Neural Network`.

Artificial Neural Networks

A neural network is a `multi-layer perceptron` model that is designed to `recognize patterns` in the `data`.



A `Neural Network` is made up of `layers` of `Perceptrons`. You can clearly see that there are `five layers` in the `Neural Network` above.

- The `first layer` is called the `input layer` and it takes the `input` from the `data`.
- The `middle layers` are called the `hidden layers` and these layers `learn` to `recognize patterns` in the `data`.
- The `last layer` is called the `output layer` and it `provides` the `output` from the `hidden layer`.

Every `Perceptron` in a `layer` is `connected` to every `Perceptron` in the `next layer` and works as a `input` for the `next layer` and so on.

This `multi-layer perceptron` model is called a `Feedforward Neural Network` because the `input` is `fed forward` through the `network` and the `output` is `provided` at the `end`.

Every `Neural Network` consists of `three layers`:

- `Input Layer`
- `Hidden Layer`
- `Output Layer`

> Input Layer: The `input layer` is the `first layer` of the `Neural Network`. It can have `n` number of `neurons/perceptrons` where `n` is the number of `features` in the `data`.

> Output Layer: The `output layer` is the `last layer` of the `Neural Network`. It can have `m` number of `neurons/perceptrons` where `m` is the number of `classes` in the `data` meaning the number of `output` we might have. Also if the output is numerical then the `output layer` is called the `regression layer` and it can only have one `neuron`.

> `Hidden Layer`: The `hidden layer` is the `middle layer` of the `Neural Network`. A hidden layer can have `n` number of `layers` and it's because of the `hidden layer` that the `Neural Network` can `learn` `complex patterns`. The `hidden layer` is the `magic` of the `Neural Network` where all the `processing` is done and the `patterns` are `learned`.

`Hidden layers` are hard to `interpret` and `understand` because every `Perceptron` in the `hidden layer` takes input from every `Perceptron` in the `previous layer`, so the internal workings become more and more mathematically complicated. As the layers get deeper, the internal workings become more and more mathematically complicated.

> Hidden layers have high connectivity and are distantly connected to the input and output layers.

> A `Neural Network` becames a `Deep Neural Network` when it has more than one `hidden layer`.

So, the picture above is a `Feedforward Neural Network` with `three layers` and three `hidden layers`. So, we can say that this is a `Deep Neural Network`.

Previously, we have seen that a `Perceptron` can be represented as a `mathematical model` or a `function` like:

$$y = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

But this is a linear function and if we think of this as a `Neural Network`, it'll still be a `linear function`. But a dataset with non linear values cannot actually be represented by a `linear function`.

So, we need to `generalize` the `Perceptrons` to `non-linear` values.

> We can do that with `Activation Functions`.

Activation Functions

The `W` clearly implies the weight of incoming `input`.

`b` is the `bias` value which is a constant value that helps the `Perceptrons` input to have a `non-zero` value. We can also think of the `bias` as a `offset` value that makes **xw** have to reach a certain threshold before it can have any effect on the output.

For example if `b=-10` in `x*w+b` then the effects of `x*w` won't really start to overcome the bias until `x*w` is greater than `10`.

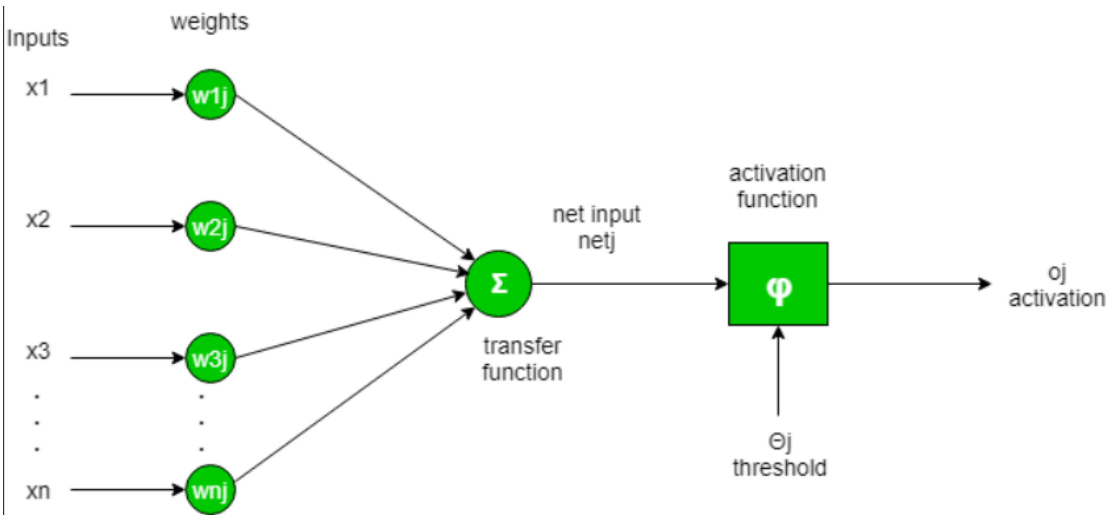
After that, the effect will solely be determined by the value of `w` or the weight of the input.

That's why it's called a `bias` because it can `bias` the output of the `Perceptron` to be `positive` or `negative` depending on the value of the `bias`.

Now, the `output` of the `Perceptron` is `linear` in nature. It is a `straight line` and it can only `classify` `linearly separable` problems. But in the real world, we have `non-linear` problems that cannot be `classified` by a `linear Perceptron`.

So, we need to `introduce non-linearity` to the `output` of the `Perceptron` to `classify` `non-linear` problems. This is where the `Activation Functions` come in.

> To be completely dumb, you can say that Activation Functions are functions that are added to the outputs of the Perceptrons to transform the output into a non-linear form.



