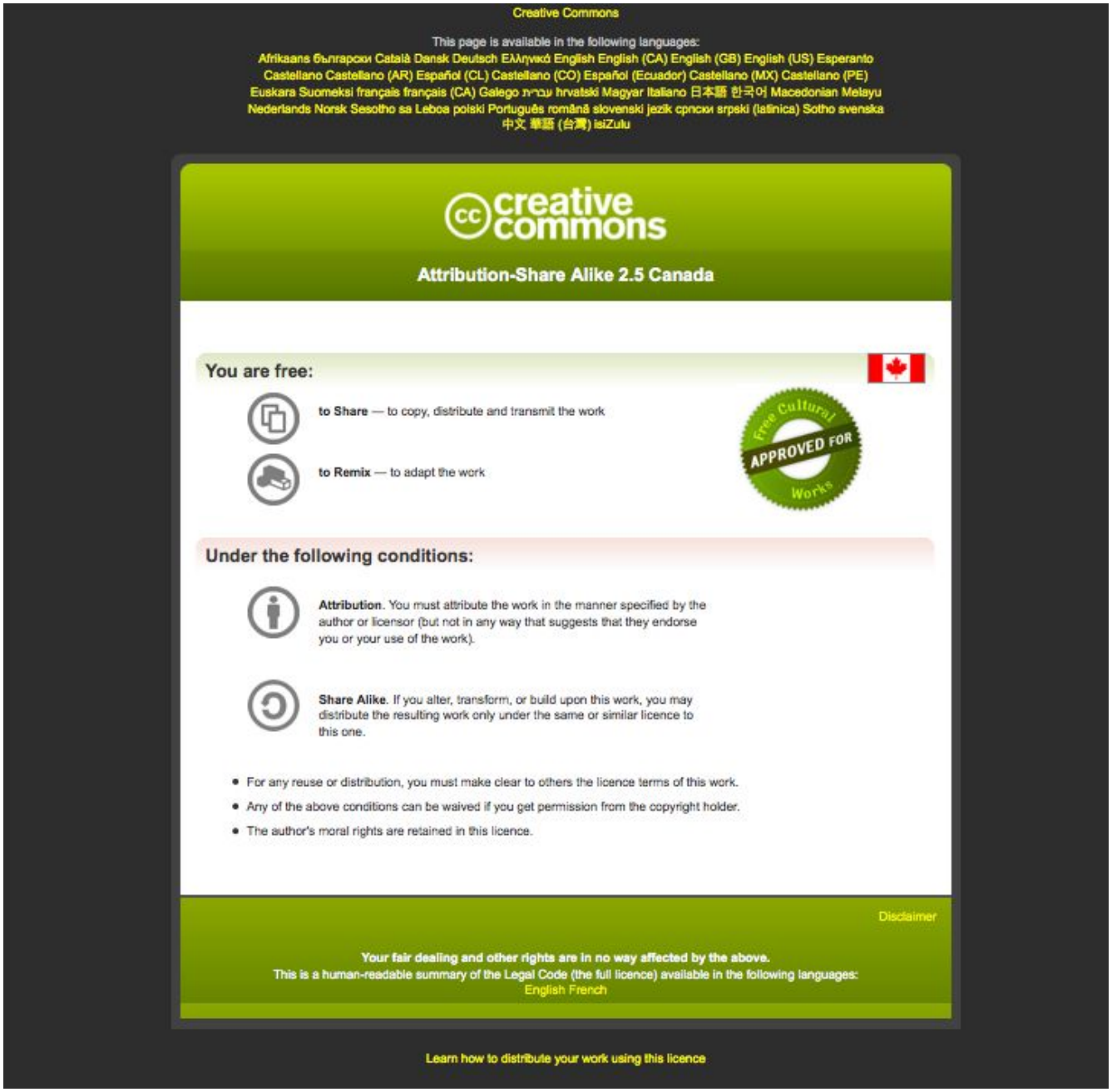
bioinformatics.ca

Canadian Bioinformatics Workshops

www.bioinformatics.ca

bioinformaticsdotca.github.io


Creative Commons
This page is available in the following languages:
Afrikaans Български Català Dansk Deutsch Ελληνικά English English (CA) English (GB) English (US) Esperanto
Castellano Castellano (AR) Español (CL) Castellano (CO) Español (Ecuador) Castellano (MX) Castellano (PE)
Euskara Suomeksi français français (CA) Galego עברית hrvatski Magyar Italiano 日本語 한국어 Macedonian Melayu
Nederlands Norsk Sesotho sa Leboa polski Português română slovenski jezik српски srpski (latinica) Sotho svenska
中文 華語 (台灣) isiZulu
creative commons
Attribution-Share Alike 2.5 Canada
You are free:
to Share — to copy, distribute and transmit the work
to Remix — to adapt the work
Free Cultural
APPROVED FOR
Works
Under the following conditions:
Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar licence to this one.
For any reuse or distribution, you must make clear to others the licence terms of this work.
Any of the above conditions can be waived if you get permission from the copyright holder.
The author's moral rights are retained in this licence.
Disclaimer
Your fair dealing and other rights are in no way affected by the above.
This is a human-readable summary of the Legal Code (the full licence) available in the following languages:
English French
Learn how to distribute your work using this licence

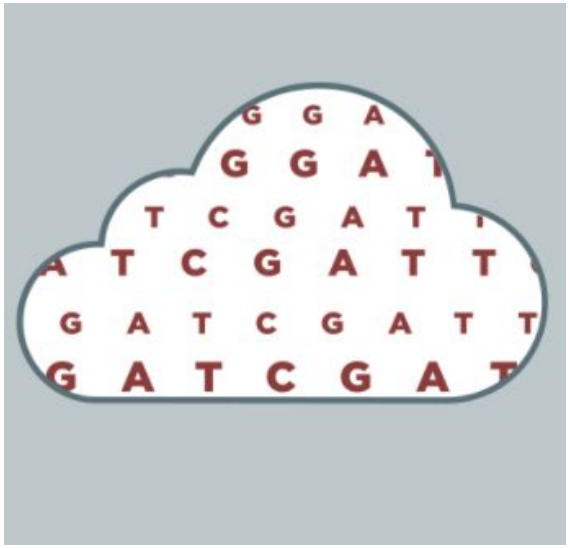
Module 3: Introduction to Neural Networks

David Wishart

Machine Learning for Bioinformatics

Aug. 16-17. 2023

bioinformatics.ca



Schedule

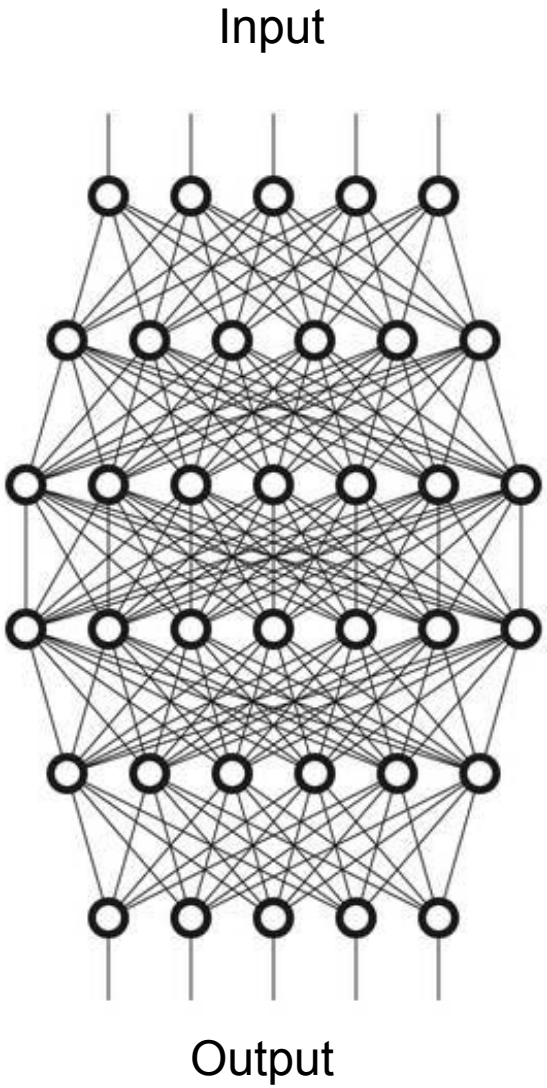
Day 1		Day 2	
Time (EDT)	Wednesday Aug. 16	Time (EDT)	Thursday Aug. 17
10:00	Introductions and Technology Check	10:00	Module 5: Gene Finding with NNs (Lecture and Lab)
10:45	Module 1: Introduction to Machine Learning (Lecture)	11:30	Break (30 min)
12:15	Break (30 min)	12:00	Module 6: Machine Learning with Keras and Scikit-Learn (Lecture/Lab)
12:45	Module 2: Decision Trees (Lecture and Lab)	14:00	Break (1 hour)
14:15	Break (45 min)	15:00	Module 7: Machine Learning with Keras and Scikit-Learn (Cont'd)
15:00	Module 3: Neural Networks (Lecture and Lab)	16:00	Break (30 min)
16:30	Break (30 min)	16:30	Module 8: Information Extraction with ChatGPT (Lecture and Lab)
17:00	Module 4: Neural Networks for 2^o Structure (Lecture and Homework)	17:45	Survey and Closing Remarks
18:00	End of Day 1	18:00	End of Day 2

Learning Objectives

- To explain how biological systems learn
- To introduce the concept of neural networks
- To explain key concepts in neural network algorithms (one-hot encoding, back propagation, hidden layers)
- To review and explain python code for an ANN for Iris classification
- To compare performance of ANNs with DTs

Artificial Neural Network

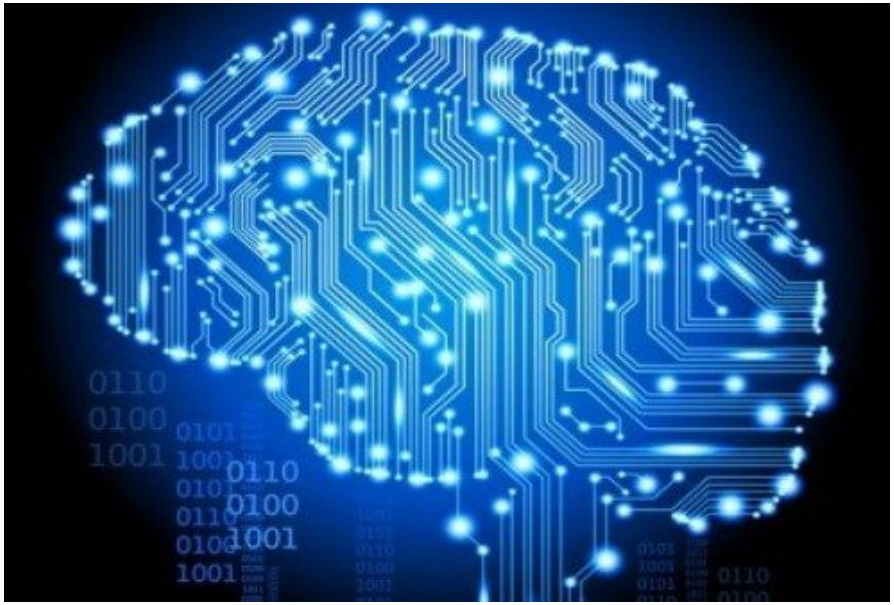
- Attempts to simulate the function and activity of the brain using inputs, outputs and neurons
- ANNs are based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain
- Can be used for both classification and regression

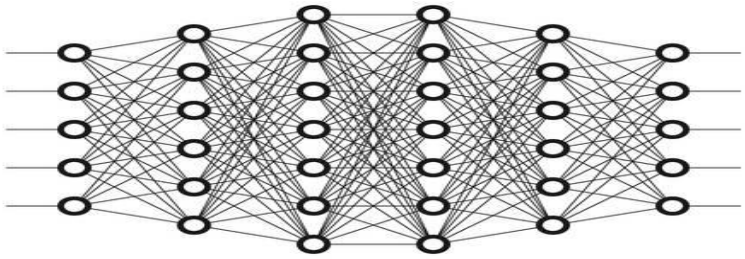


Explaining Neural Networks

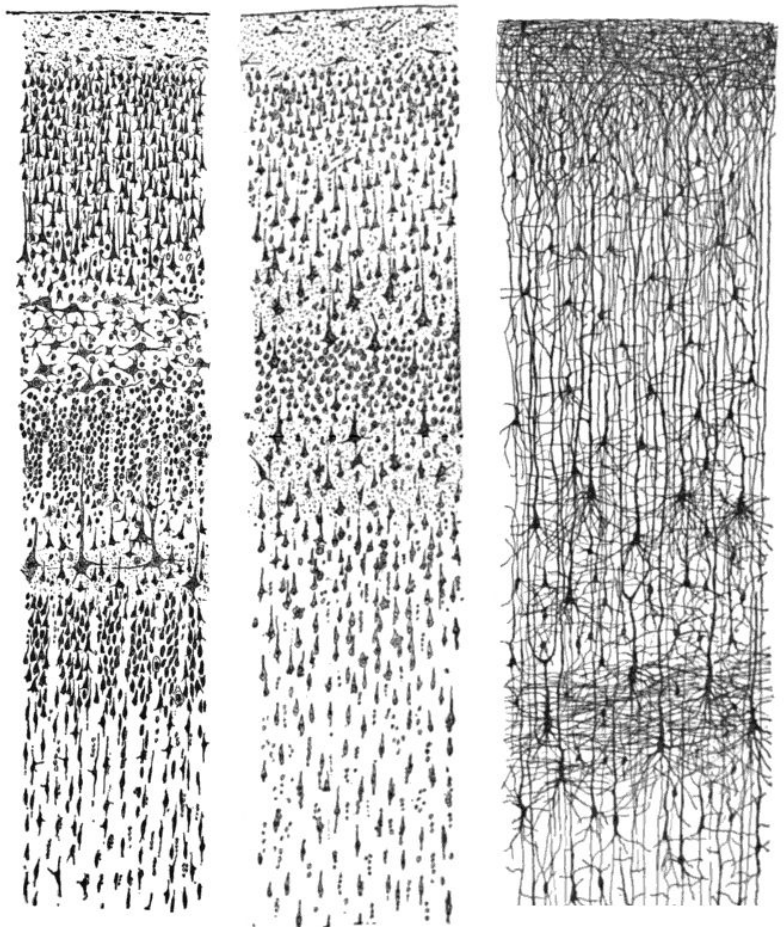
- ANNs can be used for supervised or unsupervised classification
- First described in detail in 1986
- Mimic the way the brain works
- Layers of neurons connected to each other to perform pattern analysis and logic operations
- Computer converts neuron locations and connections to tables or matrices

