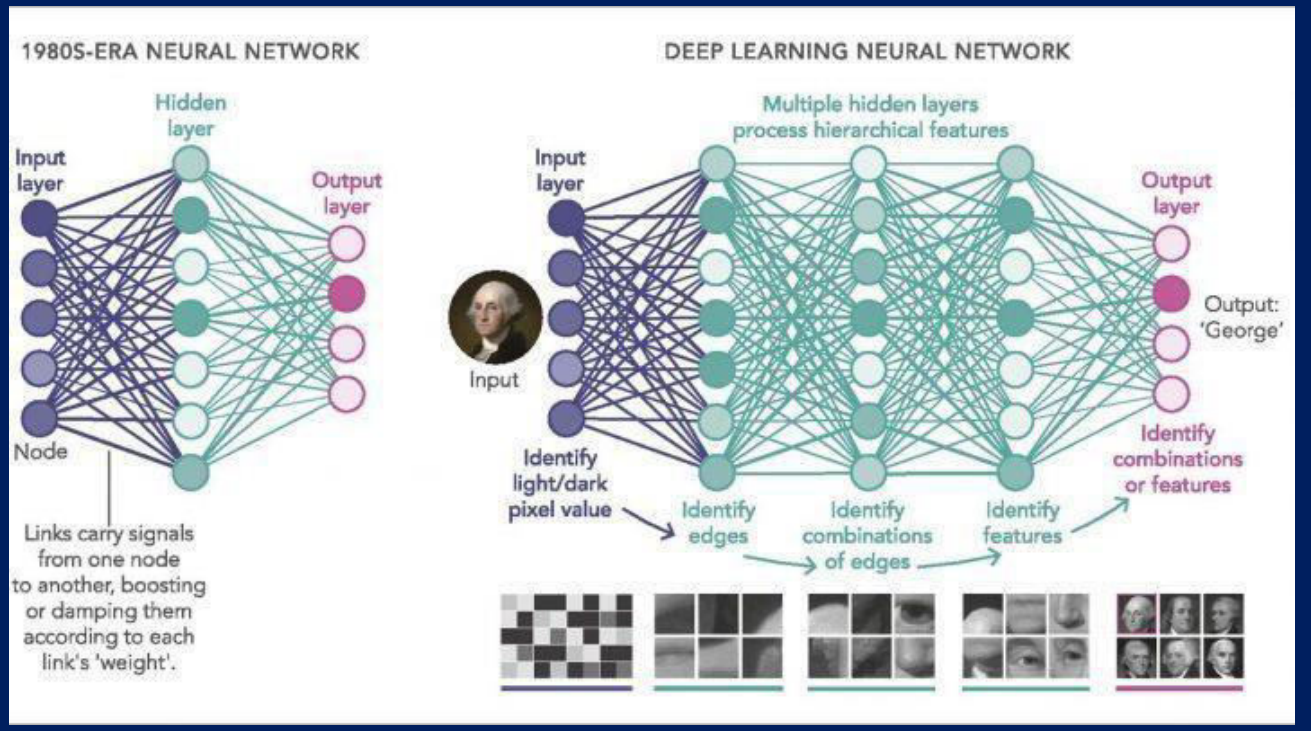
CSE4006- Deep learning

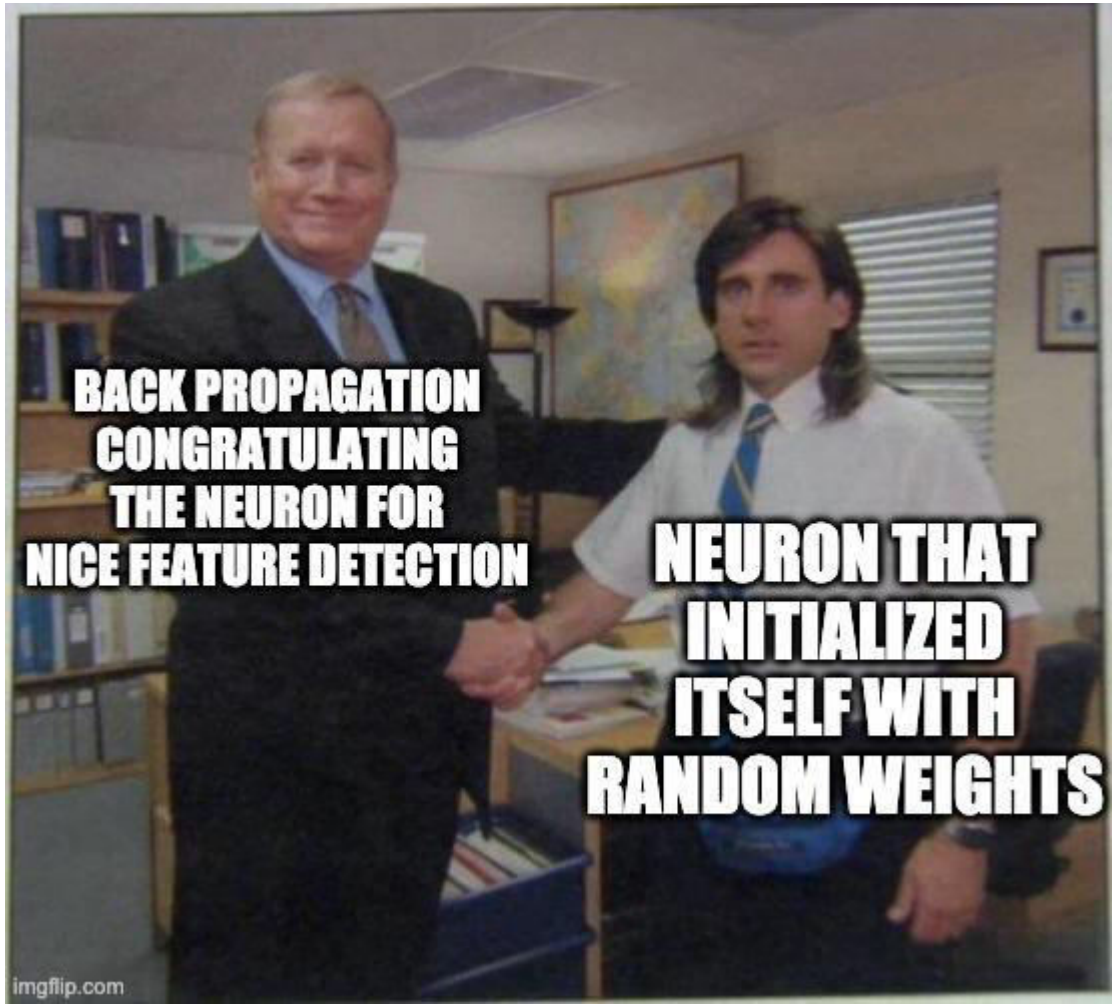
Dr. D.Sumathi



Deep Neural Networks

Multilayer Perceptron-Gradient based Learning- **Backpropagation Algorithm**- Regularization for Deep Learning- Optimization for training deep models