There are many `Activation Functions` that are used in `Neural Networks`. Some of the most popular `Activation Functions` are:

- `Sigmoid` (Logistic Function)
- `Tanh` (Hyperbolic Tangent)
- `ReLU` (Rectified Linear Unit)

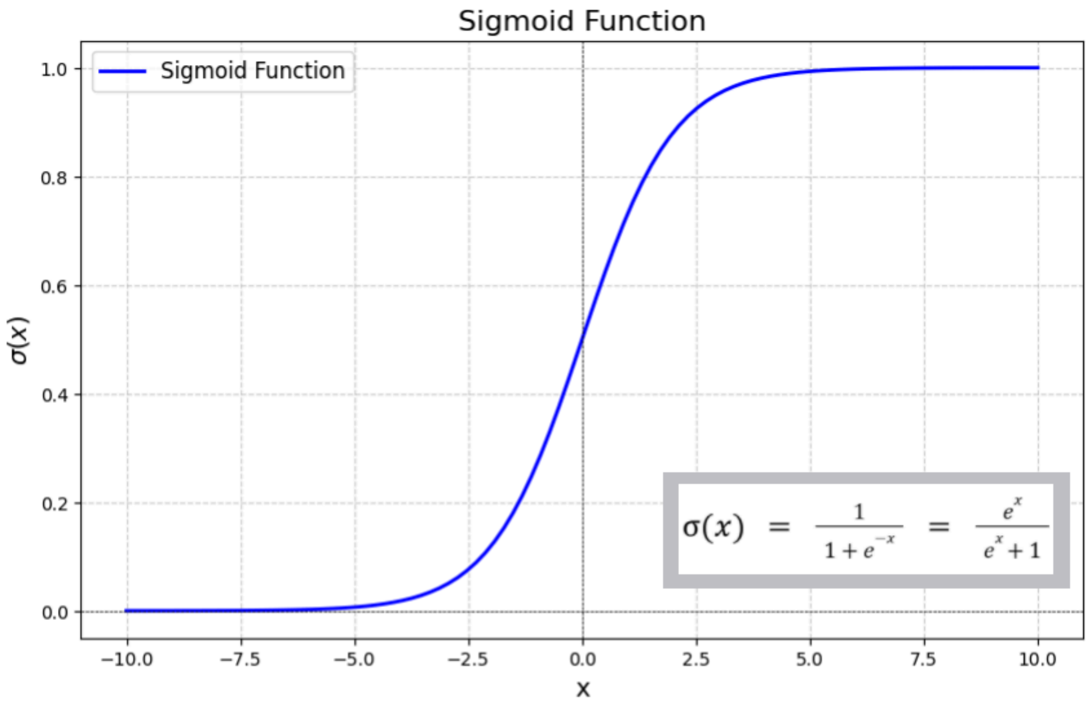
Sigmoid Activation Function

The `Sigmoid Activation Function` is a `non-linear` function that is used in `binary classification` problems. It is also called the `Logistic Function`.

The `Sigmoid Activation Function` is represented as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

The `Sigmoid Activation Function` takes the `output` of the `Perceptron` and `squashes` it between `0` and `1`. This is useful in `binary classification` problems where we need the `output` to be between `0` and `1`. You have done `logistic regression` before, right? It's the same thing.



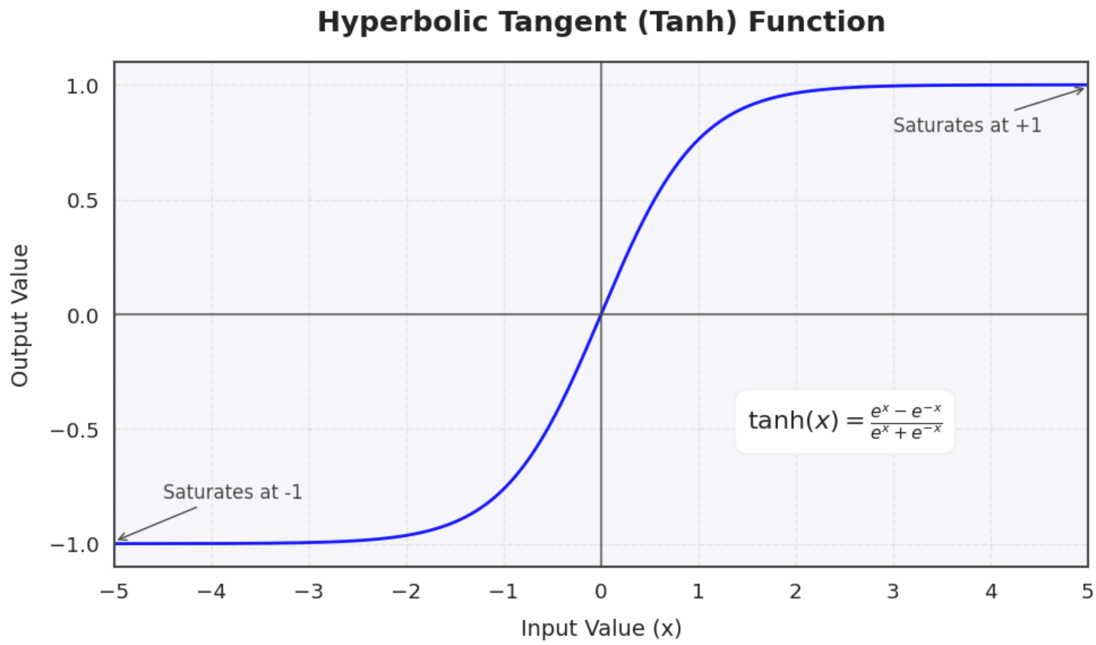
Tanh Activation Function

It is also called the `Hyperbolic Tangent`.