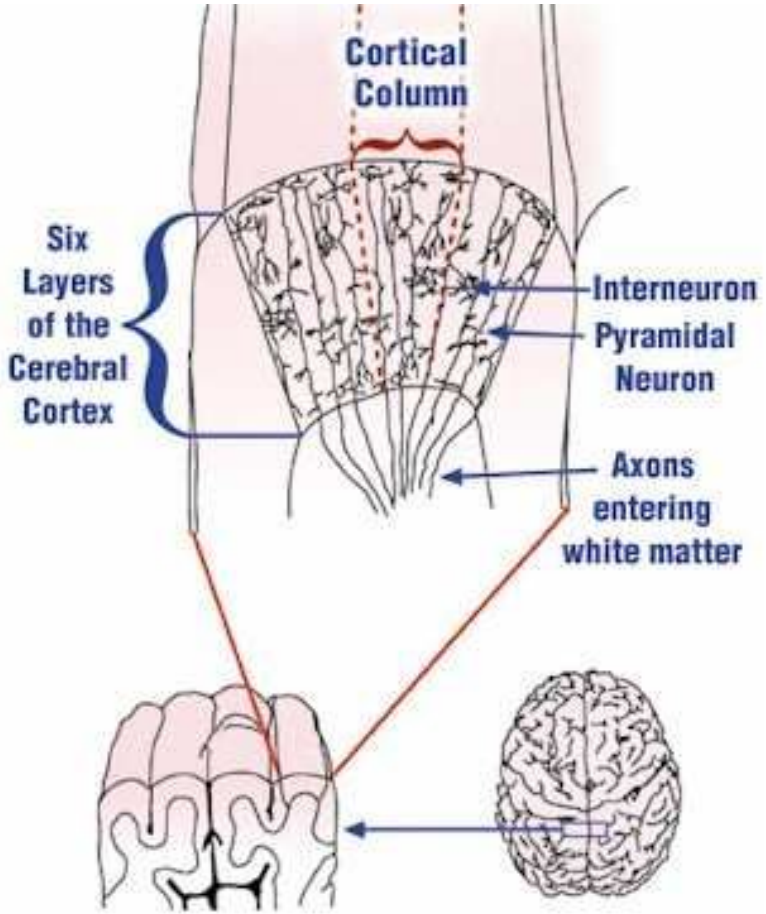
The Brain's Neural Networks



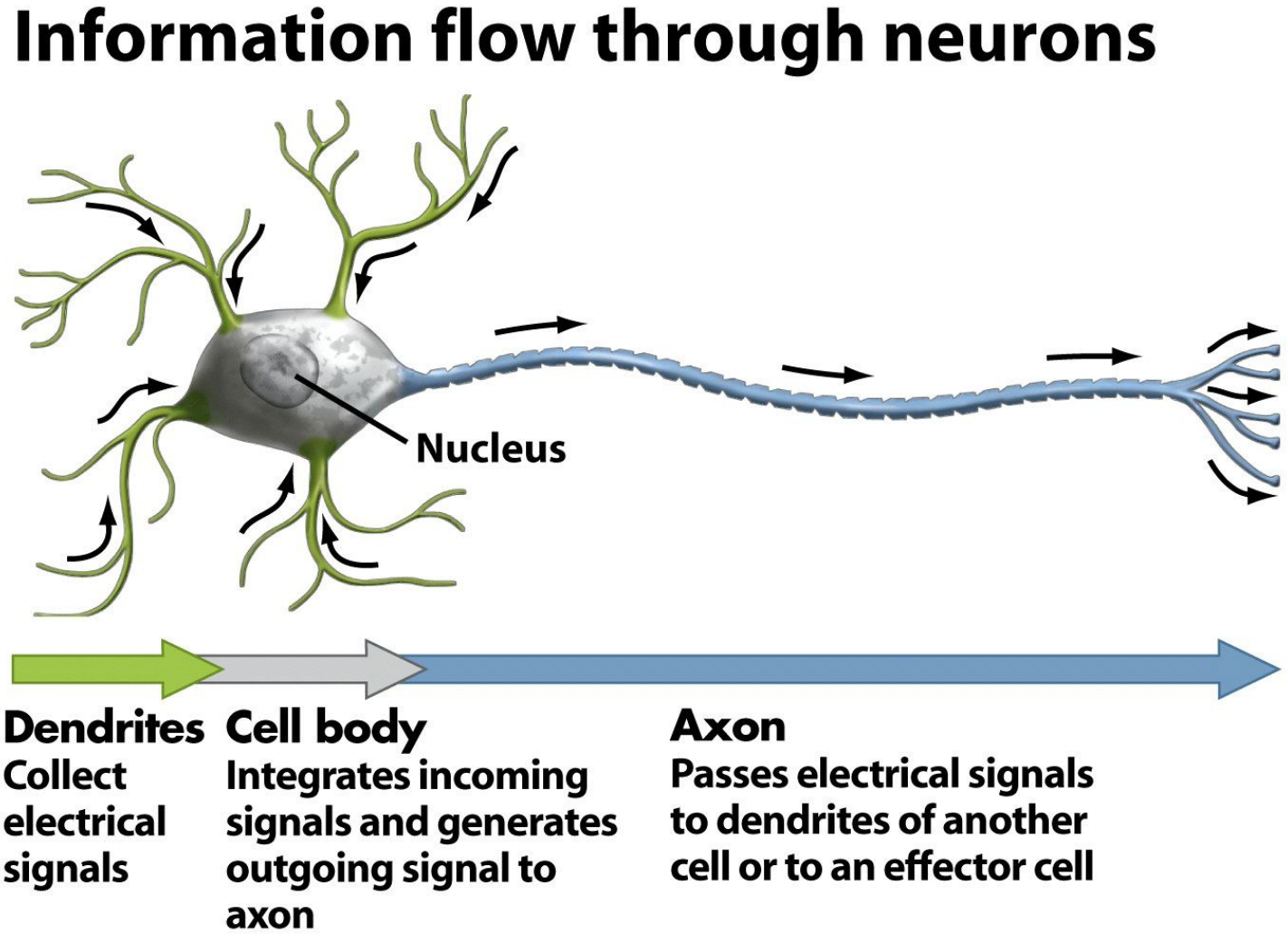


Neural Structure



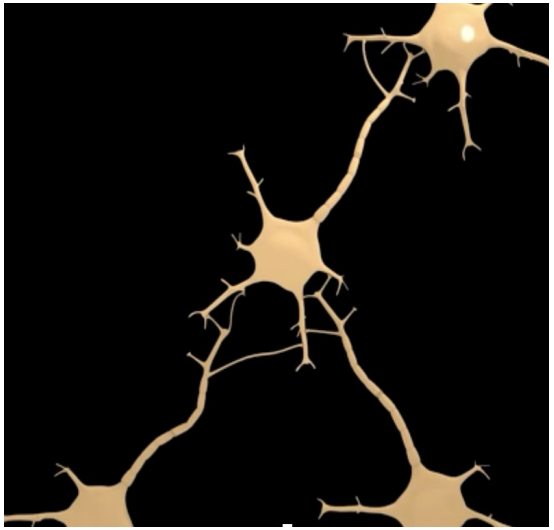


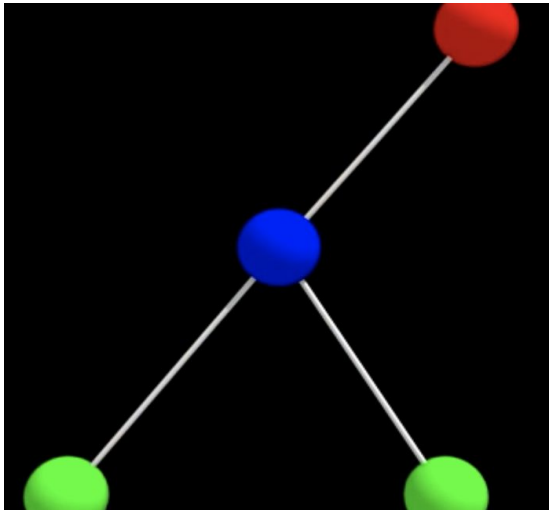
Neurons


Information flow through neurons
Nucleus
Dendrites
Collect electrical signals
Cell body
Integrates incoming signals and generates outgoing signal to axon
Axon
Passes electrical signals to dendrites of another cell or to an effector cell

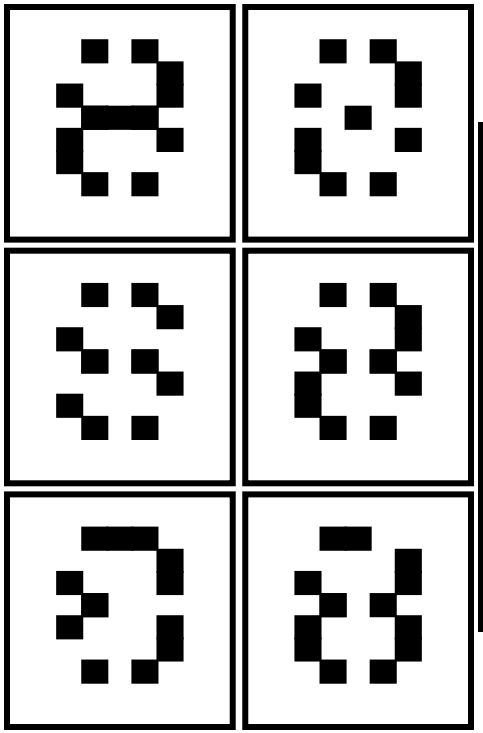
Neurons Form Networks

- Neurons in the brain accumulate signals from other neurons integrating the input signals and then if a threshold is reached, they fire, propagating signals to other neurons as outputs
- This can be schematized as a mathematical network of nodes and edges or a matrix
- The interactions between nodes can be described as addition, subtraction and differentiation functions to do Boolean logic

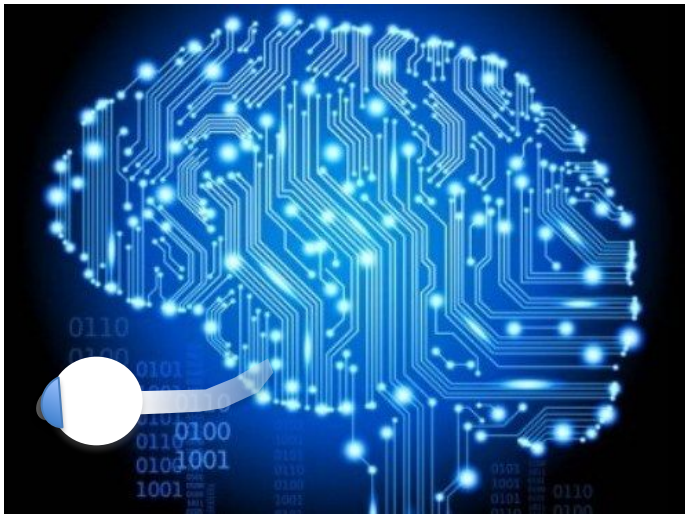


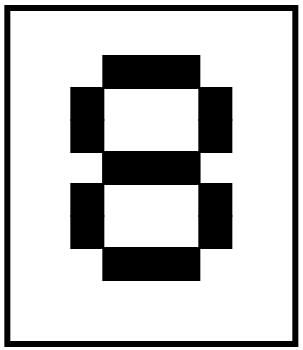


Brain's Neural Network for Image Recognition



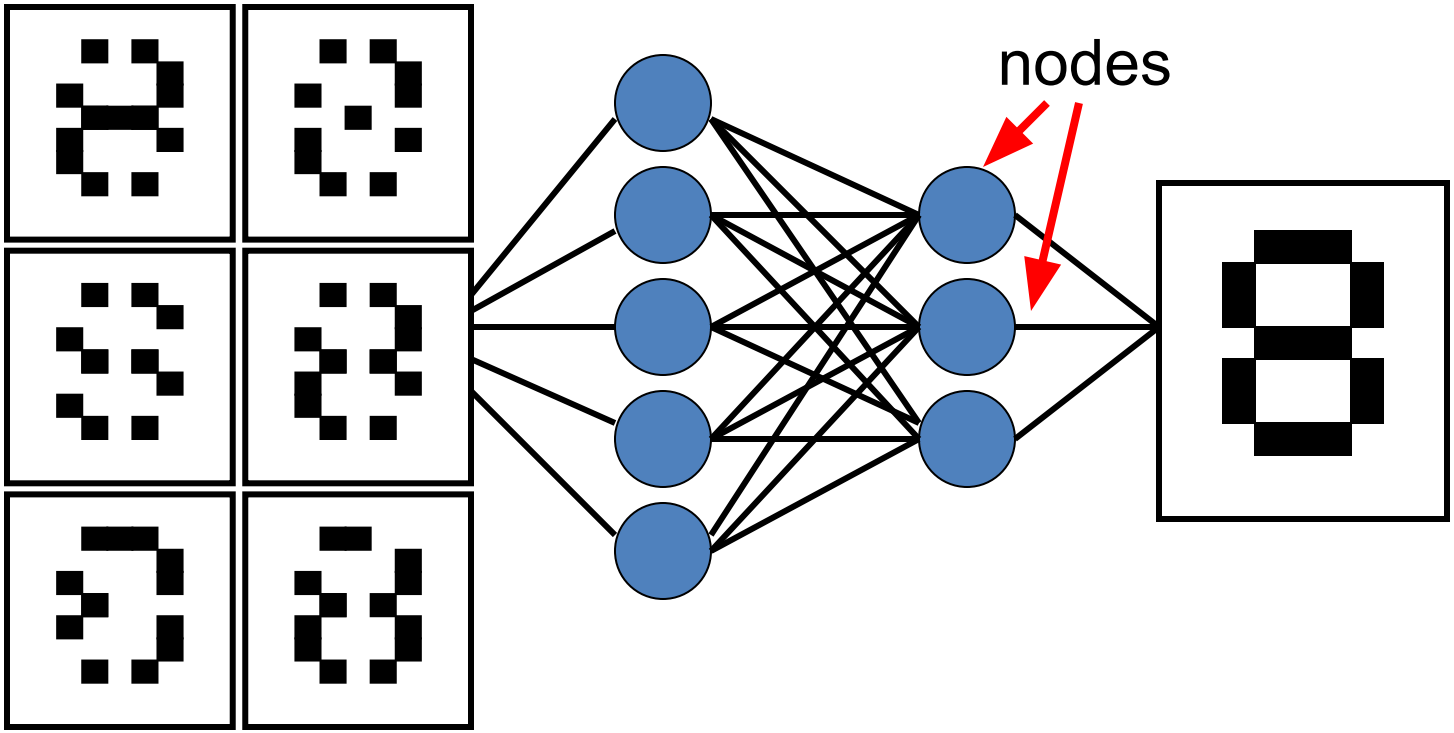
Observations



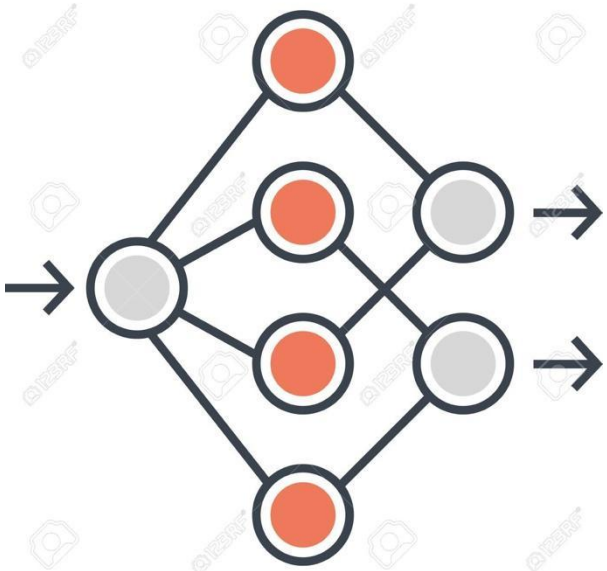


How Your Brain Interprets These Observations

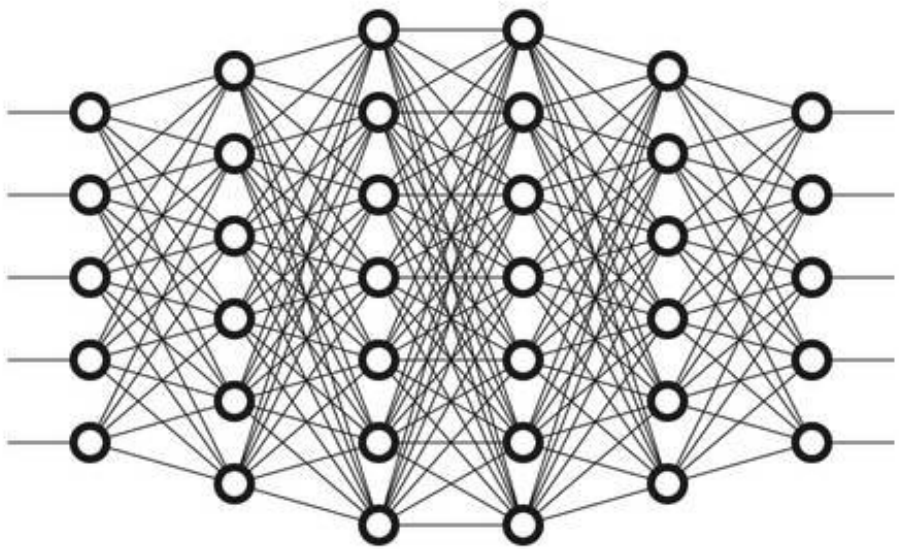
Artificial Neural Network for Image Recognition