MLP STANDS FOR...
MY LITTLE PONY
MULTILAYER PERCEPTRON
SEE MORE UNICORNS QUARRELLING, @TDDCOMICS ON

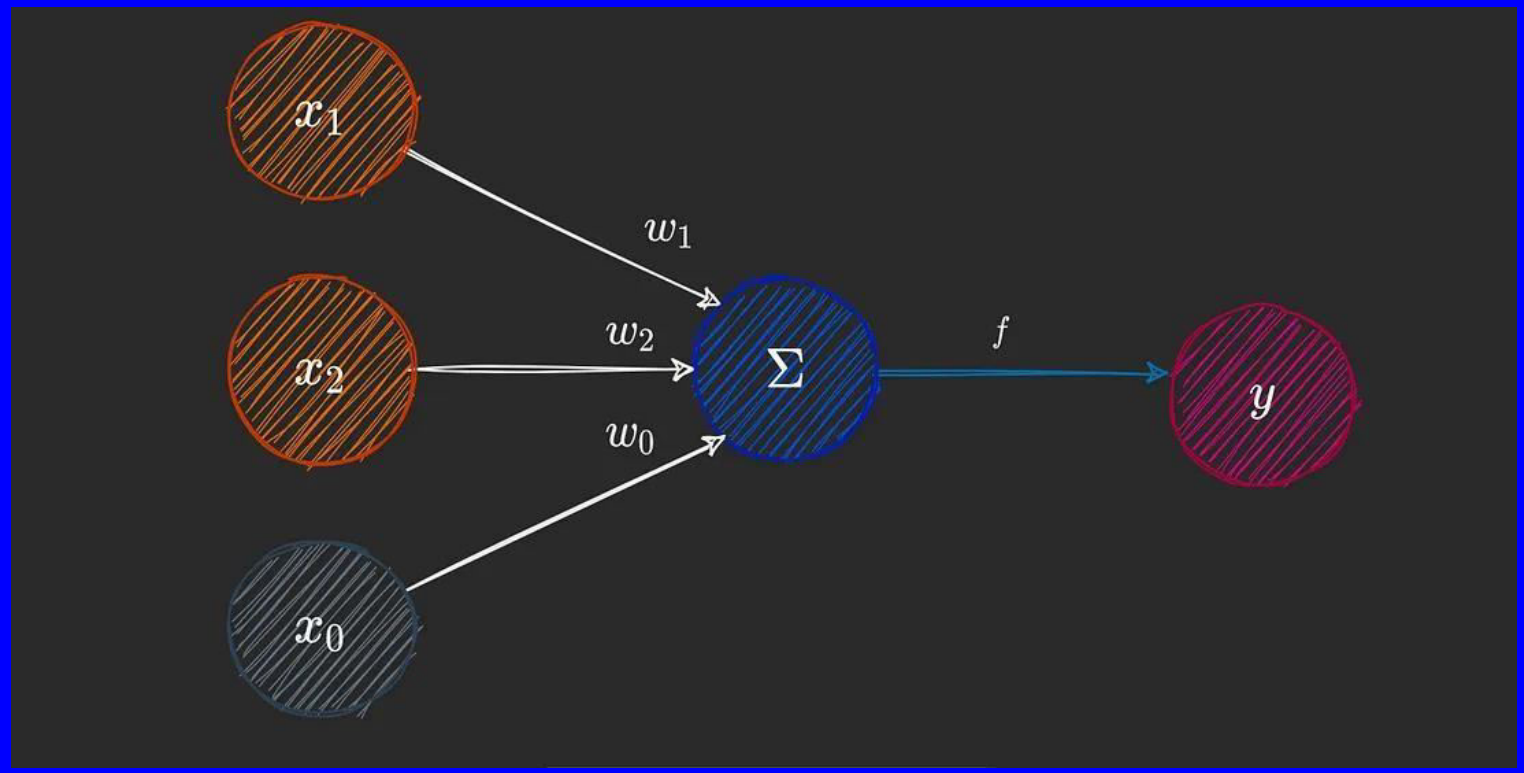

BACK PROPAGATION CONGRATULATING THE NEURON FOR NICE FEATURE DETECTION
NEURON THAT INITIALIZED ITSELF WITH RANDOM WEIGHTS
imgflip.com

RECAP

- **The perceptron** is a classification algorithm. Specifically, it works as a linear binary classifier. It was invented in the late 1950s by Frank Rosenblatt.
- The perceptron basically works as a threshold function — non-negative outputs are put into one class while negative ones are put into the other class.

Perceptron –Components

- Input nodes
- Output node
- An activation function
- Weights and biases
- Error function

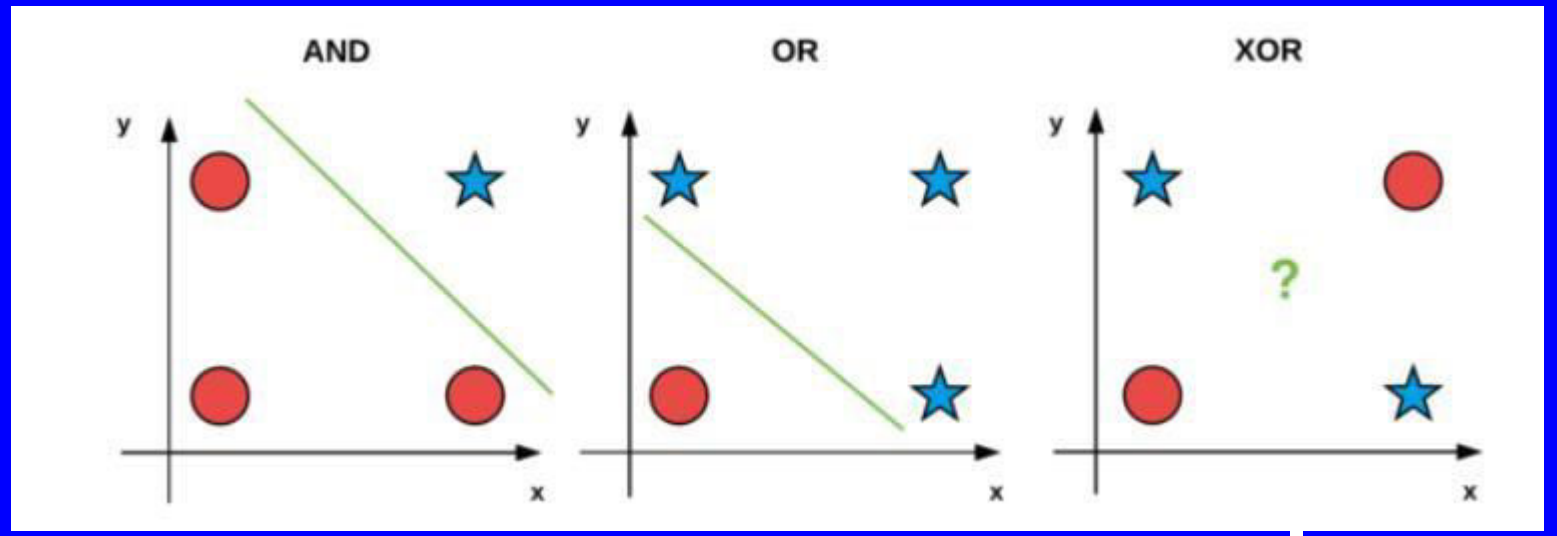


Classification

$$Class_1 : w \cdot X + b \geq 0$$

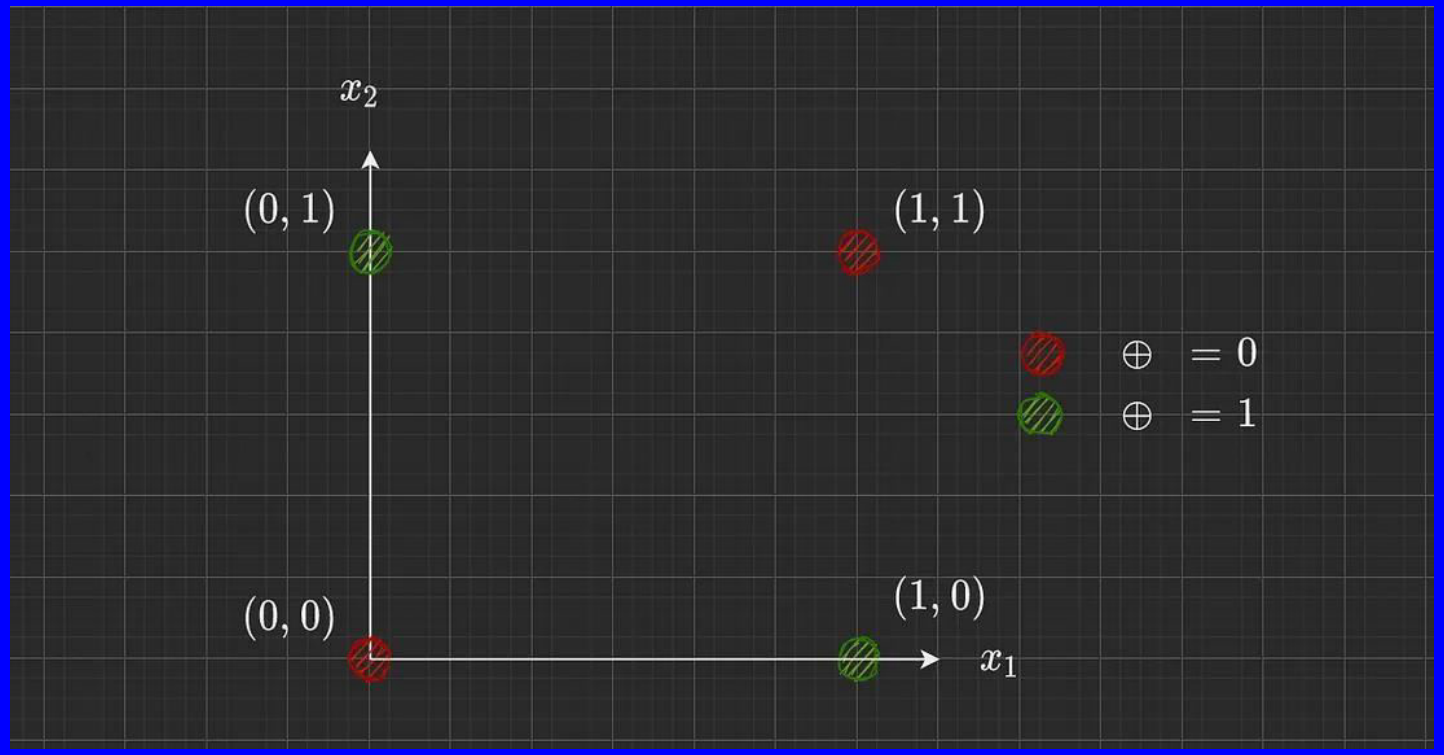
$$Class_2 : w \cdot X + b < 0$$

Implement linear classification in terms of AND,OR gates..