The `Tanh Activation Function` is represented as:

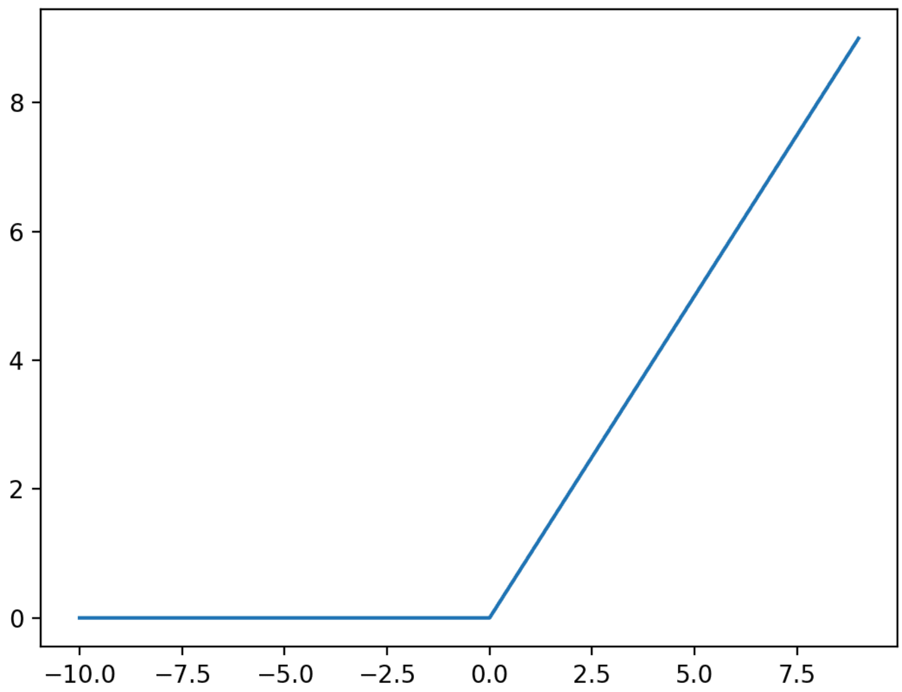
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The `Tanh Activation Function` takes the `output` of the `Perceptron` and `squashes` it between `-1` and `1`. This is useful in `binary classification` problems where we need the `output` to be between `-1` and `1`.



ReLU Activation Function

One the most popular and widely used Activation Function is the `ReLU Activation Function`. `ReLU` stands for `Rectified Linear Unit`.



ReLU is a `linear` function that `returns` `0` if the `output` is `less than 0` and `returns` the `output` if the `output` is `greater than 0` as shown in the picture above.

The `ReLU Activation Function` is represented as:

$$f(x) = max(0, x)$$

This function shows that the `output` will only be `0` if the `output` is `less than 0`.

The `ReLU Activation Function` is used in `Deep Neural Networks` because it is `computationally efficient` and it `speeds up` the `training` of the `Neural Network`. There have been many `improvements` to the `ReLU Activation Function` like the `Leaky ReLU` and the `Parametric ReLU` and there are still research going on to `improve` the `ReLU Activation Function`.

There are many more `Activation Functions` that are used in `Neural Networks`. But these are the most popular `Activation Functions` that are used in `Deep Neural Networks`.

You can visit the `wiki` page of `Activation Functions` to learn more about the `Activation Functions` that are used in `Neural Networks`. (https://en.wikipedia.org/wiki/Activation_function)

Now the next part is very critical and without any way to visualize it's pretty hard to explain so I'll go thorugh them one by one and try my best to explain them also I will attach some links so that you can learn more about them.

Multi-Class Classification

`Multi-class classification` means the `target` has **more than two classes**.

There are two types:

Non-Exclusive Classes

A data point can belong to **more than one class**. Example: an image can be both `cat` and `black`.

This is called **multi-label classification**.

Mutually Exclusive Classes

A data point can belong to **only one class**. Example: an image is either `cat` or `dog`.

This is the standard **multi-class classification** problem.

Output Layer Design

If the target has `n` classes, the neural network must have **`n` output neurons**.

The output is a **vector**, where each value represents how strongly the model believes the input belongs to a class.

So instead of:

```
red, blue, green
```

we want:

```
[0.7, 0.2, 0.1]
```

These are **scores**, which we later convert into **probabilities**.

One-Hot Encoding

Neural networks cannot learn from labels like `red`, `blue`, `green`.

So we convert them into vectors using `one-hot encoding`.

label	red	blue	green
red	1	0	0
blue	0	1	0
green	0	0	1

Now the model learns **numbers**, not words.

Output Activation

For Non-Exclusive Classes → `Sigmoid`

Each output neuron is independent. The model predicts a probability for **each class separately**.

For Mutually Exclusive Classes → `Softmax`

`Softmax` converts raw scores into **normalized probabilities** that:

- are between `0` and `1`
- always **sum to `1`**

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

This forces the network to **choose one class**.

Cost Function & Learning

The neural network does not know if it is correct. So we use a `cost function` to measure **how wrong it is**.

It compares:

- `y` → actual value
- `a` → predicted value

And returns **one scalar number**: the error.

Training means:

> **change the weights to minimize this number.**

Common Cost Functions

Mean Squared Error (MSE)

$$C = \frac{1}{2n} \sum (y - a)^2$$

Used mostly for `regression`.

Large mistakes are punished more because of the square.

Cross-Entropy Loss

For classification:

$$C = -\sum y \log(a)$$

It strongly punishes **confident but wrong predictions**. That is why it works better with `Sigmoid` and `Softmax`.

Gradient Descent

The `cost function` depends on **thousands of weights**. So the error surface is not a curve — it is an **n-dimensional landscape**.

We cannot visualize it. So we **walk downhill** using derivatives.

That is `Gradient Descent`.

$$w = w - \alpha \frac{\partial C}{\partial w}$$

- `α` is the `learning rate`
- The derivative tells us the **direction of steepest increase**
- We move in the **opposite direction**

Learning Rate

- Too small → slow

- Too large → overshoot and diverge

So the learning rate must be **tuned**.