Artificial Neural Networks & Deep Learning



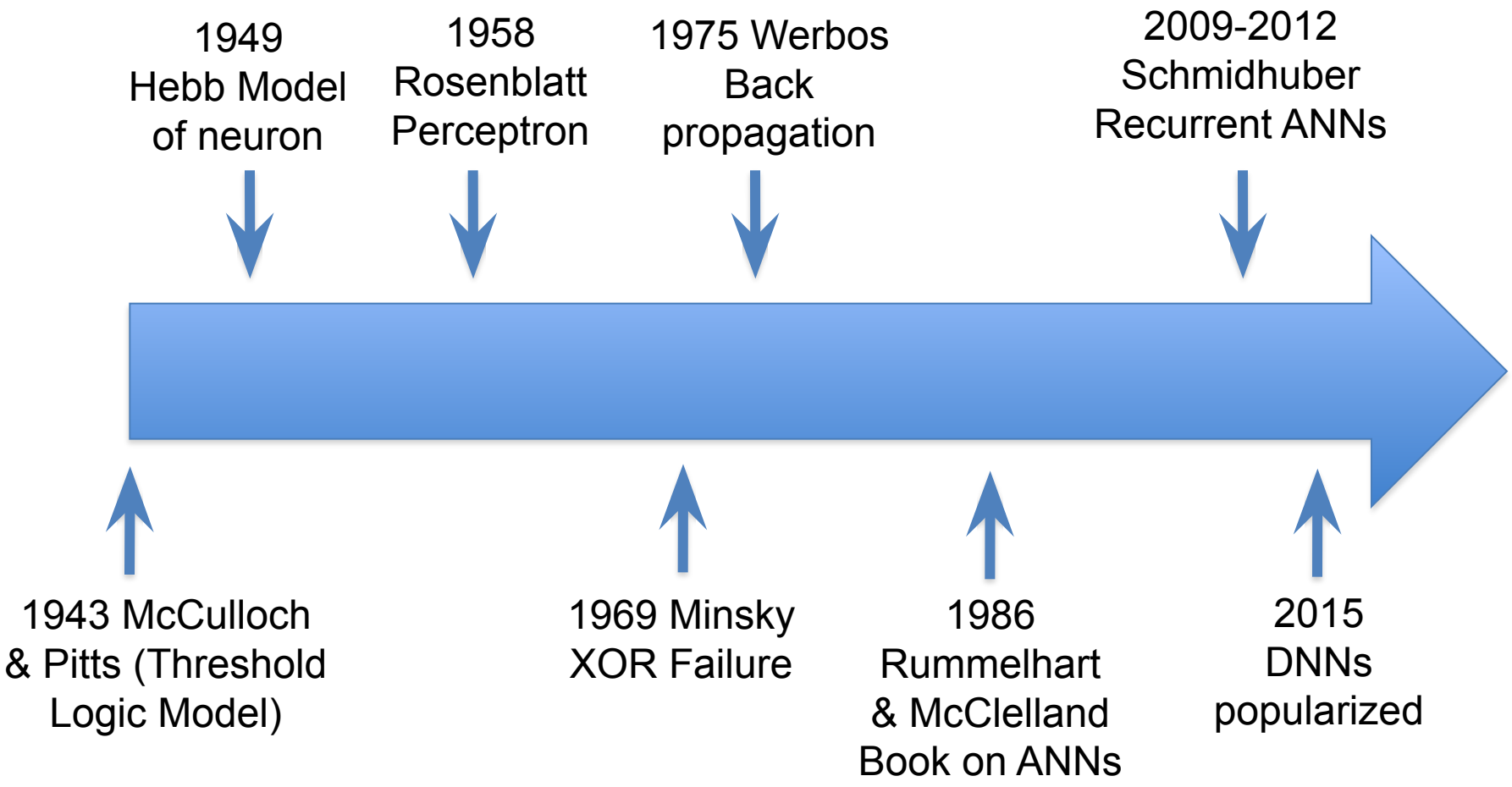
Conventional ANN



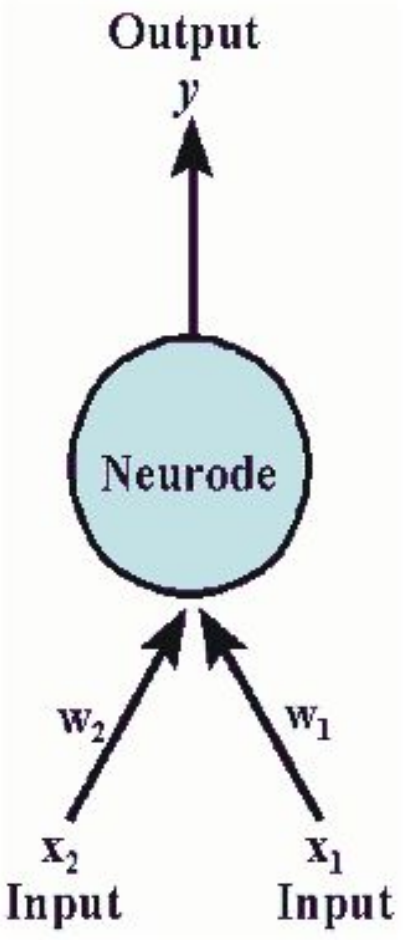
Deep Neural Net

*Deep learning is the “hottest” area of machine learning as DNNs seem to allow more complex rules/patterns to be learned or found than other ML methods

Neural Network Timeline



McCulloch & Pitts Threshold Logic



AND Function

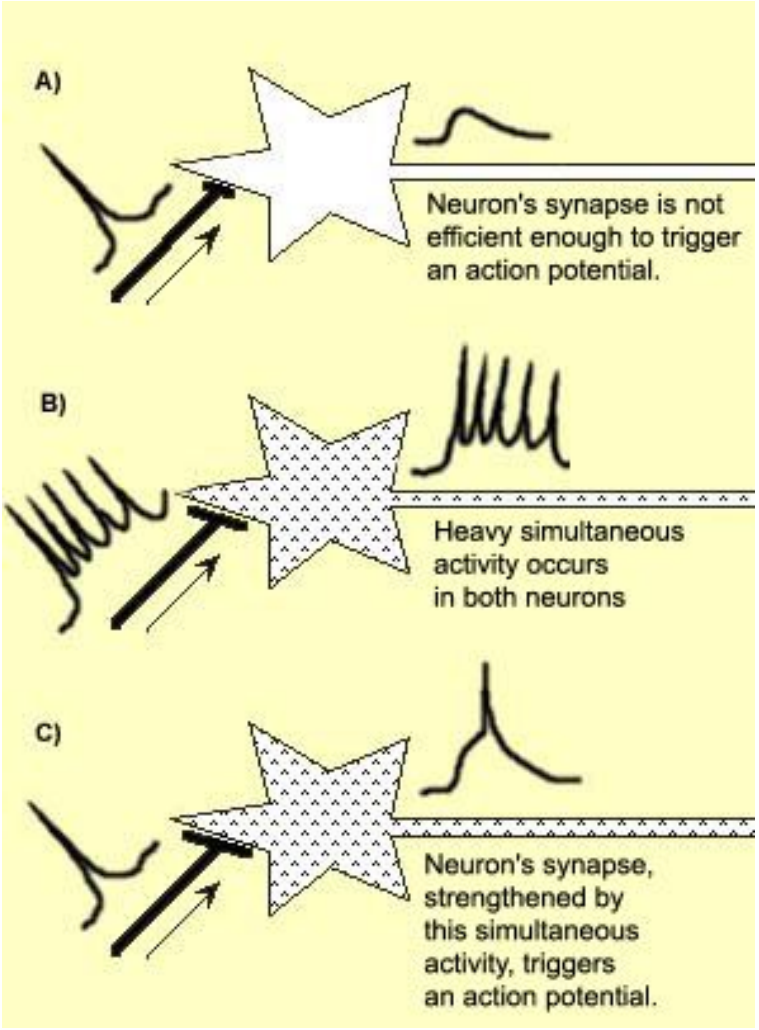
Input x_1	Input x_2	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR Function

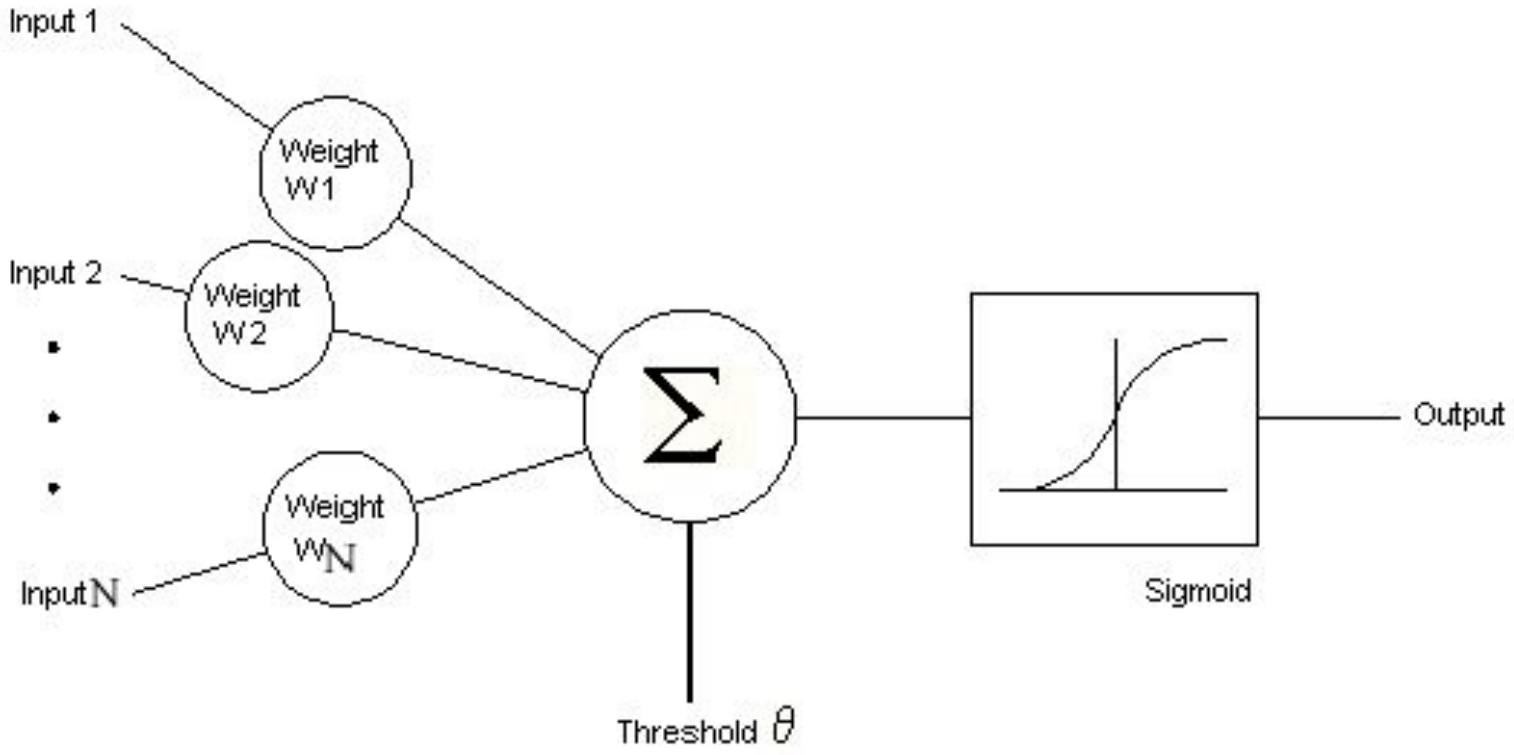
Input x_1	Input x_2	Output
0	0	0
0	1	1
1	0	1
1	1	1

Donald Hebb (McGill)