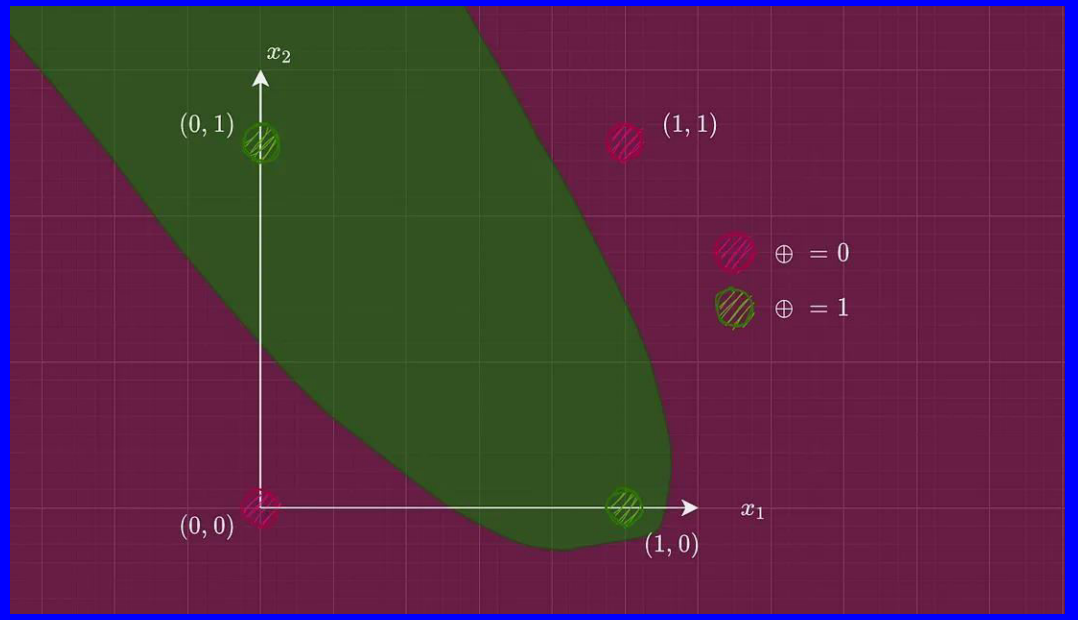
Model to mimic XoR problem

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



Attempt #1: The Single Layer Perceptron

- A perceptron can only converge on linearly separable data. Therefore, it isn't capable of imitating the XOR function.
- a perceptron must correctly classify the entire training data in one go.
- Non-linearity allows for more complex decision boundaries. One potential decision boundary for our XOR data could look like this.

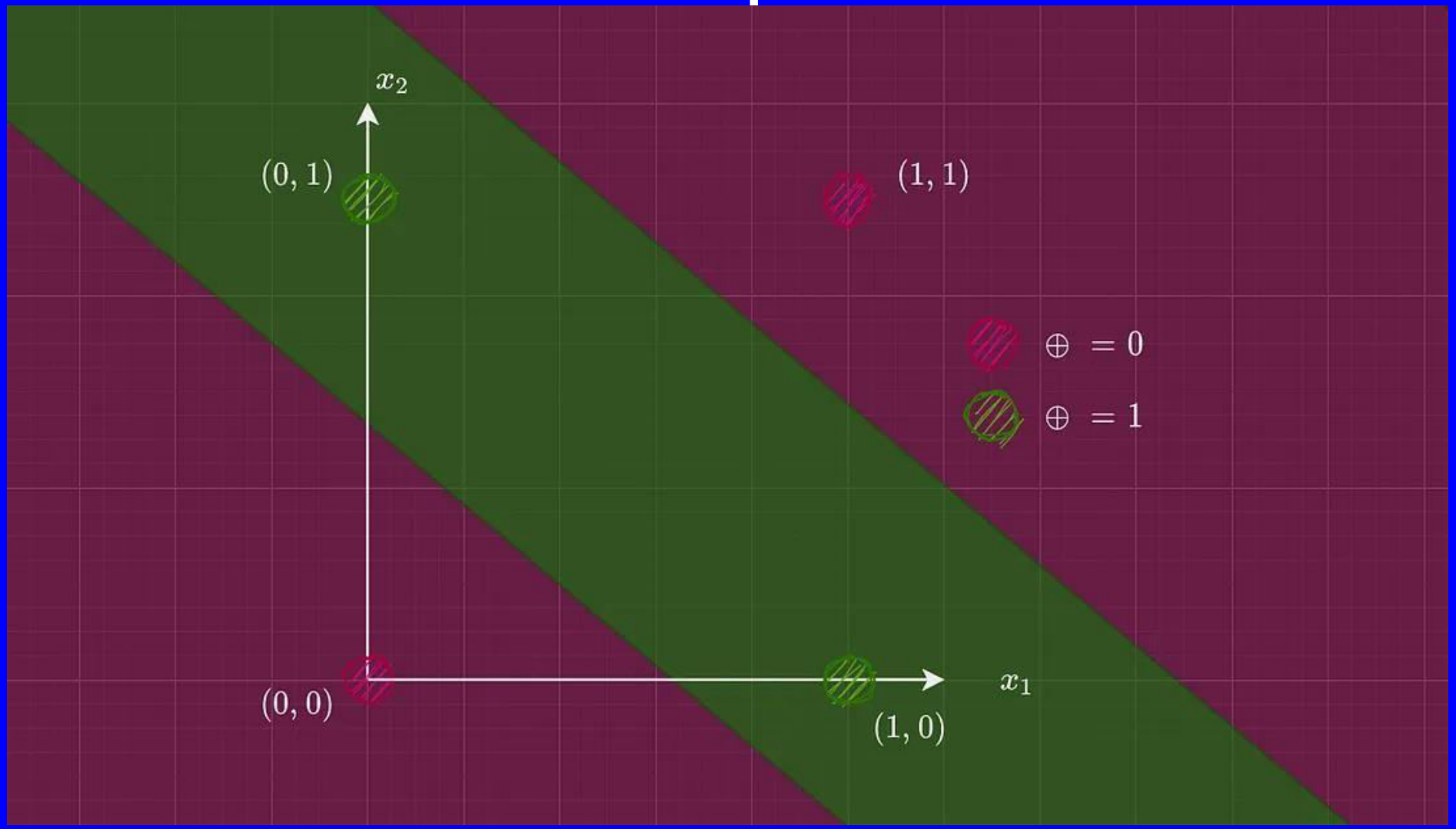


The 2d XOR problem — Attempt #2

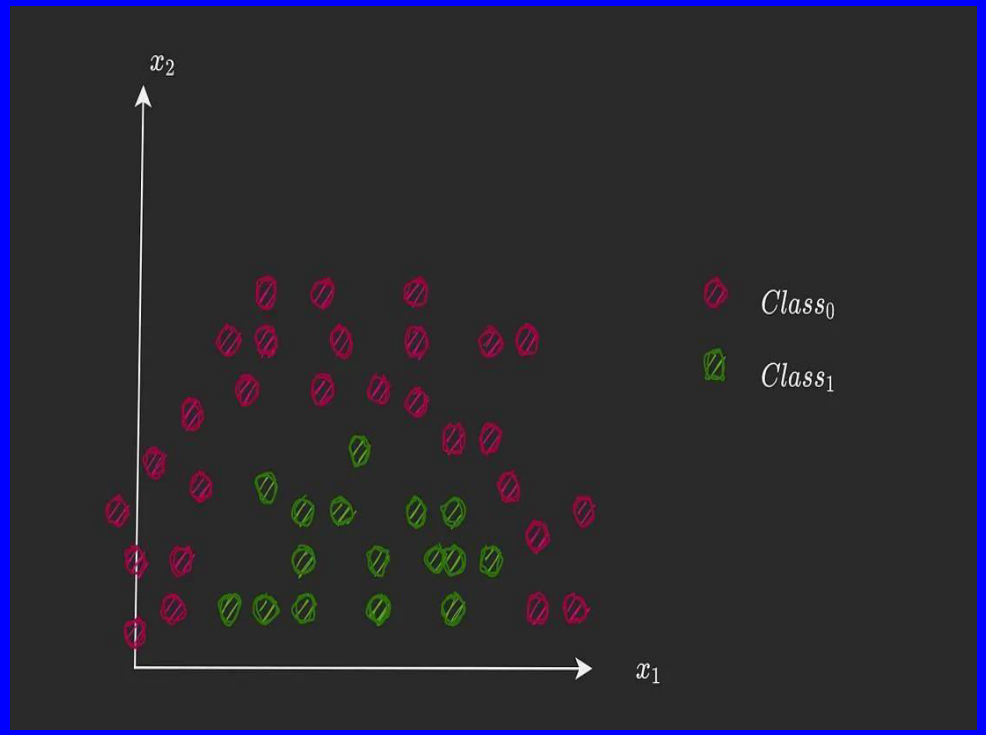
- Imitating the XOR function would require a non-linear decision boundary.
- The XOR problem with neural networks can be solved by using Multi-Layer Perceptrons or a neural network architecture with an input layer, hidden layer, and output layer.
- So during the forward propagation through the neural networks, the weights get updated to the corresponding layers and the XOR logic gets executed.