Adaptive Optimizers

Instead of using a fixed learning rate, modern optimizers **adapt** it:

- `Adagrad`
- `RMSprop`
- `Adam`

`Adam` combines momentum and adaptive scaling. That's why it is the default in most frameworks.

Backpropagation (What Actually Happens)

Training has two steps:

1. Forward Pass

Input flows through the network. We compute predictions and the `cost`.

2. Backward Pass

We compute:

$$\frac{\partial C}{\partial w}$$

for **every weight** using the `chain rule`.

Then we update the weights using `gradient descent`.

This repeats until the cost stops decreasing.

Final Truth

Neural networks **do not think**.

They:

1. Guess
2. Measure error
3. Adjust weights
4. Repeat

That's it.

Here are some sources for more information:

MATH:

- [Backpropagation Algorithm(WIKIPEDIA)]

VIDEO:

- Whats really happening in Backpropagation(Channel: 3Blue1Brown)
- Backpropagation Calculus(Channel: 3Blue1Brown)
- Neural Networks Playlist(Channel: 3Blue1Brown)
- Neural networks from scratch

So, that's it for the `Backpropagation` algorithm. I tries my best to cover all internal concepts of the `Neural Network`. But without proper way to show you guys there's some things that is not possible for me to cover. With that said,

Now, let's move on to the `Neural Network` implementation in `Python` using the `TensorFlow` and `Keras` libraries.

Neural Network With Python

Introduction

In this section, we will implement a `Neural Network` in `Python` using the `TensorFlow` and `Keras` libraries. These two are the most popular libraries for `Deep Learning` and `Neural Networks`.

But Why two libraries?

TensorFlow and Keras

- `TensorFlow` is a `high-level` `API` for `Deep Learning`.
- `Keras` is a `high-level` `API` for `Neural Networks`.
- `TensorFlow` is a `low-level` `API` for `customization` and `control` over the `neural network` `architecture`.
- `Keras` is a `high-level` `API` for `quick` `prototyping` and `experimentation`.

Buuuuuut, these two are now in the same family. Keres was made using `TensorFlow` to implement `Neural Networks` faster.

In 2015 Google released `TensorFlow` and `Keras` in the same year. But these two were separate libraries. Keras used TensorFlow's `API` to implement `Neural Networks` faster. Then in 2017 google officially adopted `keras` as tensorflows `high-level` and user friendly `API` for `Neural Networks`.

So, we can do most of the things we need to implement a `Neural Network` in `Python` using `tf.keras`.

Installation

It's easy to `install` the `TensorFlow` library using the `pip` `package manager` or `conda` `package manager`.

You can get more information about the `installation` of `TensorFlow` from the `official` `TensorFlow` `website`. (https://www.tensorflow.org/install/pip)

Here's how you can `install` the `TensorFlow` library using the `pip` `package manager`:

```
python -m pip install tensorflow
```

> In the official website, they said not to use conda to install tensorflow. So, I will listen to the experts and not install it using conda.

For those who has a `nvidia` `GPU`, you can `install` the library using this command:

```
pip install tensorflow[and-cuda]
```

> Remember tensorflow is a huuuge library... Close to 5GB. So, be sure that u have enough space to install it.

Introduction to Tensorflow

Think of it as a more powerfull version of `numpy` with GPU support. More moduler, more functions, more features.

Think of `keras` as `sklearn` but for `Neural Networks`. It has almost everything you need to implement, every type of `Neural Networks` (yes! there are different types).

But for data preprocessing and other things, we can still use numpy and pandas because tensorflow is fully compatible with numpy and pandas.

Now that we have `installed` the `TensorFlow` library, let's `import` the `TensorFlow` library and test if it is `installed` correctly.

In [1]:
```python
import tensorflow as tf

print(tf.__version__)
```

```
2026-02-04 17:24:06.254675: I tensorflow/core/platform/cpu_feature_guard.c
c:210] This TensorFlow binary is optimized to use available CPU instruction
s in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuil
d TensorFlow with the appropriate compiler flags.
2.20.0
```

If you installed the `gpu` version run the code below.

In [2]:
```python
import tensorflow as tf

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

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

For, cpu testing, run the code below.

In [3]:
```python
tf.reduce_sum(tf.random.normal([1000, 1000]))
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1770204248.207698  501784 gpu_device.cc:2020] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 9653 MB memory:  -> device: 0, name: NVIDIA GeForce RTX 3060, pci bus id: 0000:05:00.0, compute capability: 8.6
```

Out[3]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=1471.2747802734375>
```

Noice!

I have the `2.20` version and my gpu is `detected`. So, everything is `working` fine.

Now, let's see how good a neural network can be.

Let's Make A Neural Network

First we need data.

In [4]:
```python
import numpy as np
import pandas as pd

df = pd.read_csv("./custom_price_dataset.csv")
df.head()
```

Out[4]:

	feature_1	feature_2	price
0	0.248357	0.699678	55278.192461
1	-0.059122	0.472327	54599.930210
2	0.343864	0.049835	54572.806367
3	0.791545	-0.293438	56222.631937
4	-0.077037	0.389152	51960.208200