- When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased

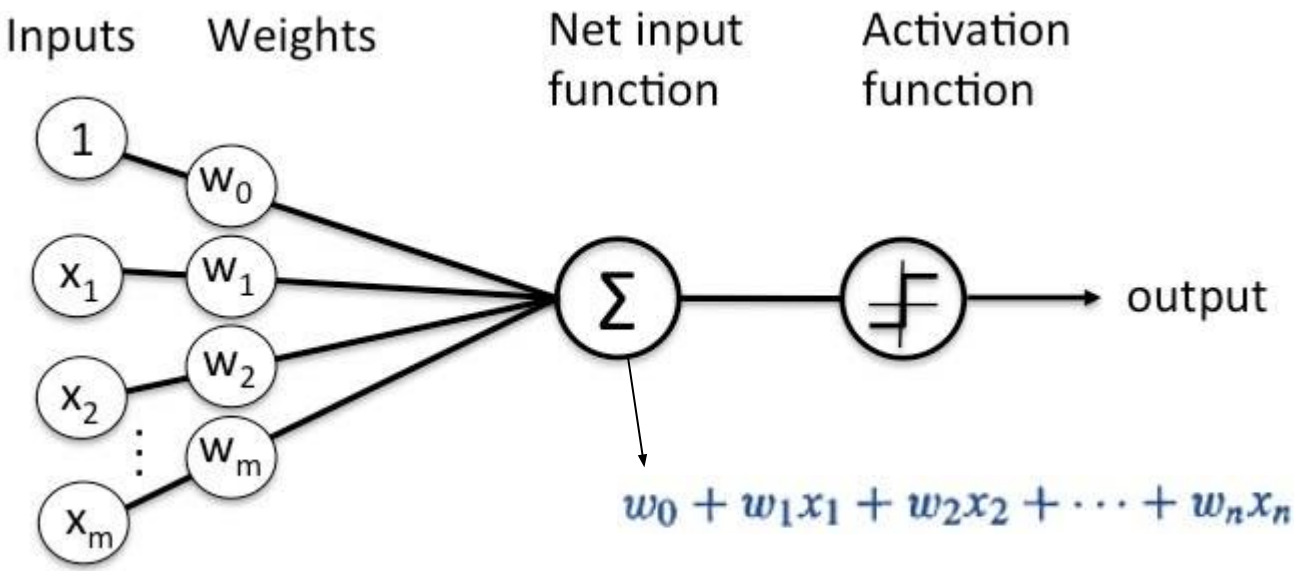


Rosenblatt Perceptron



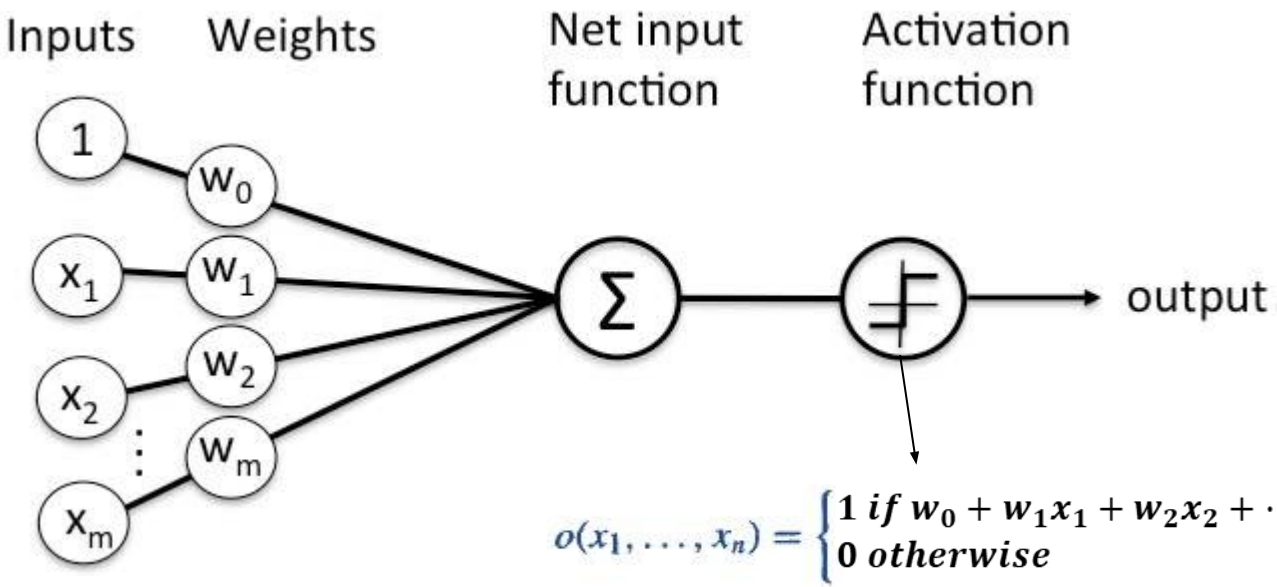
The Perceptron

- **The Perceptron is a mathematical model of a biological neuron, which takes in input values, weighs and adds them, and passes them through a function to determine an output**



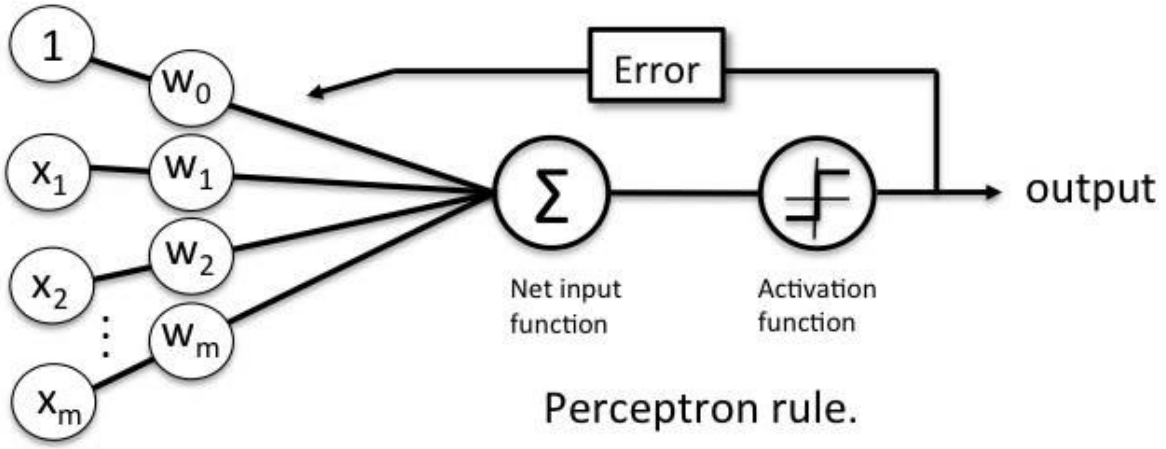
Activation Function

- The activation function receives the weighted sum of all the inputs and outputs a value
- A simple example is the step function, which will return 1 if the weighted sum is greater than 0, and 0 if otherwise



How The Perceptron Learns

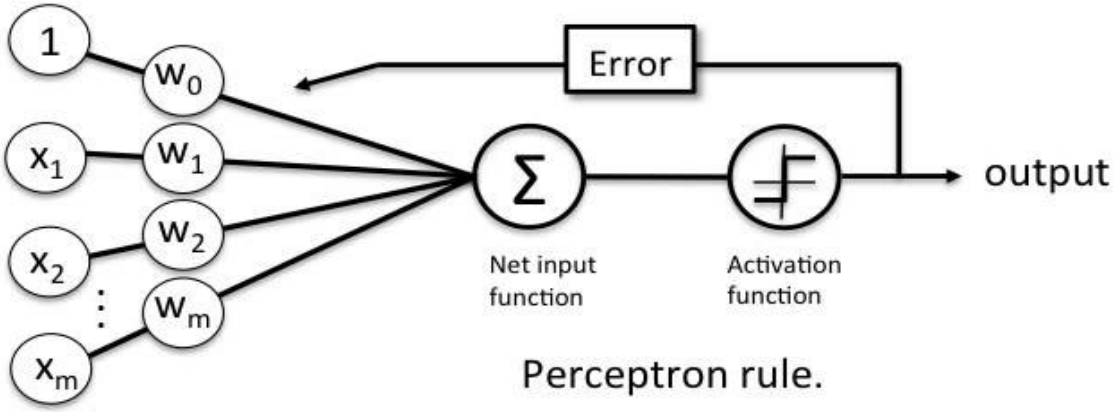
- In supervised learning, a training set is provided to the model
- This allows it to make predictions on the inputs, and compare those predictions to the actual outputs in the training set
- The error between the actual output and predicted output is then used to change the weights of the connections from the inputs
- This process is repeated until the model attains a minimum error



How the Weights Change

- To change the weights, a method called gradient descent is used
- Simply put, the weights will change according to the slope of the error function
- In this example, the value by which to update the weights is found by multiplying the errors (expected - predicted) by the inputs (x), which is further multiplied by an attenuating factor called learning rate

```
w = w - learning_rate * (expected - predicted) * x
```



Perceptron rule.

Delta Rule

$$\Delta w_{ji} = \alpha(t_j - y_j)g'(h_j)x_i,$$

where

α is a small constant called *learning rate*

$g(x)$ is the neuron's activation function

t_j is the target output

h_j is the weighted sum of the neuron's inputs

y_j is the actual output

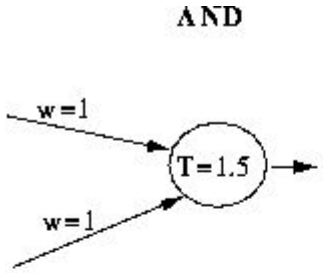
x_i is the ith input.

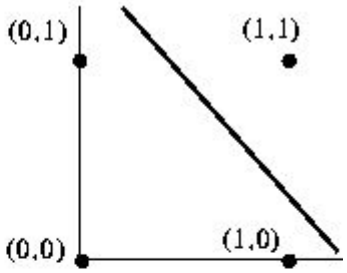
Activation Functions

Perceptron Performance

- **A perceptron, using the learning rule shown previously, is able to classify linearly separable problems, that is, problems where input data is classified into one of two possible categories along a straight line**
- **This is due to the fact that the output of the perceptron is decided based on a linear combination of its inputs**

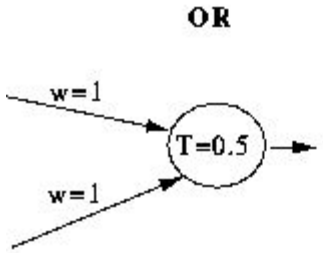
$$o(x_1, \ldots, x_n) = \begin{cases} 1 \; if \; w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ 0 \; otherwise \end{cases}$$

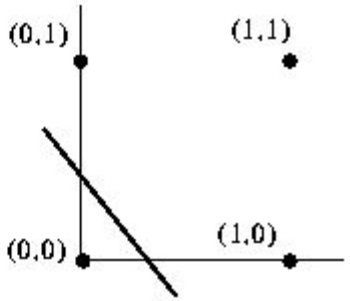




1 AND 1 = 1
0 AND 0 = 0
0 AND 1 = 0

Logical Multiplication



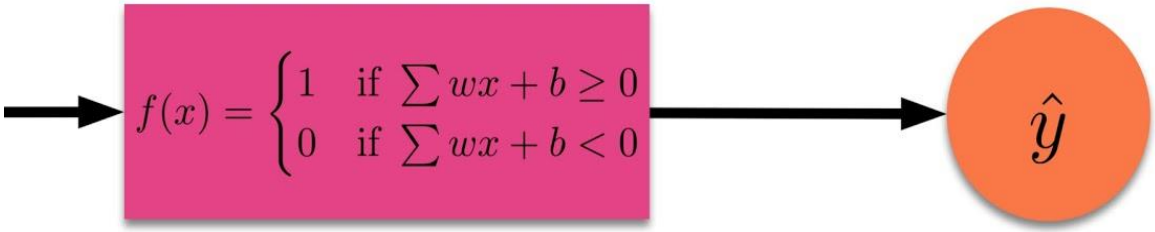


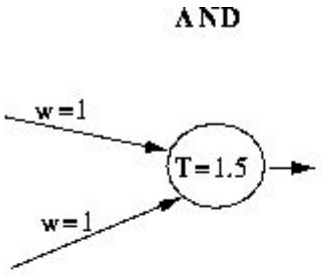
1 OR 1 = 1
0 OR 0 = 0
0 OR 1 = 1

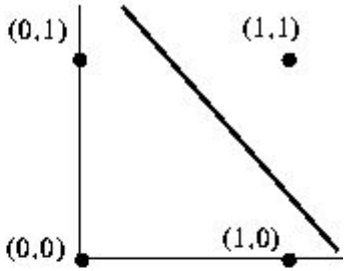
Logical Addition

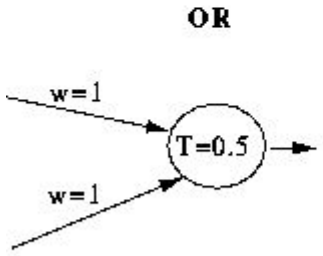
Perceptron Performance *cont...*

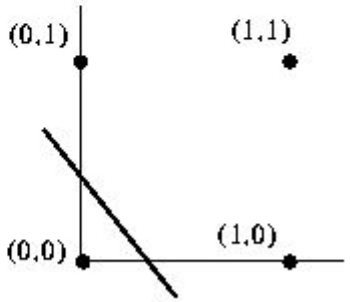
- These examples show the performance of a perceptron in modeling the AND and OR logical functions
- If the weighted sum of the two inputs x1 and x2 is greater than the threshold T, then the output is classified as 1
- Otherwise, the output is classified as 0





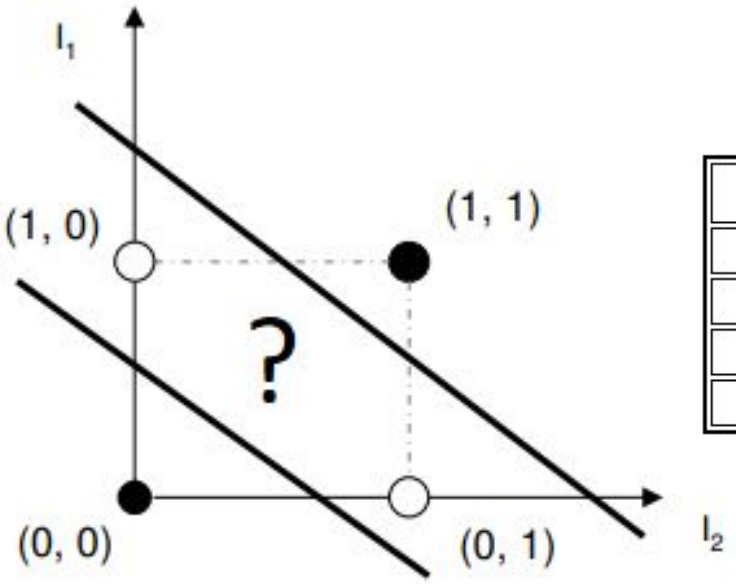






Limits of Perceptron Performance

- Because the output is dependent on a linear combination of inputs, a perceptron cannot classify data that is not linearly separable
- A simple example of this is the logical function XOR
- Because no single line is able to separate the two classes, a perceptron will not be able to categorize the data