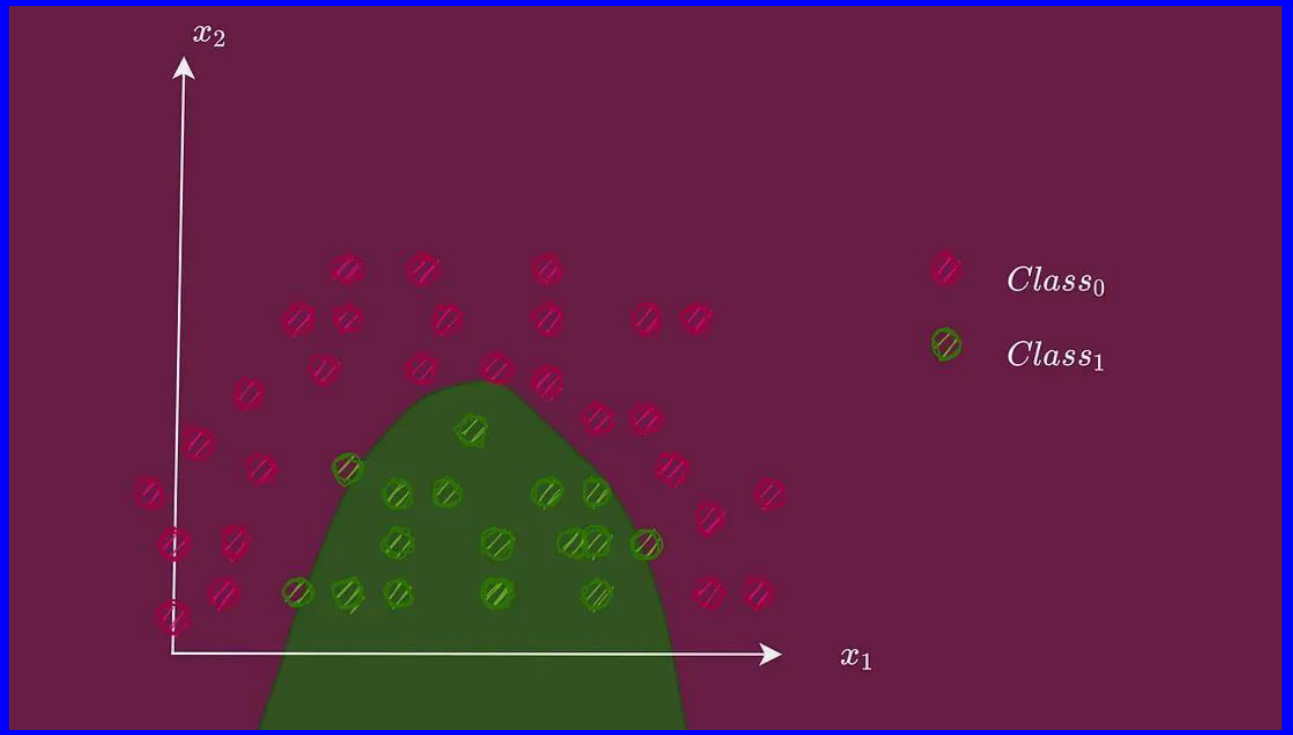
Perceptron to solve XOR problem

- Out of all the 2 input logic gates, the XOR and XNOR gates are the only ones that are not linearly-separable



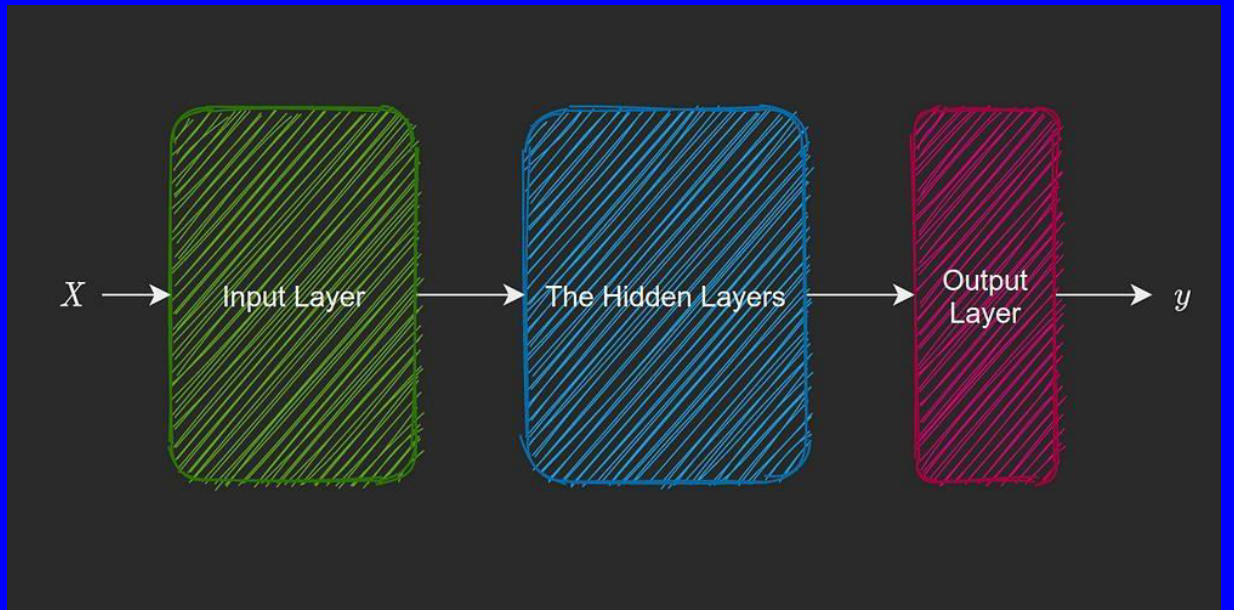
potential decision boundary could be something to classify????





The Multi-layered Perceptron

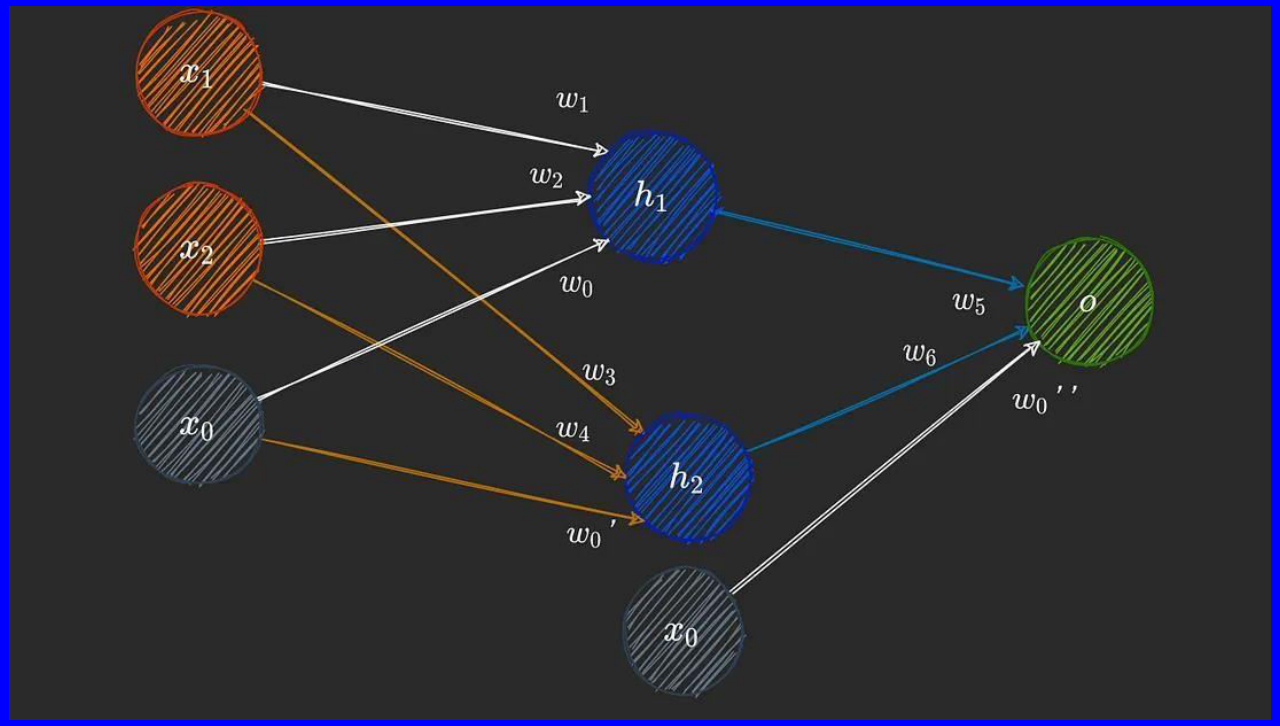
- Components are
- input and output nodes,
- activation function
- weights and
- Biases
- An MLP can have hidden layers. An MLP is generally restricted to having a minimum of single hidden layer