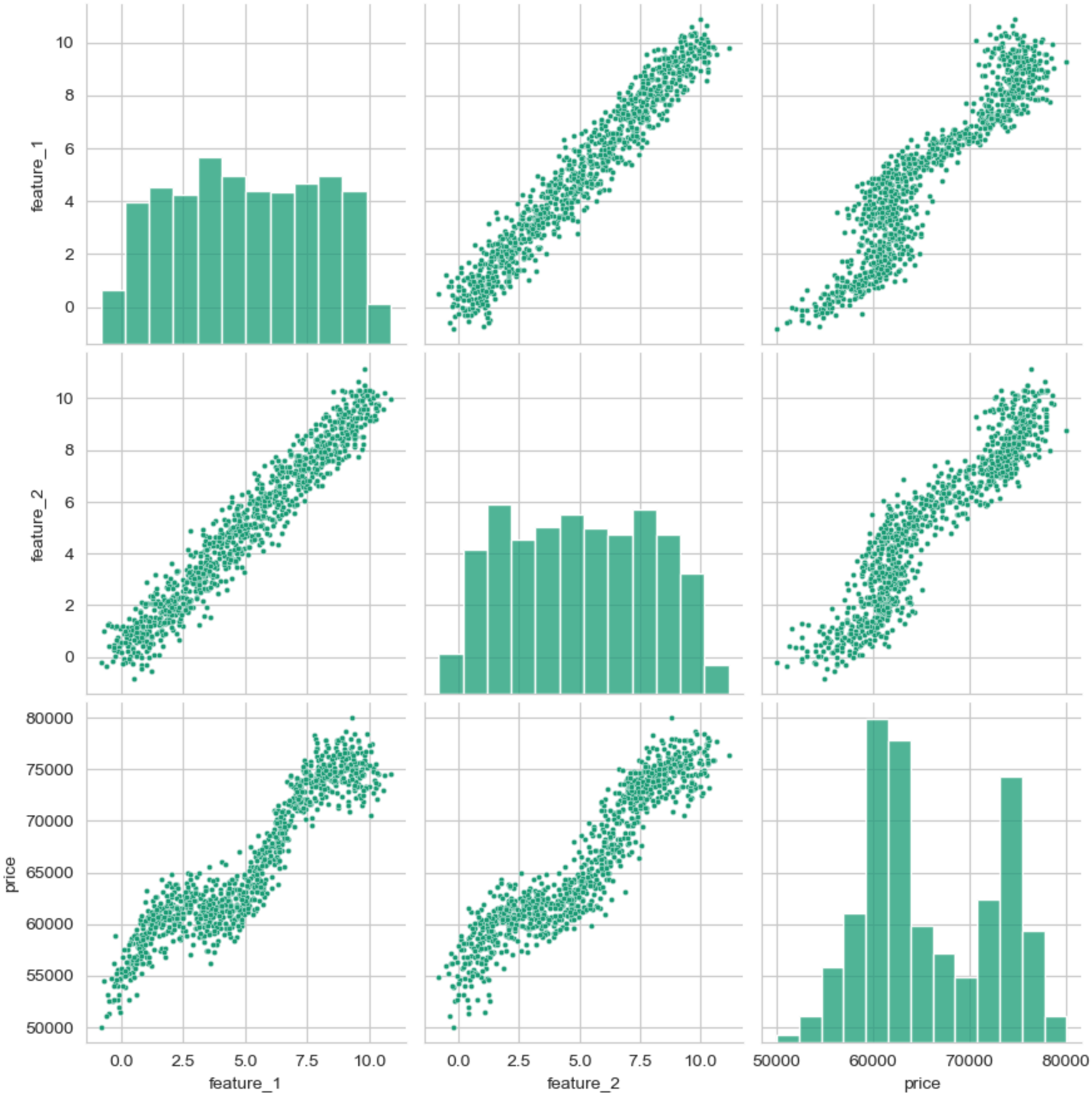
This is a `fake_dataset` where you are given prices according to the `features`. SO, you can see that the dataset has three features, `feature1`, `feature2`, and `price`. The `price` is the `target` that we need to `predict` using the `features`.

We can do some visualizations to understand the `data` better.

In [5]:
```python
import seaborn as sns

sns.set_style('whitegrid')
```

```
sns.set_palette('Dark2')
sns.pairplot(df, height=3, plot_kws={'s': 10})
```

Out[5]: <seaborn.axisgrid.PairGrid at 0x75d26eccb010>



I made custom patters to make the `data` more `interesting` to see if the `Neural Network` can `learn` the `patterns` in the `data`.

Training and Testing Data

I suppose that I you guys know how to `split` the `data` into `training` and `testing` `data`.

If you don't you should go see my other articles and also visit my youtube channel.