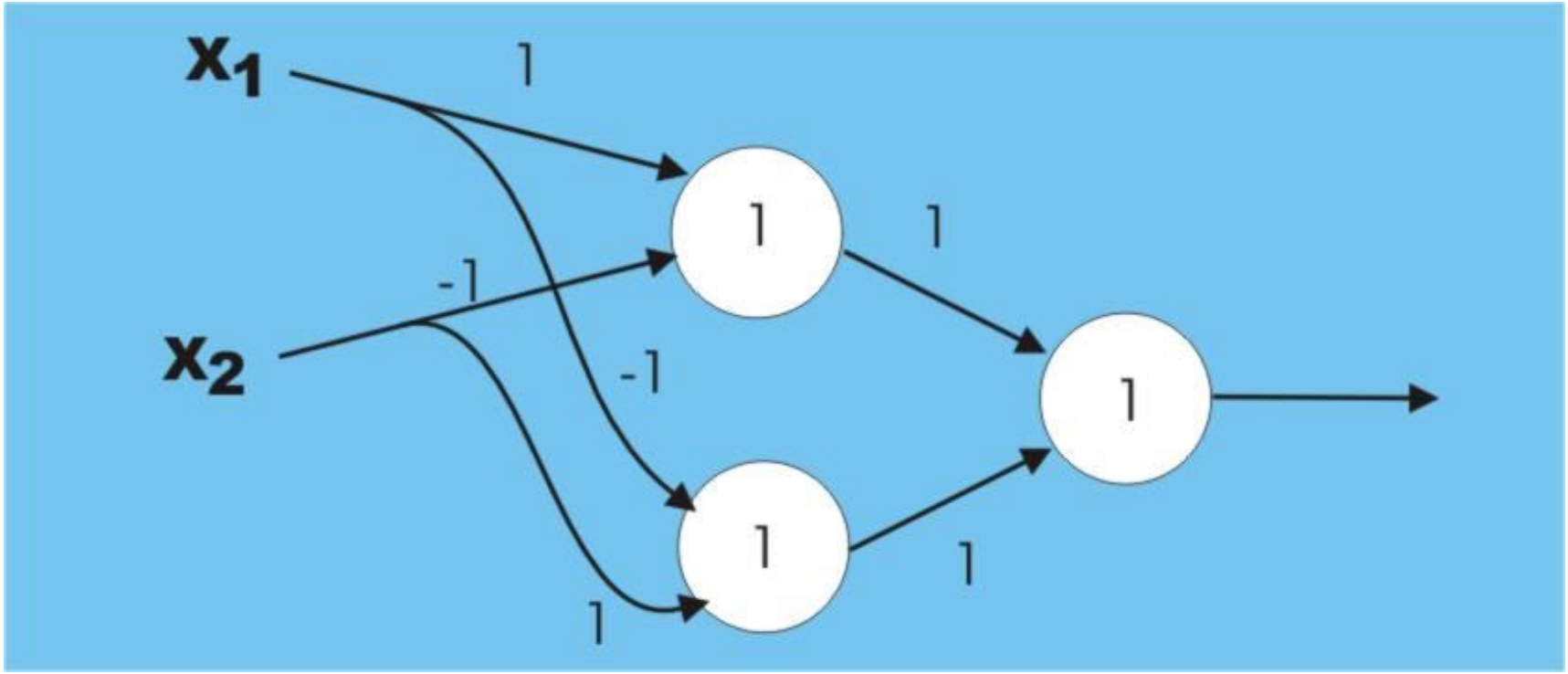


XOR

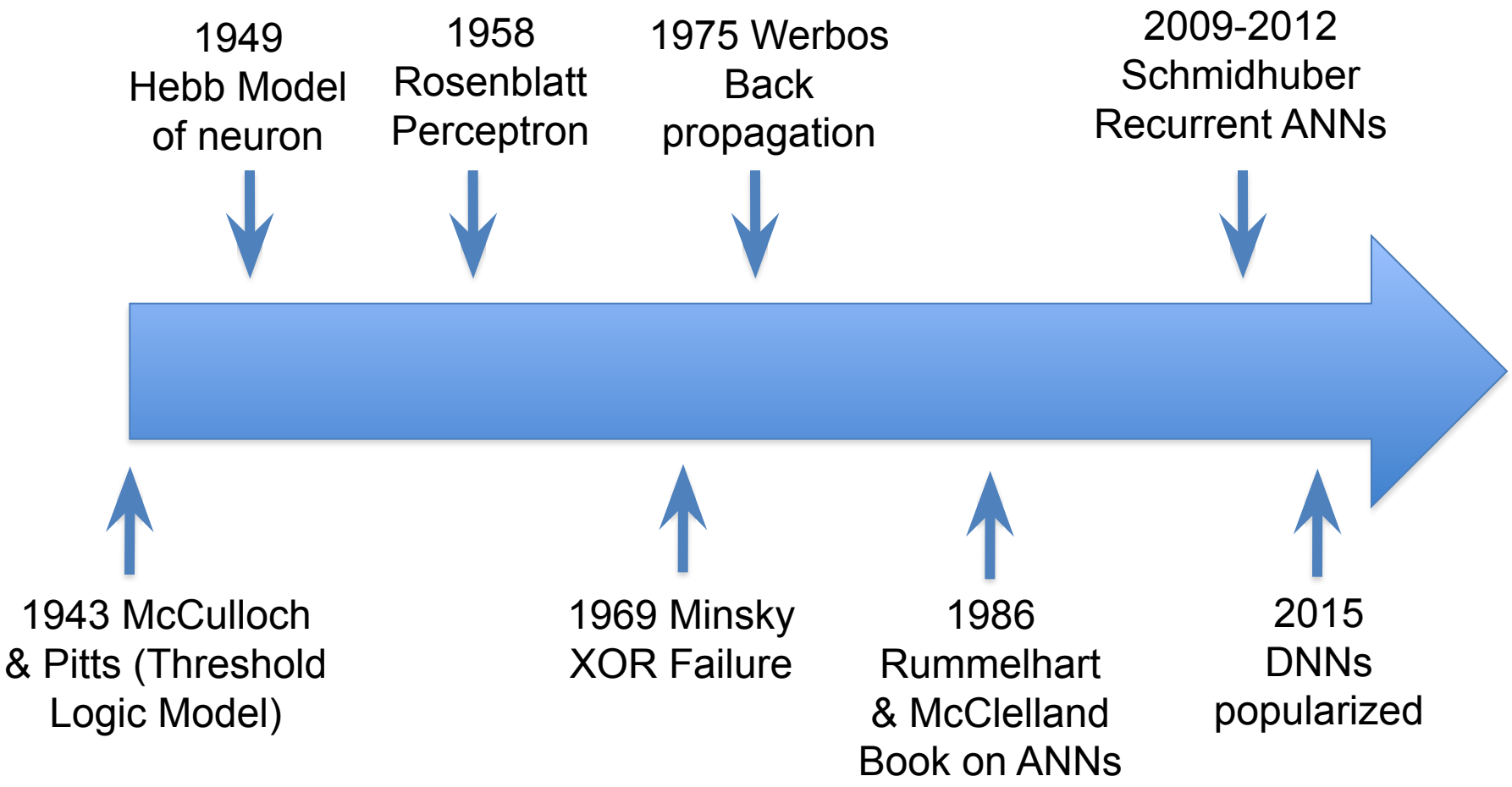
Input x_1	Input x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Logical Difference or Inequality

XOR Requires 2 layers



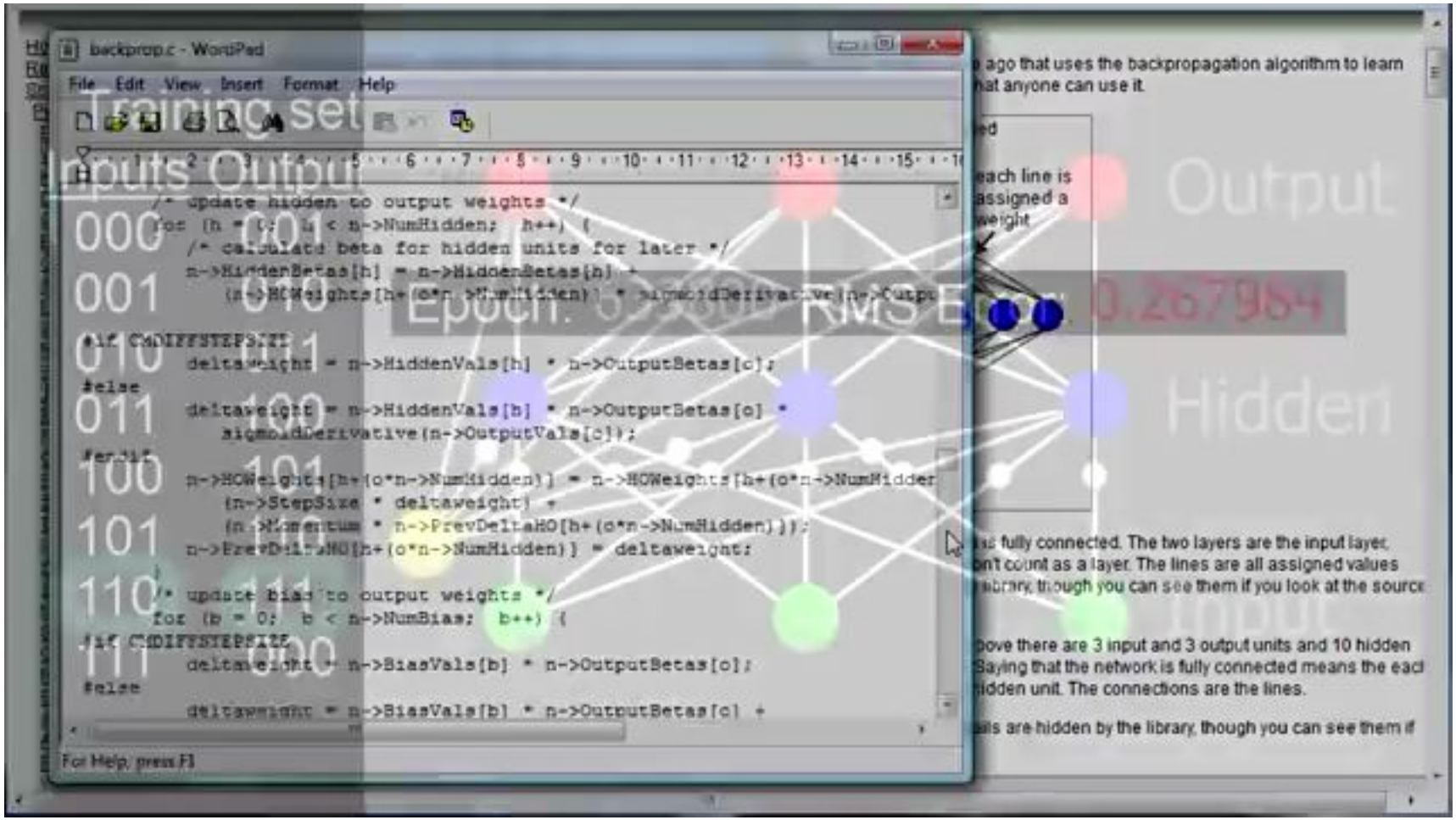
Neural Network Timeline



Neural Networks or Multi-Layered Perceptron (MLP)

- Extension of the perceptron
- Uses multiple layers to mimic layered structure of the brain
- NNs contain at least three layers, an input layer, a hidden layer, and an output layer
- Multiple layers allow perceptrons to exhibit more learning and computational complexity
- Uses more sophisticated learning algorithm (back propagation)
- Capable of handling XOR, therefore can handle nonlinear separation and classification
- “Rediscovered” and revived in the 1980s

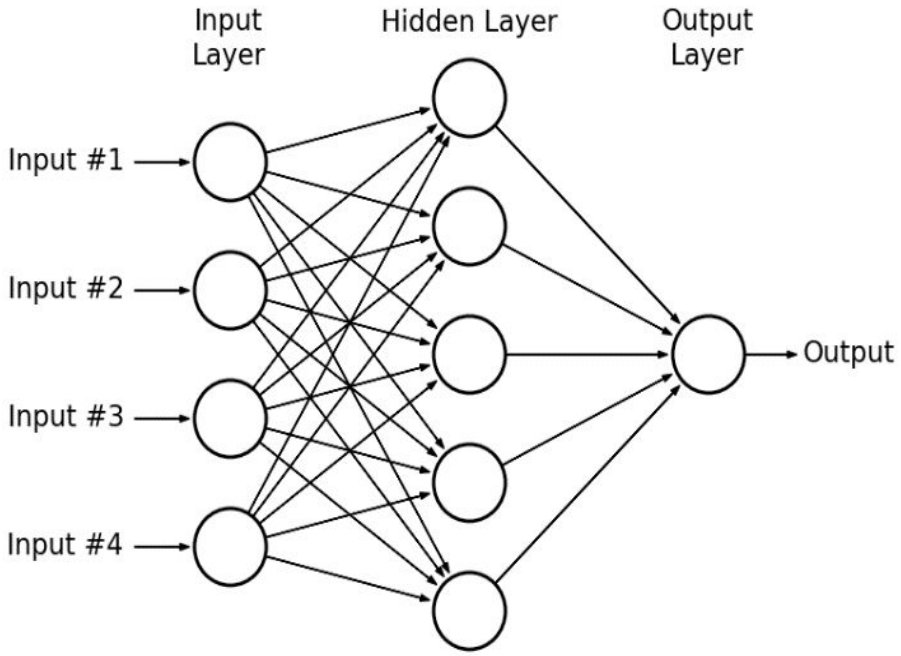
Neural Network Video



https://www.youtube.com/watch?v=WZDMNM36PsM

Neural Networks

- **Feedforward neural networks (NNs) or artificial neural networks (ANNs) or multi-layer perceptrons (MLPs) are known as universal function approximators, and are able to distinguish data that are not linearly separable**



ANN Terminology

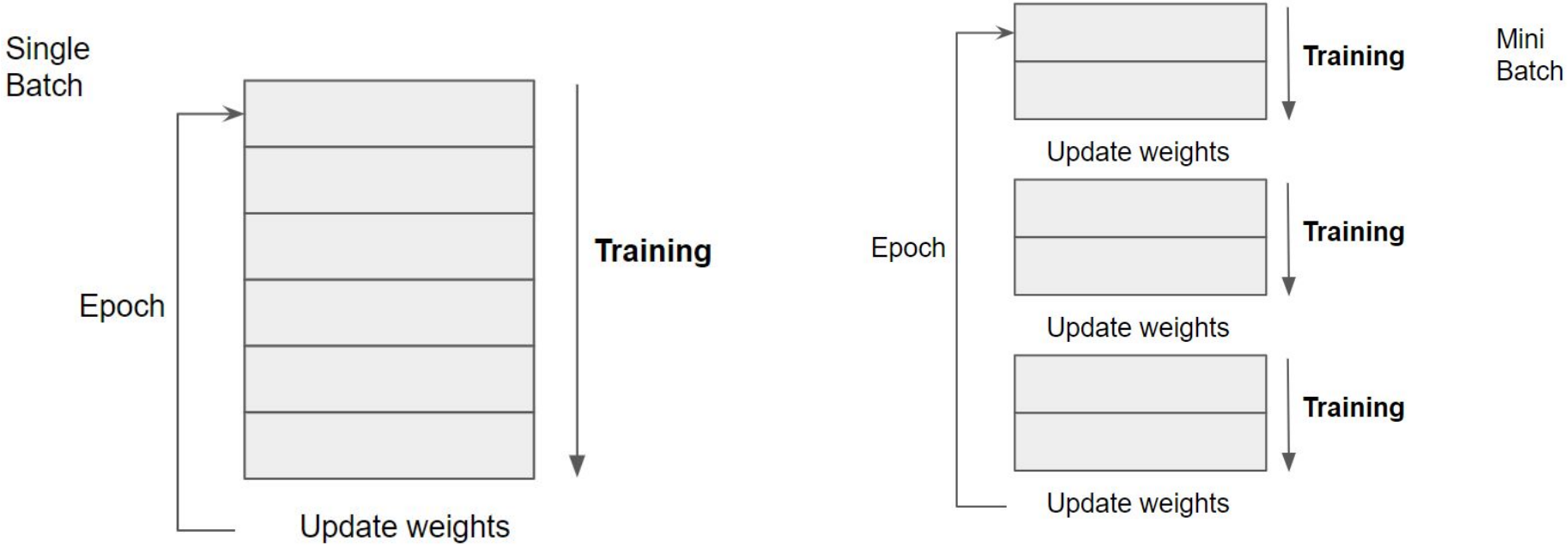
- **Hidden Layer** – Any layer between the input and output is a hidden layer. These layers perform transformations on, and make associations between inputs.
- **Forward Propagation** – The first step in learning, where a sample (input) is passed through the network. Inputs are multiplied by weights and evaluated by the activation function.