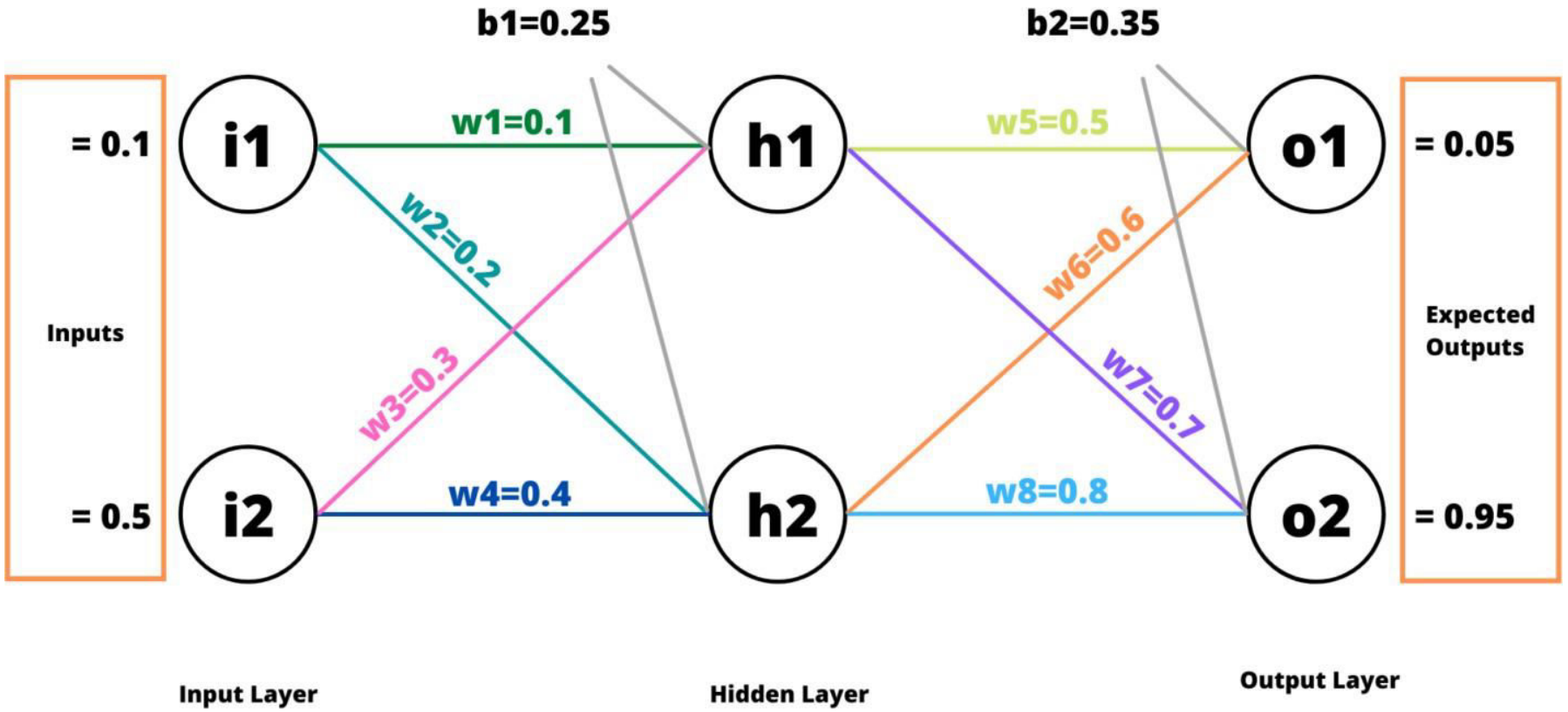
- Activation functions should be differentiable, so that a network's parameters can be updated using backpropagation.
- Though the output generation process is a direct extension of that of the perceptron, updating weights isn't so straightforward.
- **Backpropagation** is an algorithm for update the weights and biases of a model based on their gradients with respect to the error function, starting from the output layer all the way to the first layer.

Architecture-MLP

- The architecture of a network refers to its general structure — the number of hidden layers, the number of nodes in each layer and how these nodes are inter-connected.



Example-1



$$sum_{h1} = i_1 * w_1 + i_2 * w_3 + b_1$$

$$sum_{h1} = 0.1 * 0.1 + 0.5 * 0.3 + 0.25 = 0.41$$

$$output_{h1} = \frac{1}{1 + e^{-sum_{h1}}}$$

$$output_{h1} = \frac{1}{1 + e^{-0.41}} = 0.60108$$

Similarly for h2, we perform the weighted sum operation sum_{h2} and compute the activation value $output_{h2}$.

$$sum_{h2} = i_1 * w_2 + i_2 * w_4 + b_1 = 0.47$$

$$output_{h2} = \frac{1}{1 + e^{-sum_{h2}}} = 0.61538$$

Now, $output_{h1}$ and $output_{h2}$ will be considered as inputs to the next layer.

For o1,

$$sum_{o1} = output_{h1} * w_5 + output_{h2} * w_6 + b_2 = 1.01977$$

$$output_{o1} = \frac{1}{1 + e^{-sum_{o1}}} = 0.73492$$

Similarly for o2,

$$sum_{o2} = output_{h1} * w_7 + output_{h2} * w_8 + b_2 = 1.26306$$

$$output_{o2} = \frac{1}{1 + e^{-sum_{o2}}} = 0.77955$$

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

To compute E_{total}, we need to first find out respective errors at $o1$ and $o2$.

$$E_1 = \frac{1}{2}(target_1 - output_{o1})^2$$

$$E_1 = \frac{1}{2}(0.05 - 0.73492)^2 = 0.23456$$

Similarly for E2,

$$E_2 = \frac{1}{2}(target_2 - output_{o2})^2$$

$$E_2 = \frac{1}{2}(0.95 - 0.77955)^2 = 0.01452$$

Therefore,

$$E_{total} = E_1 + E_2 = 0.24908$$

$$\frac{\partial E_{total}}{\partial w5} = \frac{\partial E_{total}}{\partial output_{o1}} * \frac{\partial output_{o1}}{\partial sum_{o1}} * \frac{\partial sum_{o1}}{\partial w5}$$

$$\frac{\partial E_{total}}{\partial w5} = [output_{o1} - target_1] * [output_{o1}(1 - output_{o1})] * [output_{h1}]$$

$$\frac{\partial E_{total}}{\partial w5} = 0.68492 * 0.19480 * 0.60108$$

$$\frac{\partial E_{total}}{\partial w5} = 0.08020$$

The new_w_5 is,

$new\_w_5 = w5 - n * \frac{\partial E_{total}}{\partial w5}$, where n is learning rate.

$$new\_w_5 = 0.5 - 0.6 * 0.08020$$

$$new\_w_5 = 0.45187$$

$$\frac{\partial E_{total}}{\partial w6} = \frac{\partial E_{total}}{\partial output_{o1}} * \frac{\partial output_{o1}}{\partial sum_{o1}} * \frac{\partial sum_{o1}}{\partial w6}$$

The first two components of this chain have already been calculated. The last component $\frac{\partial sum_{o1}}{\partial w6} = output_{h2}$.

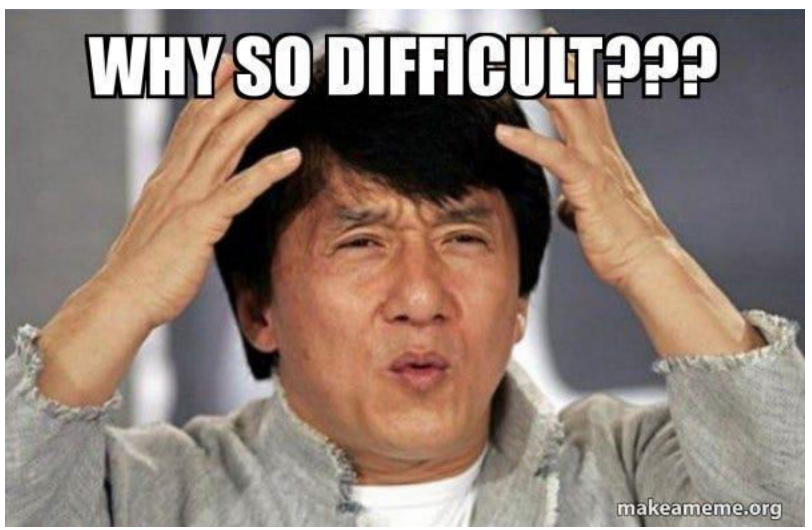
$$\frac{\partial E_{total}}{\partial w6} = 0.68492 * 0.19480 * 0.61538 = 0.08211$$

The new_w_6 is,

$$new\_w_6 = w6 - n * \frac{\partial E_{total}}{\partial w6}$$

$$new\_w_6 = 0.6 - 0.6 * 0.08211$$

$$new\_w_6 = 0.55073$$



$new\_w_7 = 0.71056$

Proceeding similarly, we get $new\_w_8 = 0.81081$ (with $\frac{\partial E_{total}}{\partial w8} = -0.01802$).