In [6]:
```
from sklearn.model_selection import train_test_split
```

In [7]:
```
X = df.drop(columns='price')
y = df['price']

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

I'm doing a `80% training` and `20% testing` split.

In [8]:
```python
X_train.shape
```

Out[8]:
```
(800, 2)
```

In [9]:
```python
X_test.shape
```

Out[9]:
```
(200, 2)
```

> If you guys have large data. Than you should scale it before splitting it into `training` and `testing` `data` because the `Neural Network` is `sensitive` to the `scale` of the `data`.

But for this dataset the features values range is not that large.

In [10]:
```python
X.describe()
```

Out[10]:

	feature_1	feature_2
count	1000.000000	1000.000000
mean	5.009666	5.035418
std	2.949330	2.920090
min	-0.826510	-0.813661
25%	2.564901	2.531286
50%	4.932779	5.030279
75%	7.495682	7.538554
max	10.868813	11.148454

Building the Neural Network

So, let's build the `Neural Network`.

First, we need to define the layers.

Let's say:

- we want to make a neural network with `3` hidden layers.
  - 1st hidden layer has `4` `neurons`.
  - 2nd hidden layer has `6` `neurons`.
  - 3rd hidden layer has `4` `neurons`.
- Also we need to define the `activation function` for the `hidden layers`. Let's just say that the `activation function` for the `hidden layers` is `relu`.
- We want to predict the `price` so the `output layer` will have `1` `neuron` and it doen't need any activation function.

So, how do we do this?

First, we need to `import` the `layers` from `tensorflow`.

> As this is a fully connected `Neural Network`, we need to `import` the `Sequential` `model` from `tensorflow`.

In [11]:
```
from keras.models import Sequential
# from tensorflow.python.keras.models import Sequential
```

To define the layers we need to import another `class` from `tensorflow`, the `Dense` `class`.

In [12]:
```
from keras.layers import Dense
# from tensorflow.python.keras.layers import Dense
```

Now we can start making the `Neural Network`.

First we `initialize` the `model` using the `Sequential` `class`.

Adn inside the sequential `model` we `add` the `layers` using the `add` `method`.

> We can also directly pass a list of `layers` to the `model` using the `Sequential` `class`.

In [13]:
```
model = Sequential()
model.add(Dense(4, activation='relu')) # hidden layer with 4 perceptrons
model.add(Dense(6, activation='relu')) # hidden layer with 6 perceptrons
model.add(Dense(4, activation='relu')) # hidden layer with 4 perceptrons
model.add(Dense(1)) # output layer
```

Now, we have a `fully` connected `deep` `Neural Network`.

But this can be a little redundant.

So, we can use the second way to make a neural network.

In [14]:
```
model : Sequential = Sequential([
    Dense(4, activation='relu'),
    Dense(6, activation='relu'),
    Dense(4, activation='relu'),
    Dense(1)
])
```

And we are done... This process looks easier and cleaner.

So, Now for the next step, We have to `define` how the model will learn.

We have to `define` the `loss function` that will calculate the model's `performance` on the `training` `data`.

We need to `define` the `optimizer` that will be used to `update` the `weights` and `biases` of the `Neural Network`.

So, let's define the `loss function` and the `optimizer` now.

> We can use a method from the sequential `model` to `define` the `loss function` and the `optimizer`.

In [15]:
```
model.compile(loss='mse', optimizer='adam')
```

Inside the `compile` method, we can pass the `loss function` and the `optimizer` as arguments.

> I passed the `loss function` as `'mse'` which means `mean squared error` and the `optimizer` as `'adam'` which means `adam optimizer`.

> You can pass other `loss functions` like `'mae'`, `'binary_crossentropy'` and `'categorical_crossentropy'` and other `optimizers` like `'sgd'`, `'rmsprop'` and `'adam'`.

And our model is ready to `train`.

Just like other ml algorithms, we can use the `fit` method to `train` the `Neural Network`.

But The training process is a little different.

> We need to pass the `training` data to the `fit` method.

> Then we have to `define` the number of `epochs`. What is an `epoch`? It is the number of times the `Neural Network` will `see` the `entire training data` to `train` the `Neural Network` and go through the `training process`.

By default the `Neural Network` will `see` the entire training data `1` time to `train` the `Neural Network`.

But that is not a good idea. We can pass 100 or 1000 or 10000 `epochs` to the `fit` method to `train` the `Neural Network` and go through the `training process`.

But in this case 400. And I'll also set the `verbose` parameter to `1` to see the `training progress` of the `Neural Network` during the `training process`.

In [16]:
```
model.fit(x=X_train, y=y_train, epochs=400, verbose=1) # verbose=0, 1, 2,
```

```
Epoch 1/400
```

```
2026-02-04 17:24:09.864579: I external/local_xla/xla/service/service.cc:16
3] XLA service 0x75d160017400 initialized for platform CUDA (this does not
guarantee that XLA will be used). Devices:
2026-02-04 17:24:09.864595: I external/local_xla/xla/service/service.cc:17
1]   StreamExecutor device (0): NVIDIA GeForce RTX 3060, Compute Capability
8.6
2026-02-04 17:24:09.881407: I tensorflow/compiler/mlir/tensorflow/utils/dum
p_mlir_util.cc:269] disabling MLIR crash reproducer, set env var `MLIR_CRAS
H_REPRODUCER_DIRECTORY` to enable.
2026-02-04 17:24:09.981320: I external/local_xla/xla/stream_executor/cuda/c
uda_dnn.cc:473] Loaded cuDNN version 91801
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 4365114880.0000
Epoch 2/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 4365071360.0000
Epoch 3/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 4365027840.0000
Epoch 4/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 4364979200.0000
Epoch 5/400
 1/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 4465872384.0000
I0000 00:00:1770204250.704504  501860 device_compiler.h:196] Compiled clust
er using XLA!  This line is logged at most once for the lifetime of the pro
cess.
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 4364938752.0000
Epoch 6/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 4364899328.0000
Epoch 7/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 4364850176.0000
Epoch 8/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 4364763648.0000
Epoch 9/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 4364537344.0000

...

25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5594172.0000
Epoch 395/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 5592804.0000
Epoch 396/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5572678.5000
Epoch 397/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5566153.5000
Epoch 398/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5561696.5000
Epoch 399/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5553607.5000
Epoch 400/400
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5552910.0000
```

```
Out[16]: <keras.src.callbacks.history.History at 0x75d1c87cf150>
```

Now, we have a `trained` neural network. And we can `evaluate` the model with the `testing` set. But before that we should varify that the `Neural Network` is `learning` from the `data` in every `epoch` and `minimizing` the `loss value` to make `accurate predictions`.

How do we see that.

Every neural network has a `history attribute` that stores the `loss values` at each `epoch`.

So, we can plot the `loss` values at each `epoch` to `visualize` the training progress of the `Neural Network`.

In [17]:
```
model.history.history
```

Out[17]:
```
{'loss': [4365114880.0,
  4365071360.0,
  4365027840.0,
  4364979200.0,
  4364938752.0,
  4364899328.0,
  4364850176.0,

  ...