ANN Terminology

- **Backpropagation** – The ANN uses the error between the predicted output and true output to change the weights and biases of previous layers
- **Epoch** – The set of samples used to train the ANN can be applied multiple times. One iteration of the training set is known as an epoch

ANN Terminology

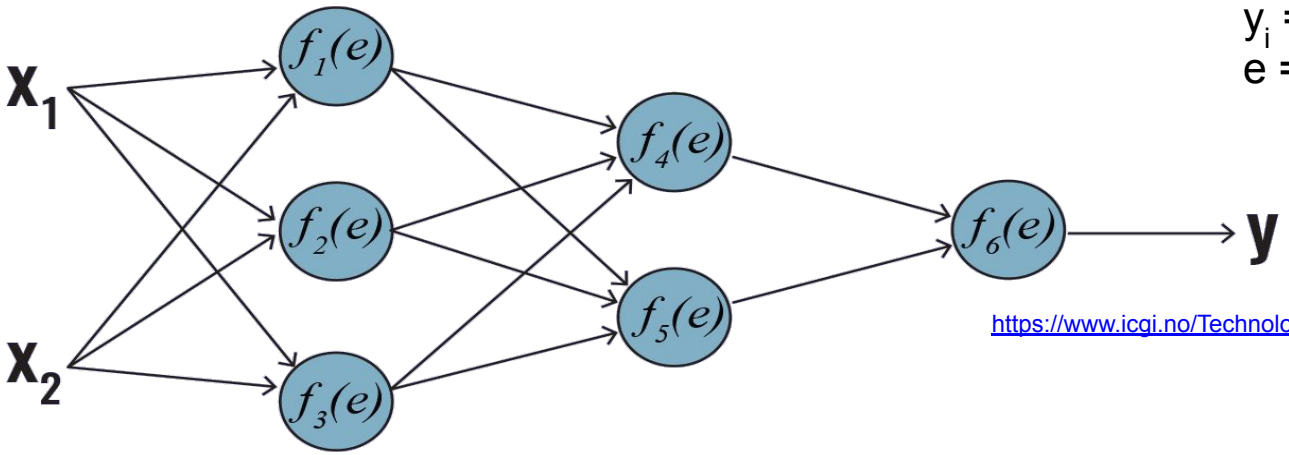
- **Batch Learning** – In batch learning, the training set is first broken up into batches
- Batch learning allows weights to be updated more frequently per epoch



ANN Learning: Forward Propagation

- **Similar to a single perceptron, an MLP learns by propagating a training set forward, finding the error between the predicted values and actual values, and using that error to change the weights and biases to reduce that error**
- **Forward propagation involves taking the weighted sum of inputs and applying the activation function**
- **The output of the activation function is then propagated to the next layer**

Arrows = matrix elements
y_i = outputs
e = linear comb of wgts x input



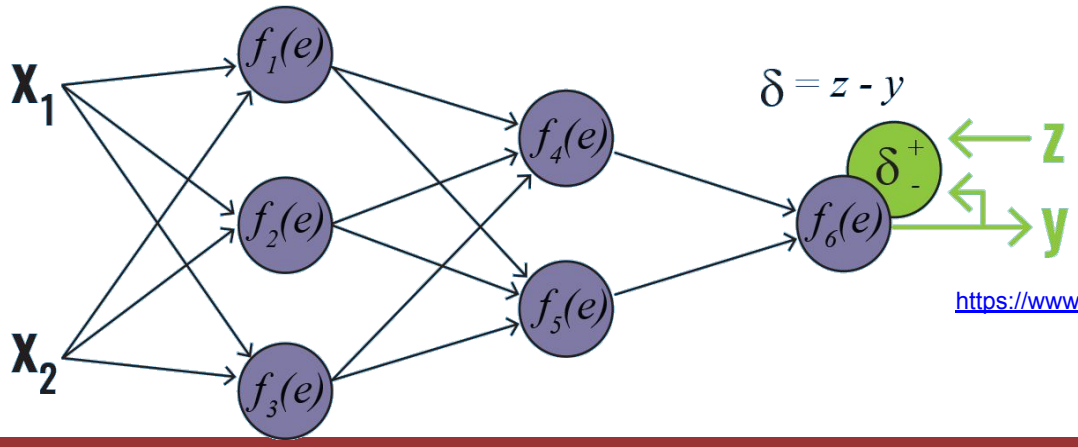
https://www.icgi.no/Technology/Deep-learning

Input Hidden 1 Hidden 2 Output

ANN Learning: Back Propagation

- The next step in learning is back propagation
- The output error is found by subtracting the actual value of the output by the predicted value
- To determine the error in output caused by error in previous layers, the output error is back propagated
- This is done by multiplying the error by the weight matrix of the previous layer
- The '□' here represents the derivative of the cost function with respect to the predictions

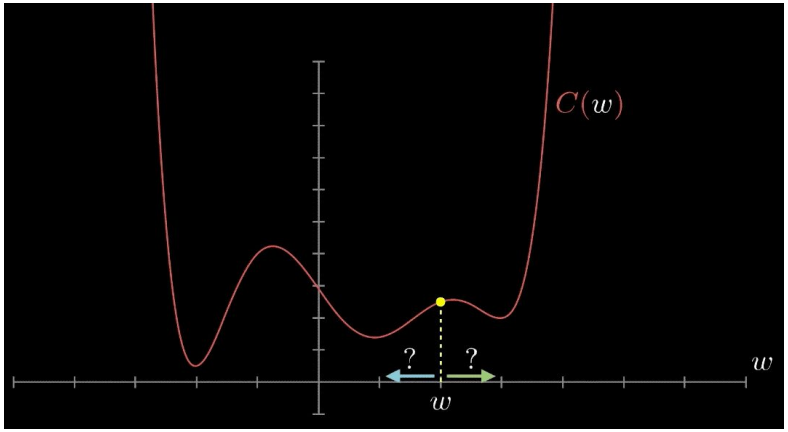
Cost function = cross entropy



https://www.icgi.no/Technology/Deep-learning

ANN Learning: Gradient Descent

- The cost function is the function used to determine the error between the output found by the model and the actual output
- To minimize the cost function, we use the derivative with respect to the weights
- This gives the slope of the cost function, and therefore which direction the weights should be moved
- If the derivative of the cost function at a particular set of weights is negative, then this subtracted by the current weights will increase the weights



ANN Learning: Updating Weights

- **Using '□s' found earlier, we can find the derivative of the cost function with respect to the weight matrices**
- **To do this, the '□' for a given layer is multiplied by the derivative of the cost function with respect to the input of that layer**
- **This is further multiplied by the corresponding input**
- **This product is then multiplied by the constant factor 'η' called learning rate**

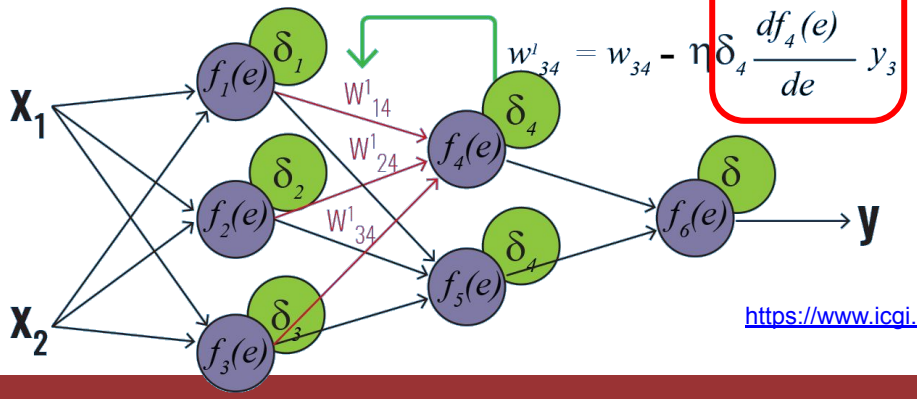
δ is the derivative of cost function
df/de is derivative of activation function relative to input "e" (weighted sum of inputs)
y_i is output of each neuron

$$w^1_{14} = w_{14} - \eta\delta_4 \frac{df_4(e)}{de} y_1$$

$$w^1_{24} = w_{24} - \eta\delta_4 \frac{df_4(e)}{de} y_2$$

$$w^1_{34} = w_{34} - \eta\delta_4 \frac{df_4(e)}{de} y_3$$

Derivative of the cost function with respect to weights



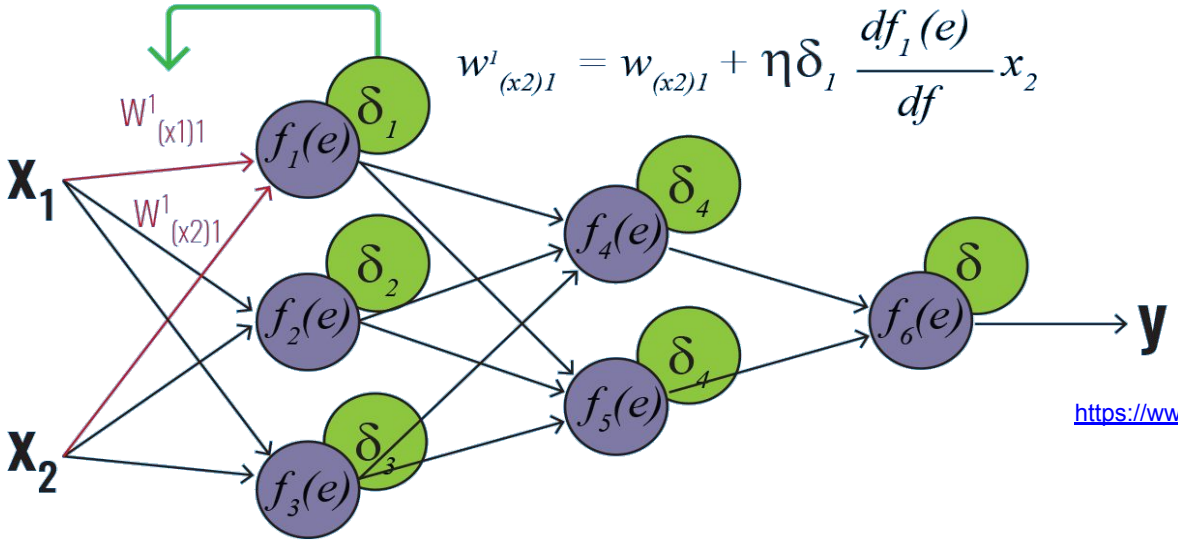
https://www.icgi.no/Technology/Deep-learning

ANN Learning: Updating Weights

- The total product is subtracted from the weights to find the new weight matrix
- Through this algorithm, the weights will be updated so that the cost function will be decreased

$$w^1_{(x1)1} = w_{(x1)1} + \eta\delta_1 \frac{df_1(e)}{df} x_1$$

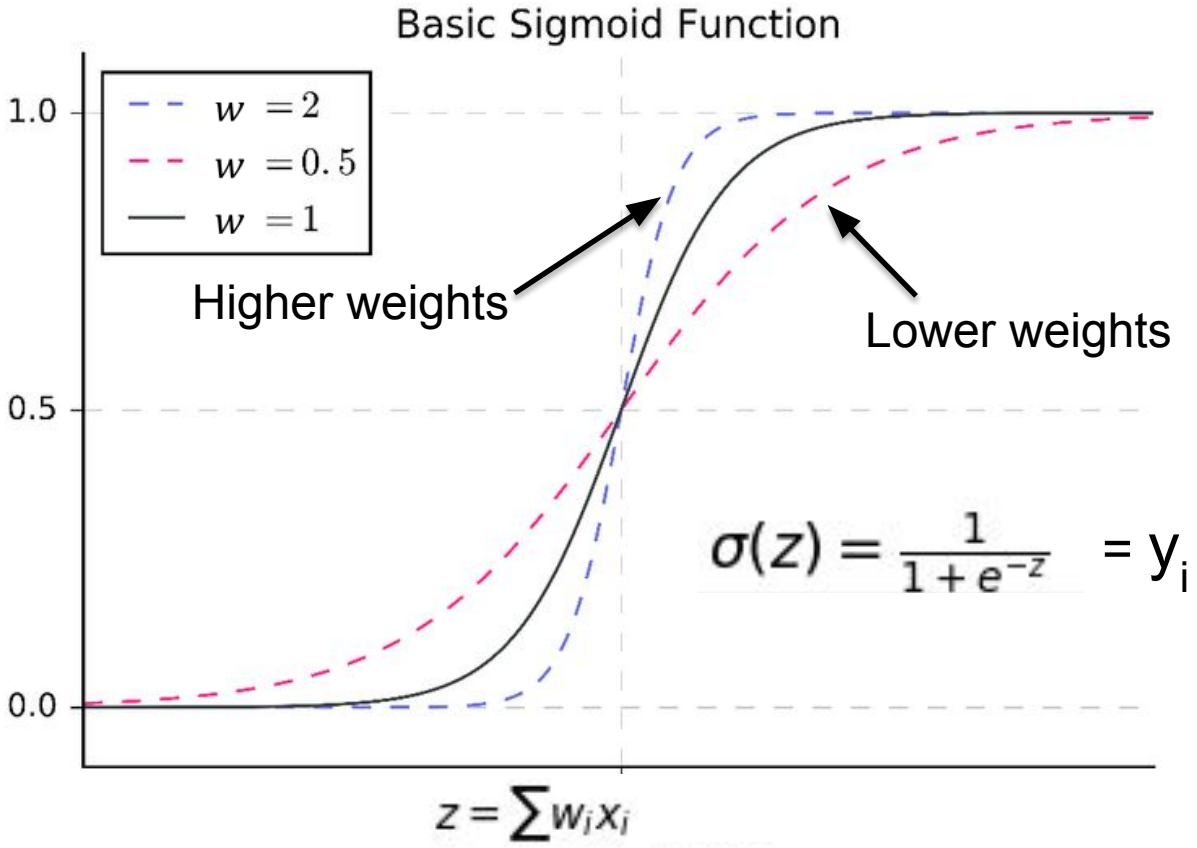
$$w^1_{(x2)1} = w_{(x2)1} + \eta\delta_1 \frac{df_1(e)}{df} x_2$$



https://www.icgi.no/Technology/Deep-learning

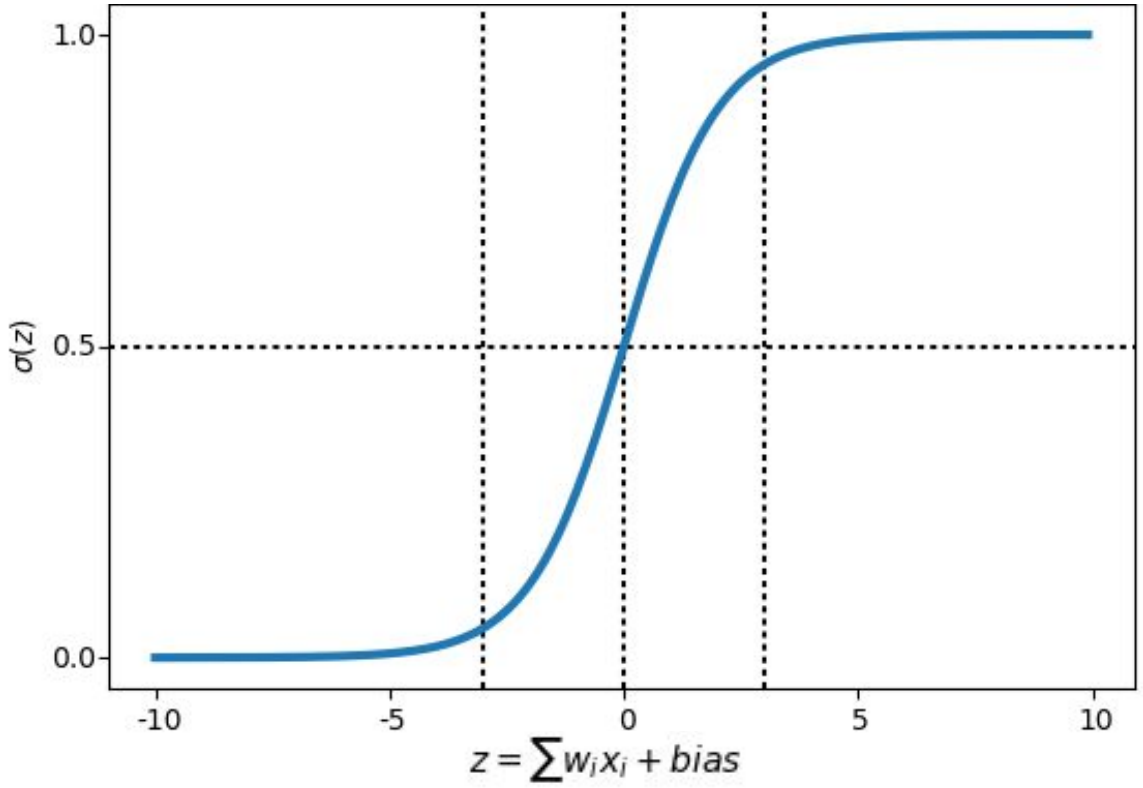
Effects of Weight Changes

- **Changing the weights can be understood as changing the steepness of the activation function curve**