For w1 (with respect to E1),

$$\frac{\partial E_1}{\partial w1} = \frac{\partial E_1}{\partial output_{o1}} * \frac{\partial output_{o1}}{\partial sum_{o1}} * \frac{\partial sum_{o1}}{\partial output_{h1}} * \frac{\partial output_{h1}}{\partial sum_{h1}} * \frac{\partial sum_{h1}}{\partial w1}$$

For the third component,

$$sum_{o1} = output_{h1} * w_5 + output_{h2} * w_6 + b_2$$

$$\frac{\partial sum_{o1}}{\partial output_{h1}} = w5$$

For the fourth component,

$$\frac{\partial output_{h1}}{\partial sum_{h1}} = output_{h1} * (1 - output_{h1})$$

For the fifth component,

$$sum_{h1} = i_1 * w_1 + i_2 * w_3 + b_1$$

$$\frac{\partial sum_{h1}}{\partial w1} = i_1$$

Putting them all together,

$$\frac{\partial E_1}{\partial w1} = \frac{\partial E_1}{\partial output_{o1}} * \frac{\partial output_{o1}}{\partial sum_{o1}} * \frac{\partial sum_{o1}}{\partial output_{h1}} * \frac{\partial output_{h1}}{\partial sum_{h1}} * \frac{\partial sum_{h1}}{\partial w1}$$

$$\frac{\partial E_1}{\partial w1} = 0.68492 * 0.19480 * 0.5 * 0.23978 * 0.1 = 0.00159$$

Similarly, for w1 (with respect to E2),

$$\frac{\partial E_2}{\partial w1} = \frac{\partial E_2}{\partial output_{o2}} * \frac{\partial output_{o2}}{\partial sum_{o2}} * \frac{\partial sum_{o2}}{\partial output_{h1}} * \frac{\partial output_{h1}}{\partial sum_{h1}} * \frac{\partial sum_{h1}}{\partial w1}$$

For the first component of the above chain,

$$\frac{\partial E_2}{\partial output_{o2}} = output_{o2} - target_2$$

For the third component,

$$sum_{o2} = output_{h1} * w_7 + output_{h2} * w_8 + b_2$$

$$\frac{\partial sum_{o2}}{\partial output_{h1}} = w7$$

The fourth and fifth components have also been already computed while computing $\frac{\partial E_1}{\partial w1}$.

Putting them all together,

$$\frac{\partial E_2}{\partial w1} = \frac{\partial E_2}{\partial output_{o2}} * \frac{\partial output_{o2}}{\partial sum_{o2}} * \frac{\partial sum_{o2}}{\partial output_{h1}} * \frac{\partial output_{h1}}{\partial sum_{h1}} * \frac{\partial sum_{h1}}{\partial w1}$$

$$\frac{\partial E_2}{\partial w1} = -0.17044 * 0.17184 * 0.7 * 0.23978 * 0.1 = -0.00049$$

Now we can compute $\frac{\partial E_{total}}{\partial w1} = \frac{\partial E_1}{\partial w1} + \frac{\partial E_2}{\partial w1}$.

$$\frac{\partial E_{total}}{\partial w1} = 0.00159 + (-0.00049) = 0.00110.$$

The new_w_1 is,

$$new\_w_1 = w1 - n * \frac{\partial E_{total}}{\partial w1}$$

$$new\_w_1 = 0.1 - 0.6 * 0.00110$$

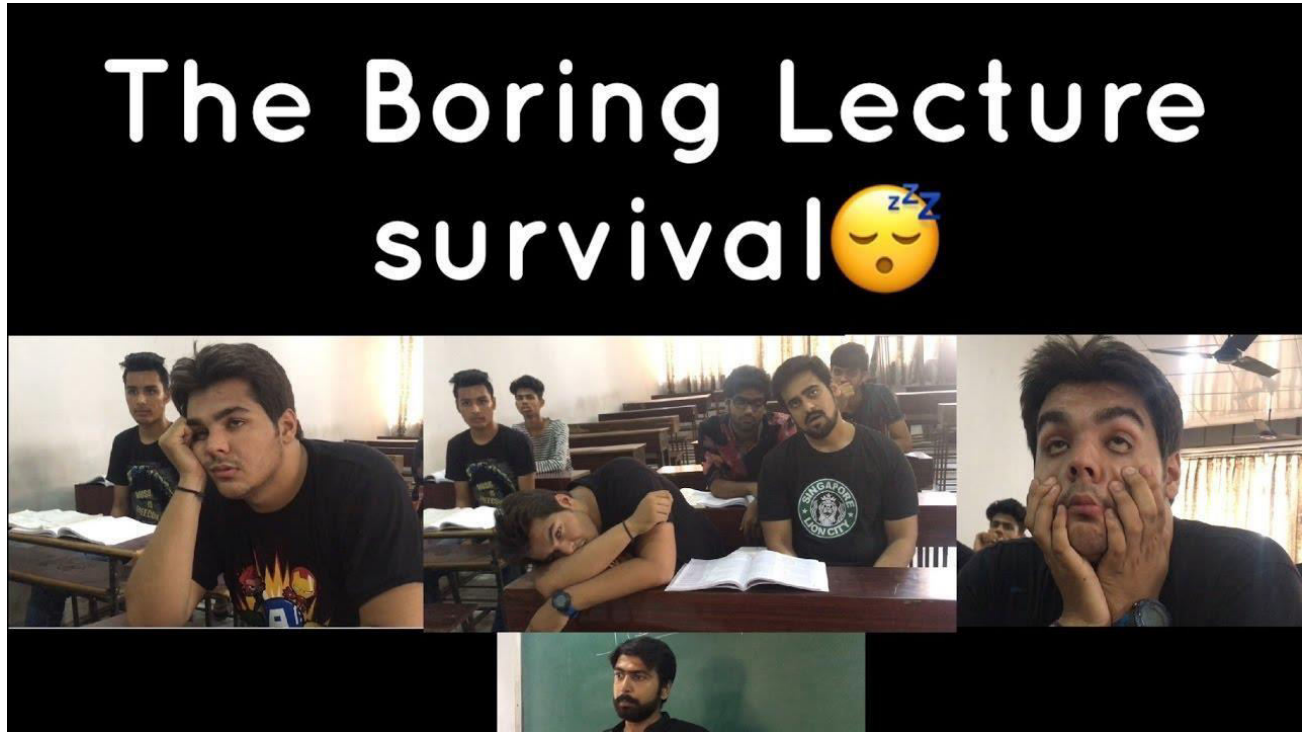
$$new\_w_1 = 0.09933$$

Proceeding similarly, we can easily update the other weights (w2, w3 and w4).

$$new\_w_2 = 0.19919$$

$$new\_w_3 = 0.29667$$

$$new\_w_4 = 0.39597$$



Process

- update all the old weights with these new weights.
- Once the weights are updated, one backpropagation cycle is finished.
- Now the forward pass is done and the total new error is computed.
- And based on this newly computed total error the weights are again updated.
- This goes on until the loss value converges to minima.
- This way a neural network starts with random values for its weights and finally converges to optimum values.

Key takeaway

- **#1** Adding more layers or nodes : It gives increasingly complex decision boundaries that could also lead to overfitting — where a model achieves very high accuracies on the training data, but fails to generalize.
- #2: Choosing a loss function: It makes some assumptions on the data (like it being gaussian) and isn't always convex when it comes to a classification problem

Using Perceptron for Sentiment Analysis

With the final labels assigned to the entire corpus, you decided to fit the data to a **Perceptron**, the simplest neural network of all.