  5690763.5,
  5691020.0,
  5670999.0,
  5664703.5,
  5678179.0,
  5652080.0,
  5630784.0,
  5617936.0,
  5616102.0,
  5595054.5,
  5594172.0,
  5592804.0,
  5572678.5,
  5566153.5,
  5561696.5,
  5553607.5,
  5552910.0]}
```

Let's plot. I'll transform it to a `dataframe` first and use pandas built-in `plot` method.

In [18]:
```
loss_df = pd.DataFrame(model.history.history)
```

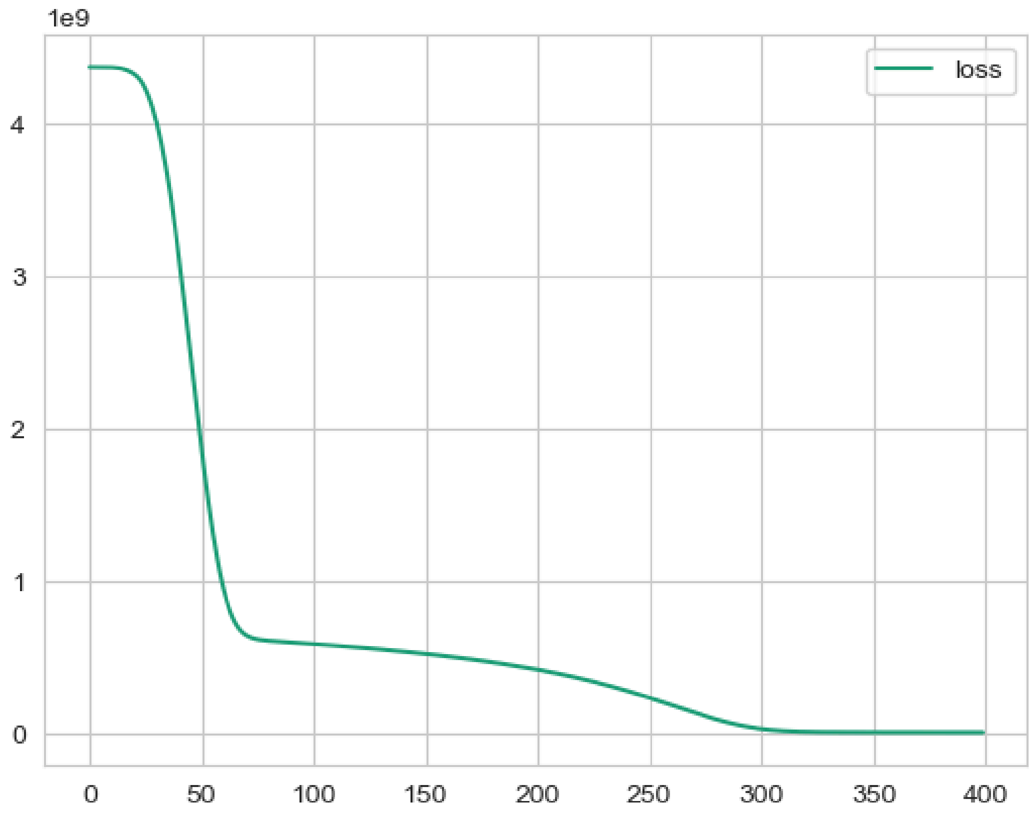
In [19]:
```
loss_df.head()
```

Out[19]:

	loss
0	4.365115e+09
1	4.365071e+09
2	4.365028e+09
3	4.364979e+09
4	4.364939e+09

In [20]:
```
loss_df.plot()
```

Out[20]:
```
<Axes: >
```



This is good. As you can see the loss started to decrease by the 20th epoch and has a steep decline after that and after that it was declining smoothly.

And that raises a question in my mind. Can the `loss` `value` decrease more. We ran this for 400 epochs. Is it possible?

Thanks to the `tensorflow` we can continue from where we left.

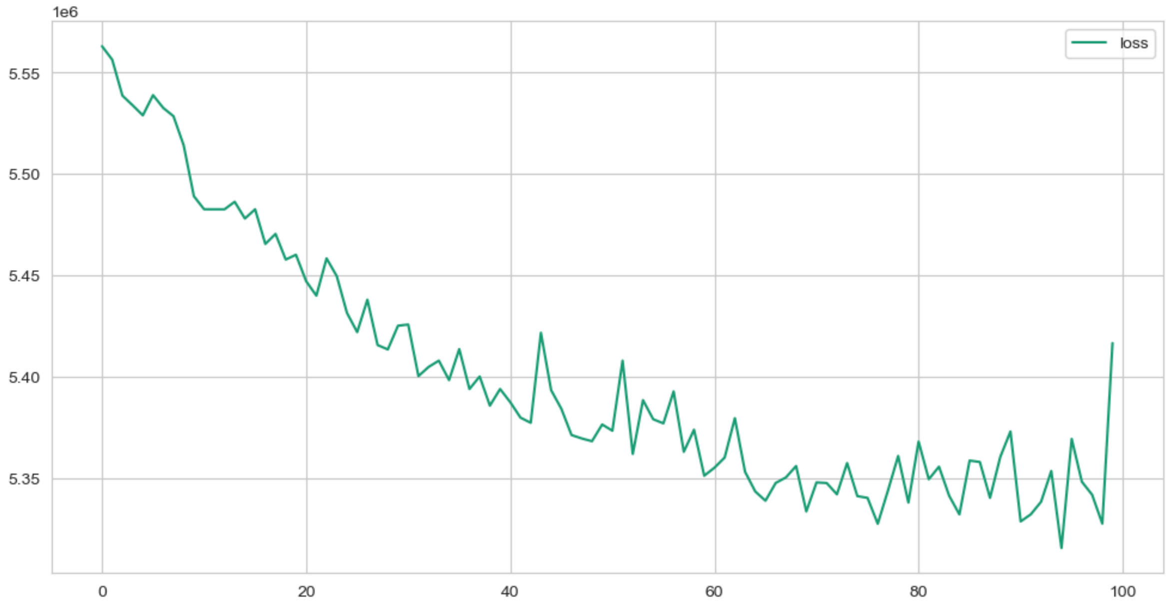
In [21]:
```
model.fit(
    X_train,
    y_train,
    epochs=100,
    verbose=1
)
```

```
Epoch 1/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5562906.0000
Epoch 2/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5556253.5000
Epoch 3/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5538517.0000
Epoch 4/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5533753.5000
Epoch 5/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5528807.5000
Epoch 6/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5538772.0000

...