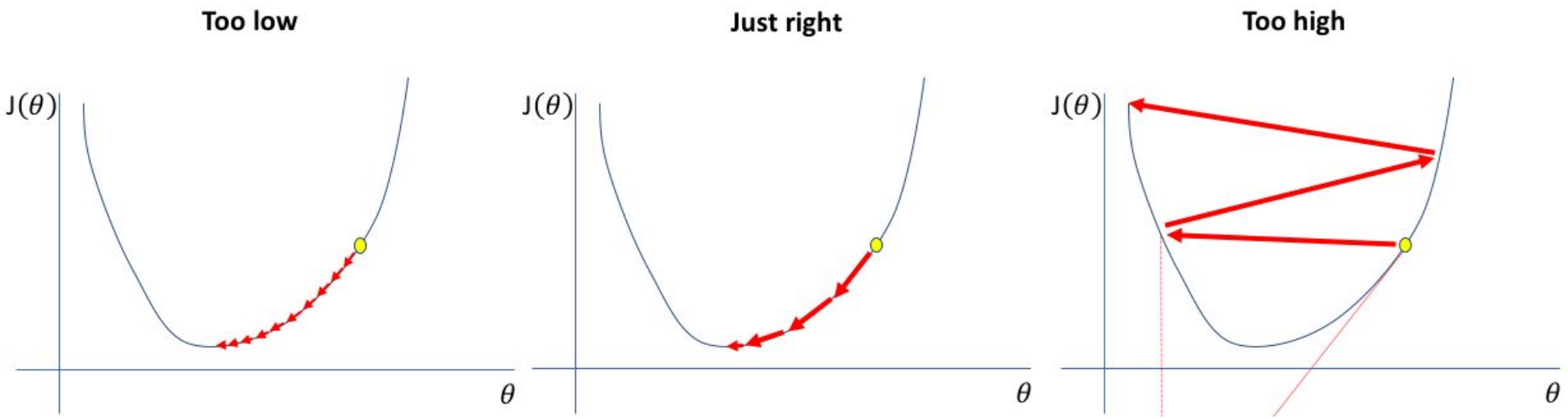
Effects of Bias Changes

- A bias term is used to shift the activation function left or right
- For example, to get an output of 0.5 for an input of 5, we can apply a bias of -5



Effects of The Learning Rate

- The learning rate (η) is a number between 0 and 1 used to decrease the amount by which the weights and biases are changed
- Without adjusting learning rate, the model will overshoot the minimum of the cost function



ANN Mathematically

$$\frac{1}{1 + e^{-x}}$$ Activation Function

1 x 2 array

[0 1]

2 x 3 array

$$\begin{bmatrix} .2 & .4 & .1 \\ .1 & .0 & .4 \end{bmatrix}$$

1 x 3 array

[.1 .0 .4]

3 x 1 array

$$\begin{bmatrix} .1 \\ .0 \\ .3 \end{bmatrix}$$

[.13]