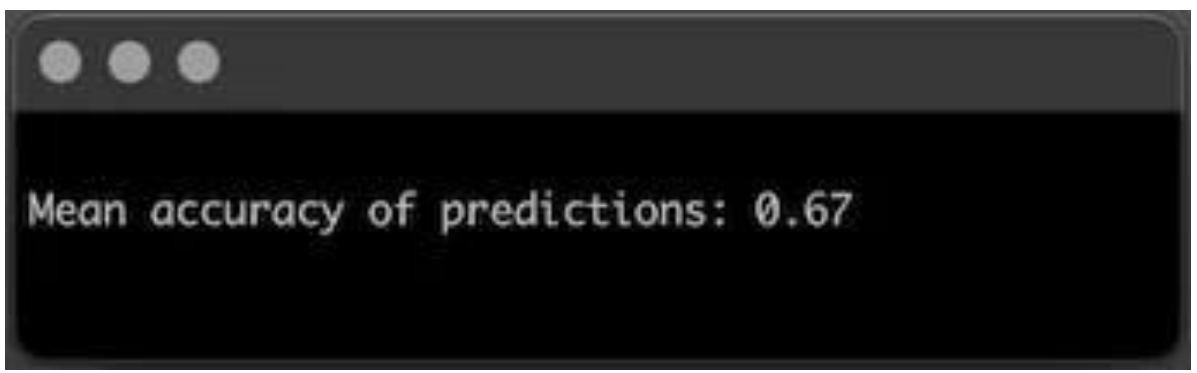
- Text from the guestbooks as a vector using the Term Frequence-Inverse Document Frequency(TF-IDF). This method encodes any kind of text as a statistic of how frequent each word, or term, is in each sentence and the entire document.
- In Python we need to use TfidfVectorizer method from ScikitLearn.
- Remove English stop-words and even applyL1 normalization.

```
TfidfVectorizer(stop_words='english', lowercase=True,
norm='l1')
```

Step 1: Corpus is initialized along with targets

```
1 perceptron.py
import numpy as np
from sklearn import metrics
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = [
    'We enjoyed our stay so much. The weather was not great, but everything else was perfect.',
    'Going to think twice before staying here again. The wifi was spotty and the rooms smaller than advertised',
    'The perfect place to relax and recharge.',
    'Never had such a relaxing vacation.',
    'The pictures were misleading, so I was expecting the common areas to be bigger. But the service was good.',
    'There were no clean linens when I got to my room and the breakfast options were not that many.',
    'Was expecting it to be a bit far from historical downtown, but it was almost impossible to drive through those narrow roads',
    'I thought that waking up with the chickens was fun, but I was wrong.',
    'Great place for a quick getaway from the city. Everyone is friendly and polite.',
    'Unfortunately it was raining during our stay, and there weren\'t many options for indoors activities. Everything was great, but there was literally no other oprionts besides being in the rain.',
    'The town festival was postponed, so the area was a complete ghost town. We were the only guests. Not the experience I was looking for.',
    'We had a lovely time. It\'s a fantastic place to go with the children, they loved all the animals.',
    'A little bit off the beaten track, but completely worth it. You can hear the birds sing in the morning and then you are greeted with the biggest, sincerest smiles from the owners. Loved it!',
    'It was good to be outside in the country, visiting old town. Everything was prepared to the upmost detail'
    'Staff was friendly. Going to come back for sure.',
    'They didn\'t have enough staff for the amount of guests. It took some time to get our breakfast and we had to wait 20 minutes to get more information about the old town.',
    'The pictures looked way different.',
    'Best weekend in the countryside I\'ve ever had.',
    'Terrible. Slow staff, slow town. Only good thing was being surrounded by nature.',
    'Not as clean as advertised. Found some cobwebs in the corner of the room.',
    'It was a peaceful getaway in the countryside.',
    'Everyone was nice. Had a good time.',
    'The kids loved running around in nature, we loved the old town. Definitely going back.',
    'Had worse experiences.',
    'Surprised this was much different than what was on the website.',
    'Not that mindblowing.'
]
# 0: negative sentiment.  1: positive sentiment
targets = [1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0]
```

```
# Splitting the dataset
train_features, test_features, train_targets, test_targets = train_test_split(corpus, targets, test_size=0.1,
                                                                              random_state=123)
# Turning the corpus into a tf-idf array
vectorizer = TfidfVectorizer(stop_words='english', lowercase=True, norm='l1')

train_features = vectorizer.fit_transform(train_features)
test_features = vectorizer.transform(test_features)

# Build the perceptron and fit the data
classifier = Perceptron(random_state=457)
classifier.fit(train_features, train_targets)

predictions = classifier.predict(test_features)
score = np.round(metrics.accuracy_score(test_targets, predictions), 2)

print("Mean accuracy of predictions: " + str(score))
```



How would MultiLayer Perceptron perform in this case?

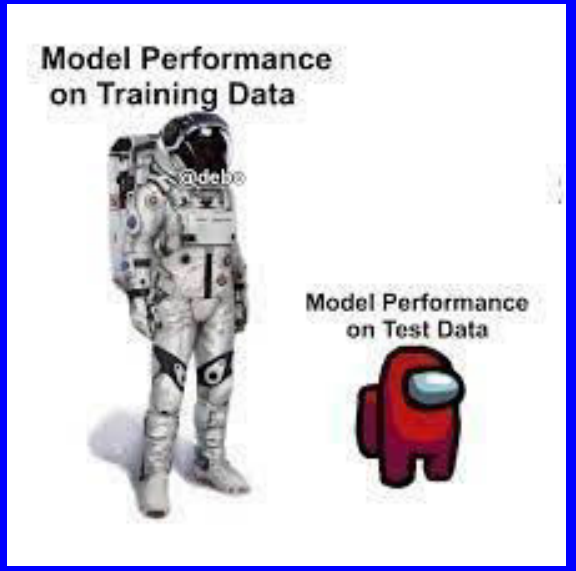
- Activation function: ReLU, specified with the parameter *activation='relu'*
- Optimization function: Stochastic Gradient Descent, specified with the parameter *solver='sgd'*
- Learning rate: Inverse Scaling, specified with the parameter *learning_rate='invscaling'*
- Number of iterations: 20, specified with the parameter *max_iter=20*

```
def buildMLPerceptron(train_features, test_features, train_targets, test_targets, num_neurons=2):
    """ Build a Multi-layer Perceptron and fit the data
        Activation Function: ReLU
        Optimization Function: SGD, Stochastic Gradient Descent
        Learning Rate: Inverse Scaling
    """

    classifier = MLPClassifier(hidden_layer_sizes=num_neurons, max_iter=35, activation='relu', solver='sgd', verbose=10, random_state=762, learning_rate='invscaling')
    classifier.fit(train_features, train_targets)

    predictions = classifier.predict(test_features)
    score = np.round(metrics.accuracy_score(test_targets, predictions), 2)
    print("Mean accuracy of predictions: " + str(score))

# Build Multi-Layer Perceptron with 3 hidden layers, each with 2 neurons
buildMLPerceptron(train_features, test_features, train_targets, test_targets)
```

- Finally, to see the value of the loss function at each iteration, you also added the parameter *verbose=True*.

```
Iteration 1, loss = 0.92036967
Iteration 2, loss = 0.92015402
Iteration 3, loss = 0.92001719
Iteration 4, loss = 0.91988325
Iteration 5, loss = 0.91975165
Iteration 6, loss = 0.91962290
Iteration 7, loss = 0.91949744
Iteration 8, loss = 0.91937559
Iteration 9, loss = 0.91925753
Iteration 10, loss = 0.91914334
Iteration 11, loss = 0.91903303
Iteration 12, loss = 0.91892654
Iteration 13, loss = 0.91882381
Iteration 14, loss = 0.91872473
Iteration 15, loss = 0.91862916
Iteration 16, loss = 0.91853698
Iteration 17, loss = 0.91844803
Iteration 18, loss = 0.91836219
Iteration 19, loss = 0.91827929
Iteration 20, loss = 0.91819919
Iteration 21, loss = 0.91812176
Iteration 22, loss = 0.91804686
Iteration 23, loss = 0.91797436
Iteration 24, loss = 0.91790412
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
Mean accuracy of predictions: 0.33
```

- What about if you added more *capacity* to the neural network? What happens when each hidden layer has more neurons to learn the patterns of the dataset?
- Simply change the *num_neurons* parameter an set it, for instance, to 5.
- buildMLPerceptron(train_features, test_features, train_targets, test_targets, num_neurons=5)
- Adding more neurons to the hidden layers definitely improved Model accuracy!