Epoch 99/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5327429.0000
Epoch 100/100
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 5416493.0000
```

Out[21]:
```
<keras.src.callbacks.history.History at 0x75d1905ad010>
```

In [22]:
```python
pd.DataFrame(model.history.history).plot(
    y='loss',
    figsize=(12, 6)
)
```

Out[22]:
```
<Axes: >
```



I trained the model 100 more times and the loss value is not decreasing that much and the loss is oscillating.

So, I think 400 epochs is enough.

> Sometimes, I just choos a epoch and see the loss graph to find the point where the loss value stops decreasing and retrain it to that epoch for a better model.

> Training a model for longer than needed can cause it to overfit and ruin it's pogress.

Now, that we have a model ready let's evaluate.

Evaluating the Neural Network

There are a lot of methods to evaluate the `performance` of the `Neural Network`. But the most common method is to `calculate` the `loss` `value` of the `Neural Network` on the `testing` `data`.

for loss of the testing data we can just use the `evaluate` `method`.

In [23]:
```python
model.evaluate(X_test, y_test, verbose=0)
```

Out[23]:
```
5999444.0
```

This will return the `mean squared error` of the `Neural Network` on the `testing` `data`.

Let's see if it's close to the `training` `loss` `value`.

In [24]:
```
model.evaluate(X_train, y_train, verbose=0)
```

Out[24]:
```
5336327.0
```

Pretty close. This means the model didn't overfit that much and it's performing well on the `testing` `data`.

So, let's see how it's actually performing by `predicting` the `prices` of the `testing` `data` and finding the `mean absolute error` and the `root mean squared error` of the `Neural Network` on the `testing` `data`.

In [25]:
```
test_predictions = model.predict(X_test)
```

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
```

In [26]:
```
test_predictions[:10]
```

Out[26]:
```
array([[67331.5  ],
       [56194.598],
       [70025.57 ],
       [70047.09 ],
       [54981.914],
       [65676.984],
       [73783.88 ],
       [59229.37 ],
       [59358.246],
       [61401.32 ]], dtype=float32)
```

In [28]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error, root_

print(
    f'Mean Absolute Error: {mean_absolute_error(y_test, test_predictions)}
)

print(
    f'Root Mean Squared Error: {root_mean_squared_error(y_test, test_predi
)
```

```
Mean Absolute Error: 2068.42349639483

Root Mean Squared Error: 2449.3759136940803
```

Hey that's not bad.

if we look at the description of the `dataframe`.

In [29]:
```
df.describe()
```

Out[29]:

	feature_1	feature_2	price
count	1000.000000	1000.000000	1000.000000
mean	5.009666	5.035418	65829.736467
std	2.949330	2.920090	6705.261420
min	-0.826510	-0.813661	50000.000000
25%	2.564901	2.531286	60710.004607
50%	4.932779	5.030279	63626.750195
75%	7.495682	7.538554	72866.770999
max	10.868813	11.148454	80000.000000

The minimum and maximum price is `50000` and `80000` respectively.

Our model is predicting prices between `50000` and `80000` with a `mean absolute error` of `2068` and `rmse` of `2449` which is a `pretty good` in the context of the `data`.

We are working with a very very small neural model.

Before going to the next thing I want to compare it's results with a traditional machine learning model. In this case `Random Forest would be good right?

Let's do that...

In [34]:

```
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor()
rf.fit(X_train, y_train)
```

Out[34]:

```
▾ RandomForestRegressor
▸ Parameters
```

In [35]:

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, root_

print(
    f'Mean Absolute Error: {mean_absolute_error(y_test, rf.predict(X_test)
)

print(
    f'Root Mean Squared Error: {root_mean_squared_error(y_test, rf.predict
)
```

```
Mean Absolute Error: 1311.5986482382473

Root Mean Squared Error: 1665.043243460561
```

WHAAAAAT!! HOW CAAAAN THIIIIS BEEE??!!! A MEASLY PROBABILISTIC MODEL OUT PERFORMING THE SUPERIOR NEURAL NETWORK. HOOOOWWWWWW!

It is what it is...

Try to find out what could be wrong. Try to tweak some stuff around like the activation functions or the learning rate.

See if you can beat the `random forest` model.

Saving and Loading Models

Now that we have a trained `Neural Network` we can `save` it using the `save_model` method of the `tensorflow.keras` `module` and then we can `load` the `model` using the `load_model` `function` from the `keras.models` `module` of the `tensorflow` `library`.

In [39]:
```
from keras.saving import save_model

save_model(model, 'first_model.keras')
```

Now you can load the `model` using the `load_model` `function` from the `keras.models` module.

In [43]:
```
from keras.models import load_model
```

In [44]:
```
model = load_model('./first_model.keras', compile=False)
```

In [47]:
```
model.predict(X_test)
```

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
```

Out[47]:
```
array([[67331.5  ],
       [56194.598],
       [70025.57 ],
       [70047.09 ],
       [54981.914],
       [65676.984],
       [73783.88 ],

       ...

       [59229.37 ],
       [59358.246],
       [72693.89 ],
       [59538.805],
       [67625.53 ],
       [64433.31 ],
       [76328.21 ]], dtype=float32)
```

AAAAANNNND, Voila. We have successfully built and `trained` Our fist `Neural Network` in Python using the `TensorFlow` and `Keras` libraries.

Final Words

Good Luck! That's all I can say.


GOOD LUCK