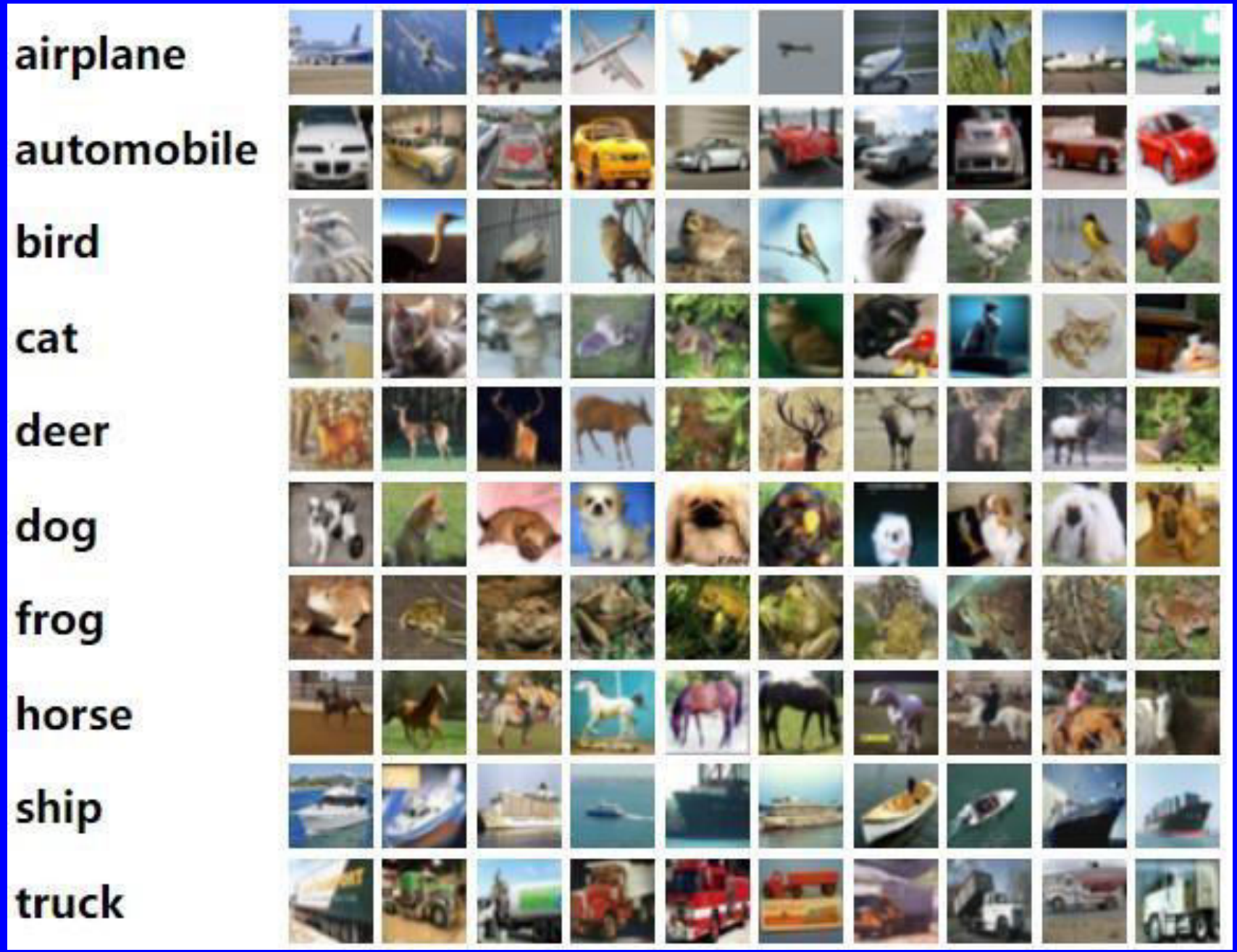
```
Iteration 1, loss = 0.76870847
Iteration 2, loss = 0.76854261
Iteration 3, loss = 0.76843742
Iteration 4, loss = 0.76833448
Iteration 5, loss = 0.76823336
Iteration 6, loss = 0.76813446
Iteration 7, loss = 0.76803812
Iteration 8, loss = 0.76794700
Iteration 9, loss = 0.76785898
Iteration 10, loss = 0.76777396
Iteration 11, loss = 0.76769193
Iteration 12, loss = 0.76761285
Iteration 13, loss = 0.76753663
Iteration 14, loss = 0.76746319
Iteration 15, loss = 0.76739244
Iteration 16, loss = 0.76732425
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
Mean accuracy of predictions: 0.67
```

Inferences

- Same neural network structure, 3 hidden layers, but with the increased computational power of the 5 neurons, the model got better at understanding the patterns in the data.
- It converged much faster and mean accuracy doubled!
- In the end, for this specific case and dataset, the Multilayer Perceptron performs as well as a simple Perceptron. But it was definitely a great exercise to see how changing the number of neurons in each hidden-layer impacts model performance.

Example -2 cifar dataset – deploy MLP

- Step 1: Preparing the Data for Training MLP Network
- Pixel scaling is an important preprocessing step that is often applied to the input data
- In image classification tasks, the pixel values in the images can range from 0 to 255 (for 8-bit images).
- Scaling the pixel values to be between 0 and 1 can make the model training process more stable and efficient. This can be done by dividing each pixel value by 255.


airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

- from tensorflow.keras.datasets import cifar10
- from tensorflow.keras.utils import to_categorical

- # Load the CIFAR-10 dataset
- (x_train, y_train), (x_test, y_test) = cifar10.load_data()

- # Scale the pixel values to between 0 and 1
- x_train = x_train / 255.0
- x_test = x_test / 255.0
- # Convert the labels to one-hot encoding makes it easy to compare the predicted probabilities to the true labels.
- y_train = to_categorical(y_train)
- y_test = to_categorical(y_test)

Defining the MLP Model Architecture

- Using sequential model
- Using the Functional API.
- In the next slide, let us see the Keras Sequential Model Structure for MLP

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten

# Create a Sequential model
model = Sequential()

# Add a Flatten layer to flatten the input image
model.add(Flatten(input_shape=(32, 32, 3)))

# Add two dense layers with 200 units and 'relu' activation function
model.add(Dense(200, activation='relu'))
model.add(Dense(150, activation='relu'))

# Add a softmax output layer with 10 units
model.add(Dense(10, activation='softmax'))

# Print the model summary
model.summary()
```

Model summary

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 flatten_2 (Flatten)          (None, 3072)              0

 dense_6 (Dense)              (None, 200)               614600

 dense_7 (Dense)              (None, 150)               30150

 dense_8 (Dense)              (None, 10)                1510

=================================================================
Total params: 646,260
Trainable params: 646,260
Non-trainable params: 0
_________________________________________________________________
```

Keras Functional API Model Structure for MLP

- from tensorflow.keras.layers import Input, Dense, Flatten
- from tensorflow.keras.models import Model
-
- # Define the input layer
- inputs = Input(shape=(32, 32, 3))
-
- # Flatten the input image
- x = Flatten()(inputs)
-
- # Add two dense layers with 200 units and 'relu' activation function
- x = Dense(200, activation='relu')(x)
- x = Dense(150, activation='relu')(x)

- # Add a softmax output layer with 10 units
- outputs = Dense(10, activation='softmax')(x)
-
- # Create the model
- model = Model(inputs=inputs, outputs=outputs)
-
- # Print the model summary
- model.summary()

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 32, 32, 3)]       0

 flatten_3 (Flatten)         (None, 3072)              0

 dense_9 (Dense)             (None, 200)               614600

 dense_10 (Dense)            (None, 150)               30150

 dense_11 (Dense)            (None, 10)                1510

=================================================================
Total params: 646,260
Trainable params: 646,260
Non-trainable params: 0
_________________________________________________________________
```

Compile and Train the MLP Model

- from tensorflow.keras import optimizers
- opt = optimizers.Adam(learning_rate=0.0005)
- model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
- model.fit(x_train, y_train, batch_size = 32, epochs = 10, shuffle = True)

```
model.fit(x_train, y_train, batch_size = 32, epochs = 10, shuffle = True)
```

```
Epoch 1/10
1563/1563 [==============================] - 16s 10ms/step - loss: 1.8440 - accuracy: 0.3346
Epoch 2/10
1563/1563 [==============================] - 15s 10ms/step - loss: 1.6654 - accuracy: 0.4040
Epoch 3/10
1563/1563 [==============================] - 16s 10ms/step - loss: 1.5856 - accuracy: 0.4352
Epoch 4/10
1563/1563 [==============================] - 16s 10ms/step - loss: 1.5304 - accuracy: 0.4530
Epoch 5/10
1563/1563 [==============================] - 16s 10ms/step - loss: 1.4916 - accuracy: 0.4693
Epoch 6/10
1563/1563 [==============================] - 16s 10ms/step - loss: 1.4615 - accuracy: 0.4813
Epoch 7/10
1563/1563 [==============================] - 17s 11ms/step - loss: 1.4312 - accuracy: 0.4915
Epoch 8/10
1563/1563 [==============================] - 16s 10ms/step - loss: 1.4068 - accuracy: 0.4977
Epoch 9/10
1563/1563 [==============================] - 16s 10ms/step - loss: 1.3838 - accuracy: 0.5066
Epoch 10/10
1563/1563 [==============================] - 16s 10ms/step - loss: 1.3659 - accuracy: 0.5123
<keras.callbacks.History at 0x7f07772e45b0>
```

Evaluate the MLP Model

```
loss, accuracy = model.evaluate(x_test, y_test)
print('Test loss:', loss)
print('Test accuracy:', accuracy)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.4257 - accuracy: 0.4929
Test loss: 1.4256958961486816
Test accuracy: 0.492900013923645
```

Functional API vs Sequential

- The Functional API method in Keras is recommended for creating more complex models that have multiple inputs, multiple outputs, or require layers to share connections.
- The Sequential model, on the other hand, is suitable for creating simple models where the layers are stacked linearly.


The End