compare

[0] Desired output

Input Vector

Weight Matrix1

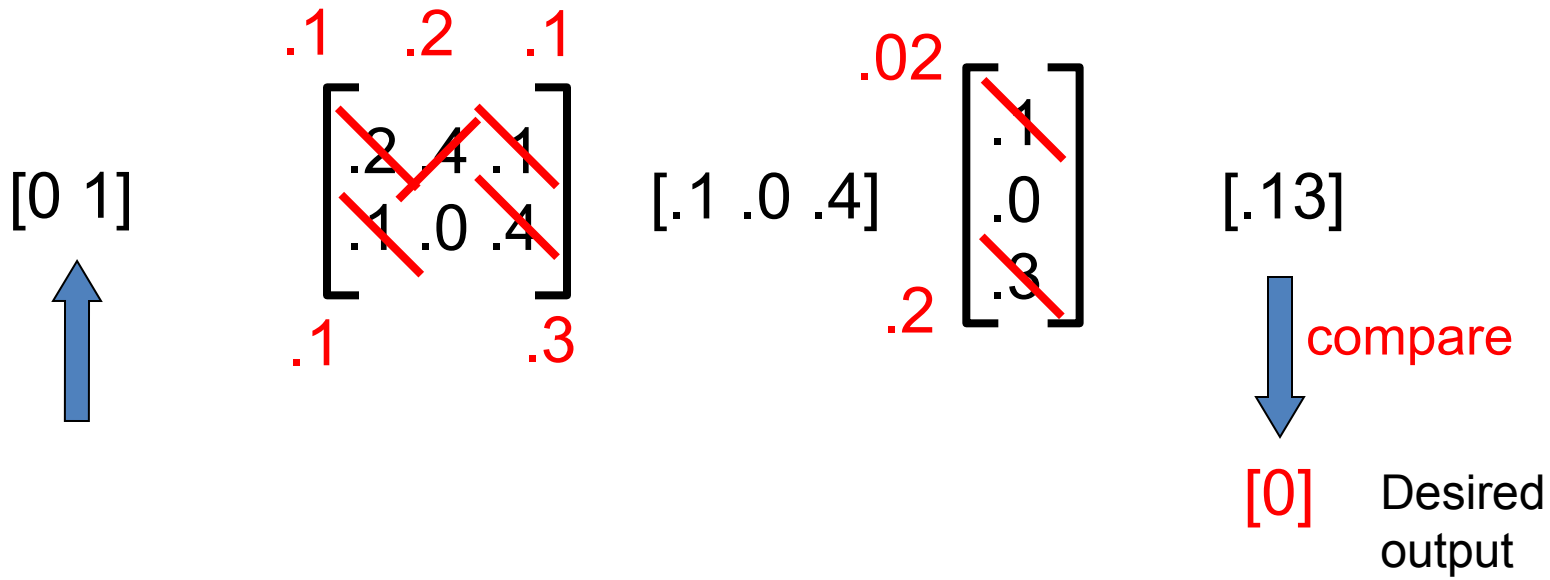
Interm. Vector

Weight Matrix2

Output Vector

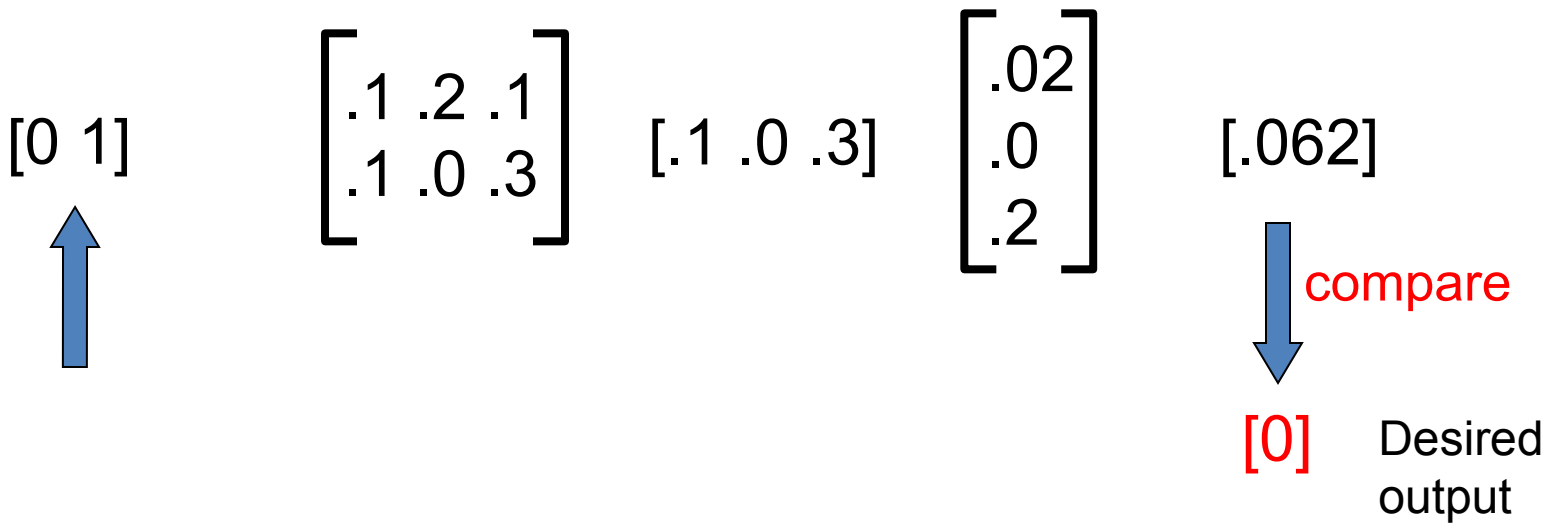
Back Propagation

$$\frac{1}{1 + e^{-x}}$$ Activation Function



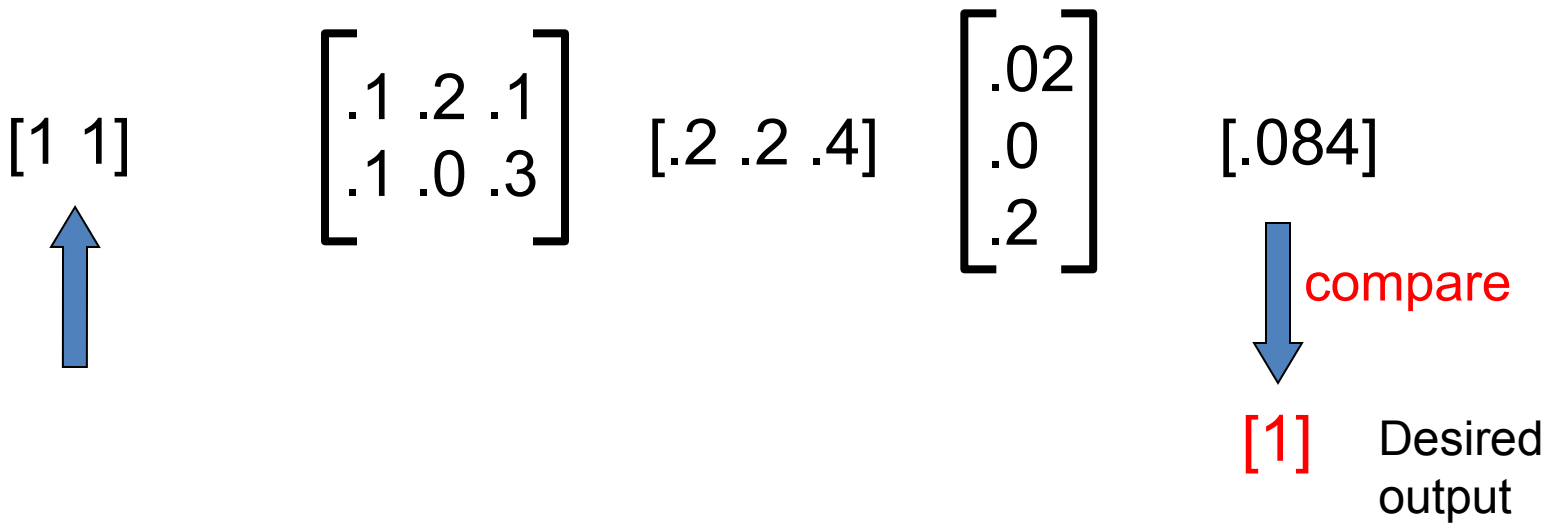
Calculate New Output

$$\frac{1}{1 + e^{-x}}$$ Activation Function



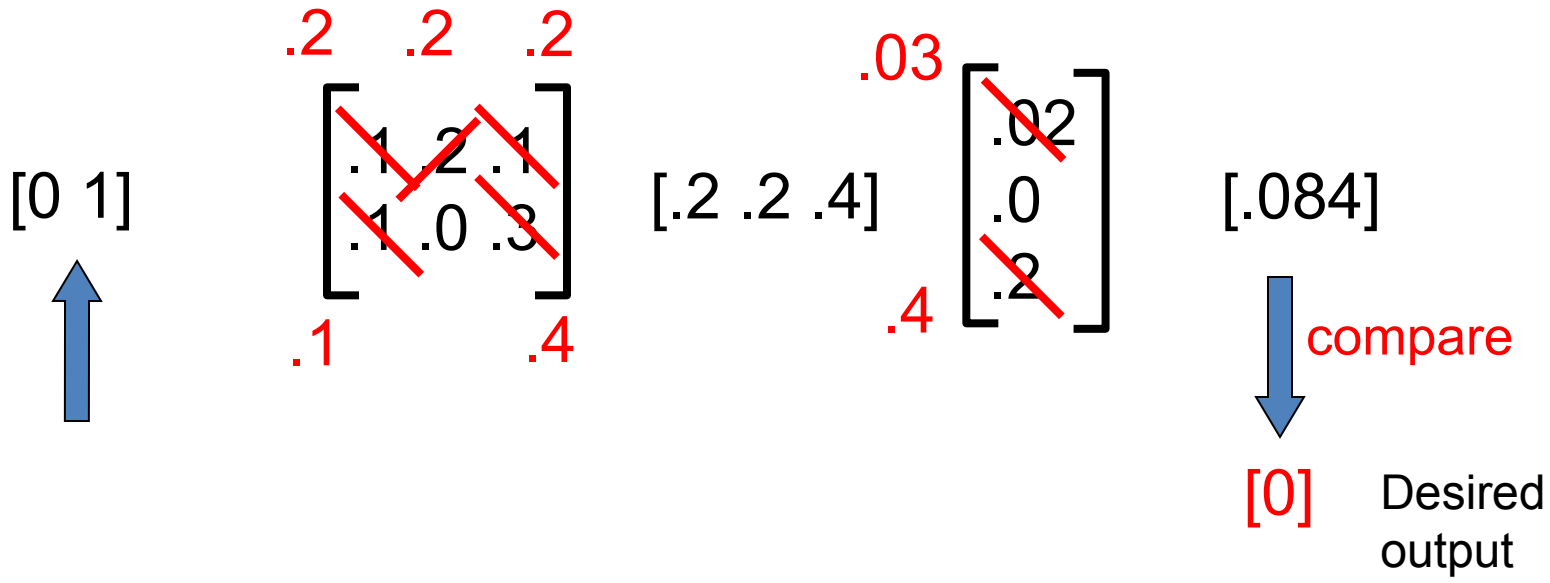
Train on Second Input Vector

$$\frac{1}{1 + e^{-x}}$$ Activation Function



Back Propagation

$$\frac{1}{1 + e^{-x}}$$ Activation Function

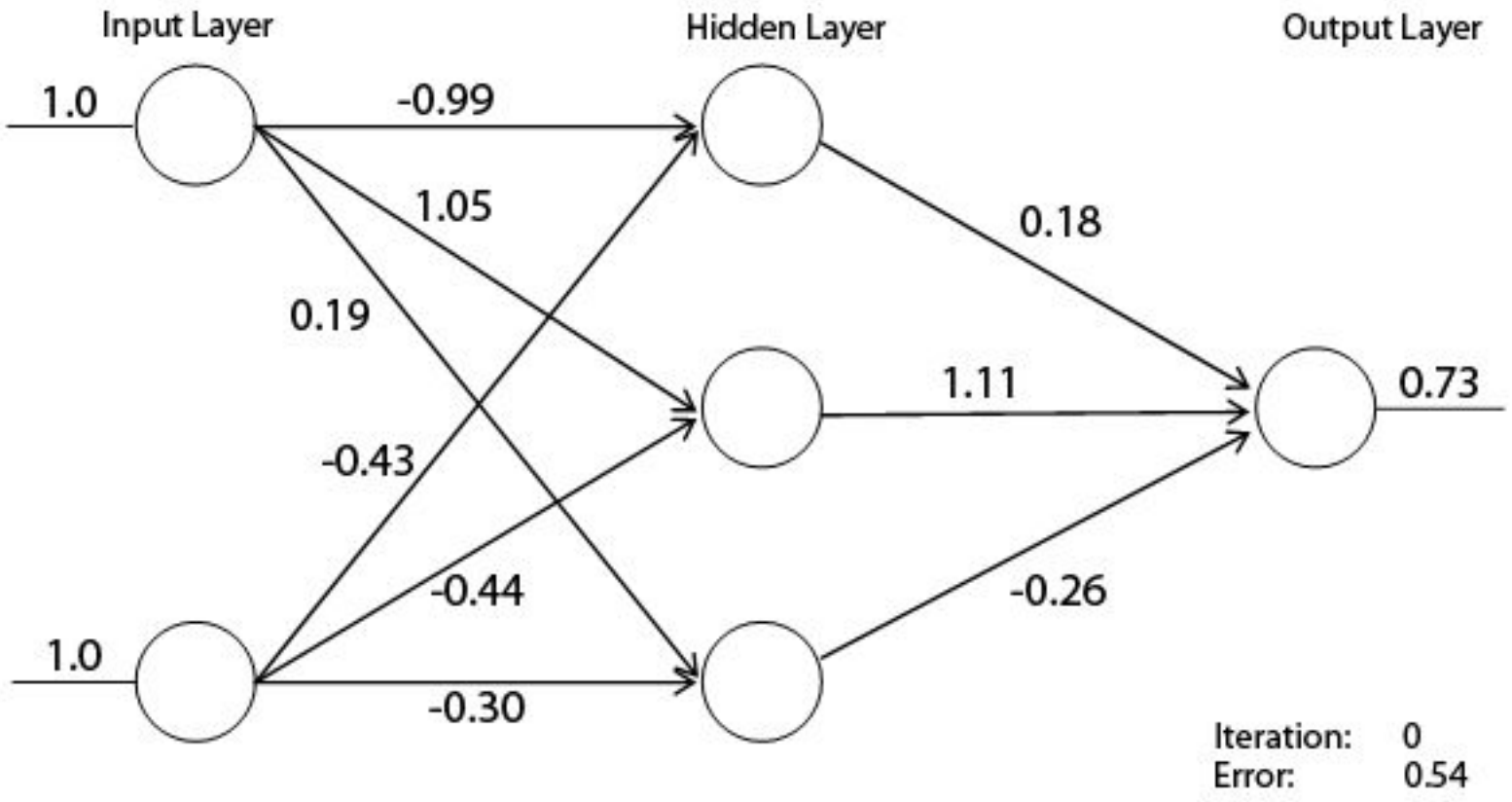


After Many Iterations….

$$\begin{bmatrix} .14 & .21 & .19 \\ .13 & .03 & .38 \end{bmatrix} \qquad \begin{bmatrix} .03 \\ .01 \\ .42 \end{bmatrix}$$

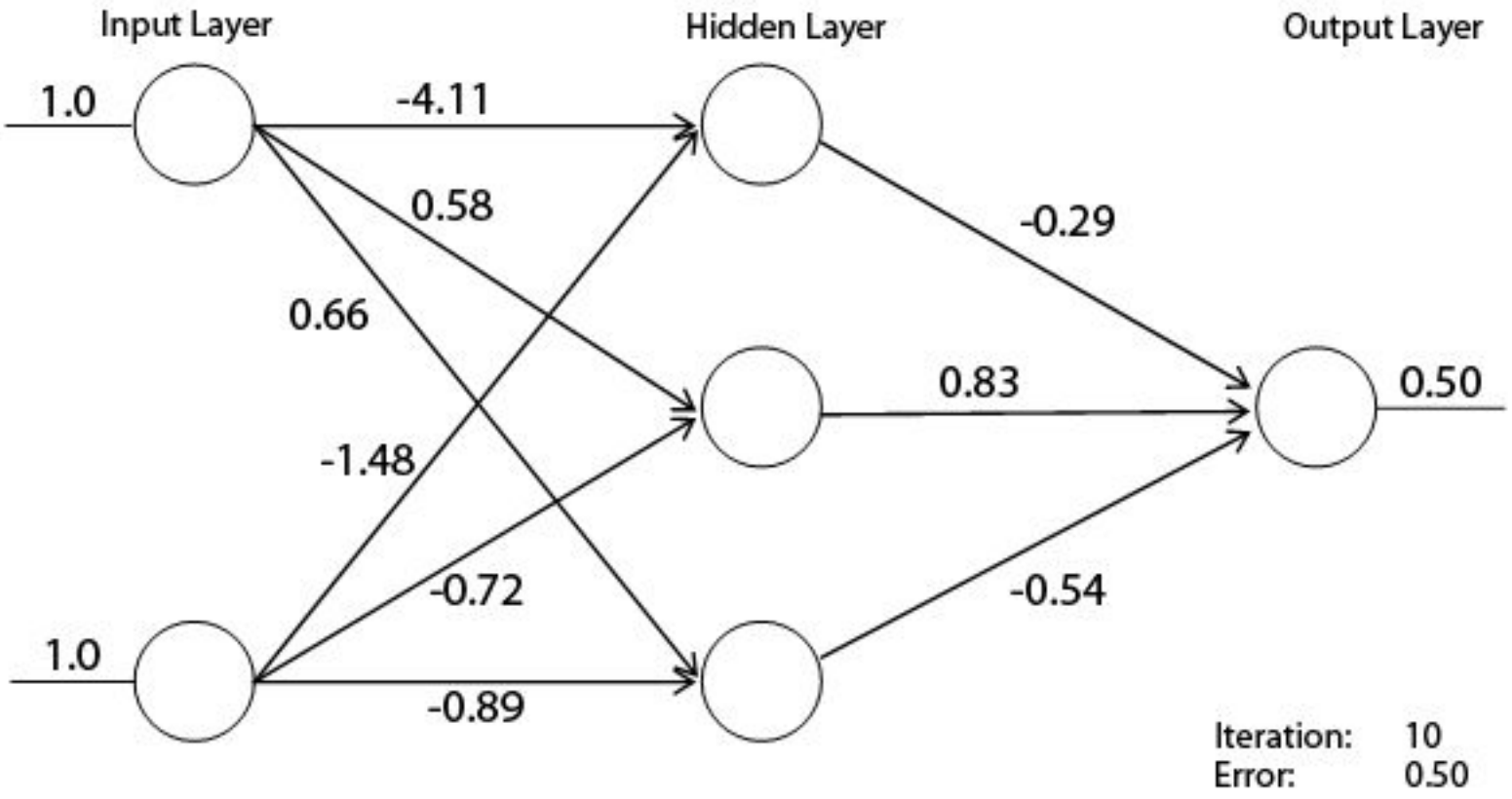
Two “Generalized” Weight Matrices

Neural Network Simulation

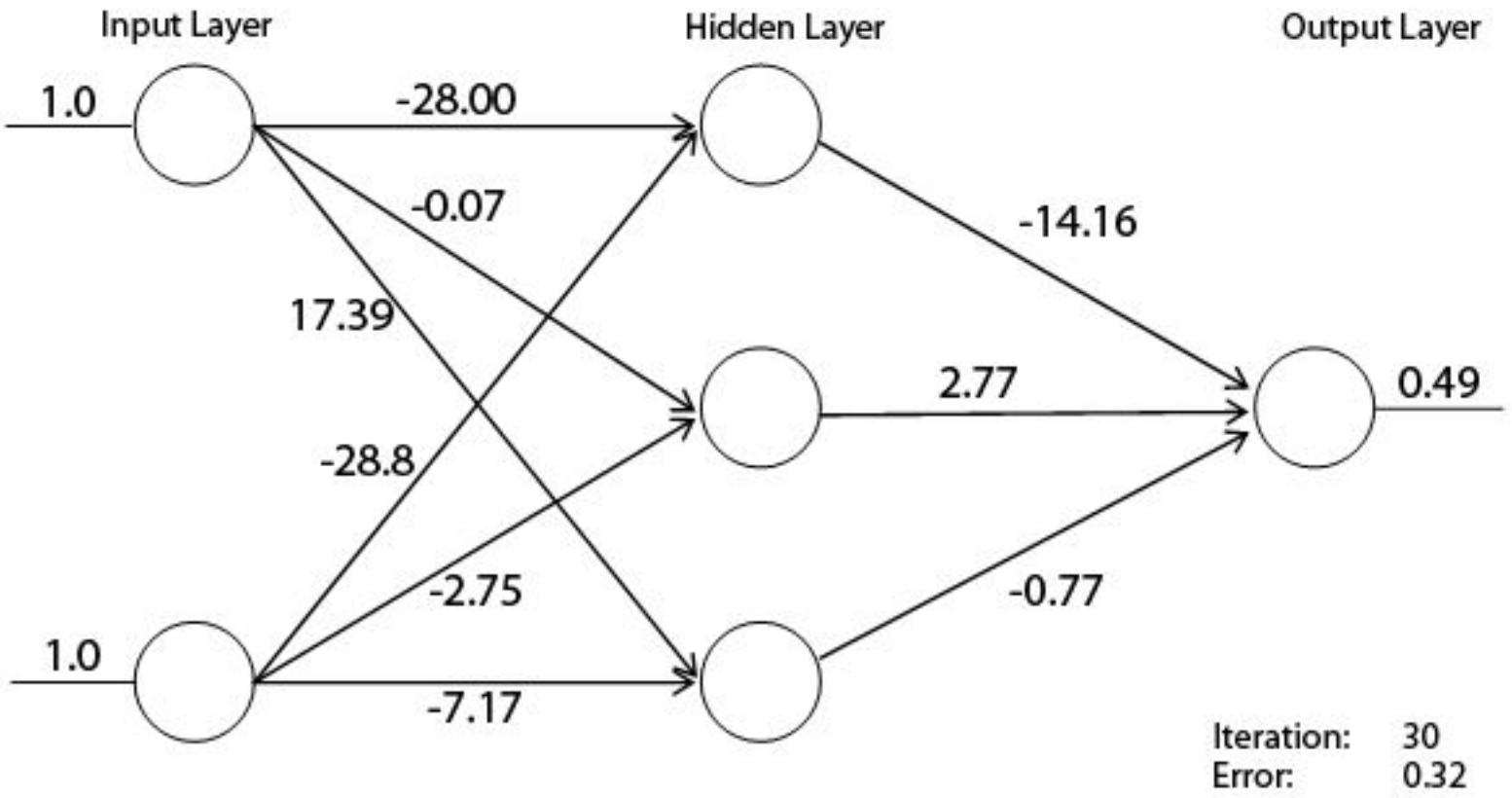


1 x 2 array 2 x 3 array 3 x 1 array 1 output

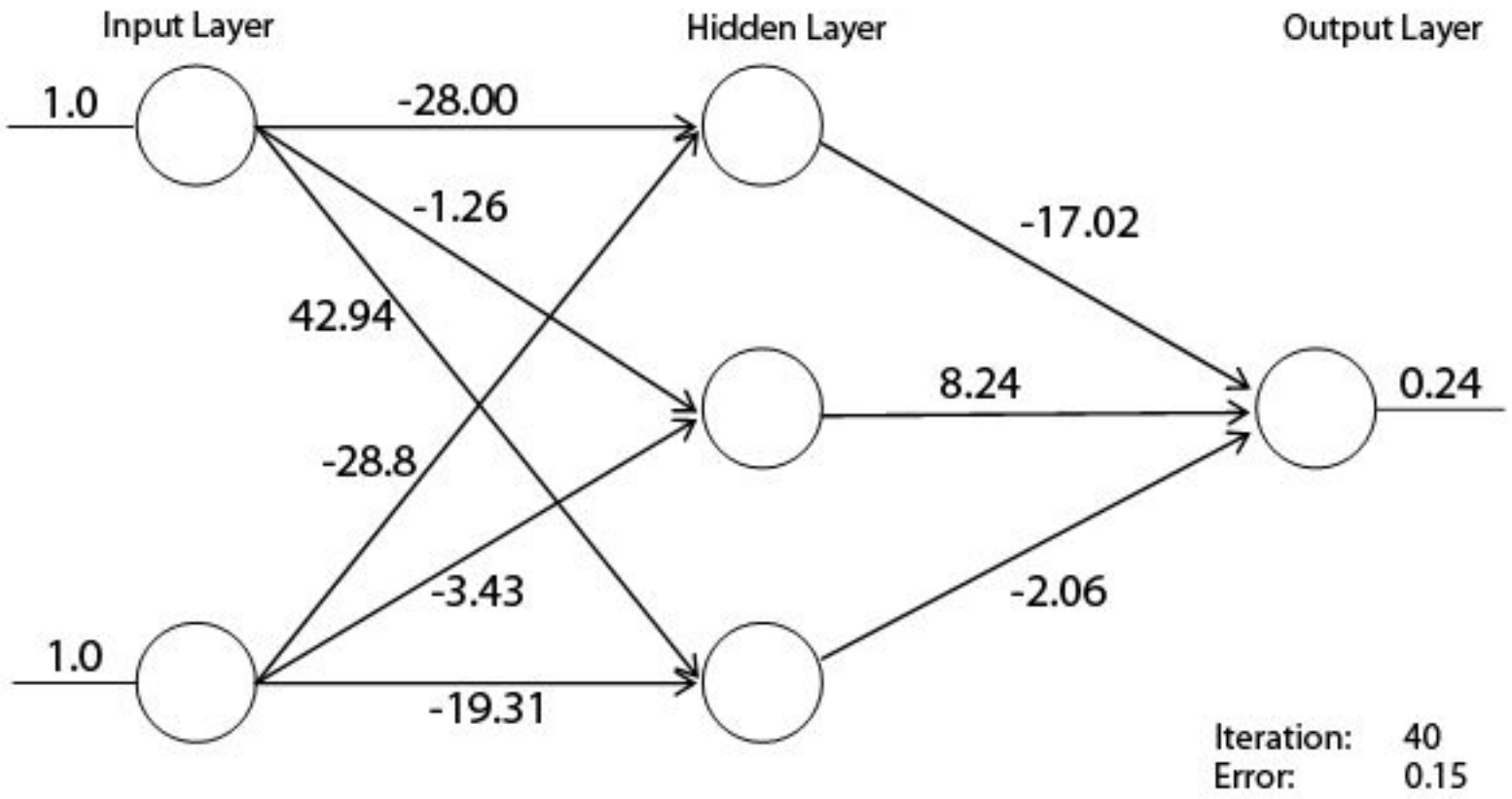
Neural Network Simulation



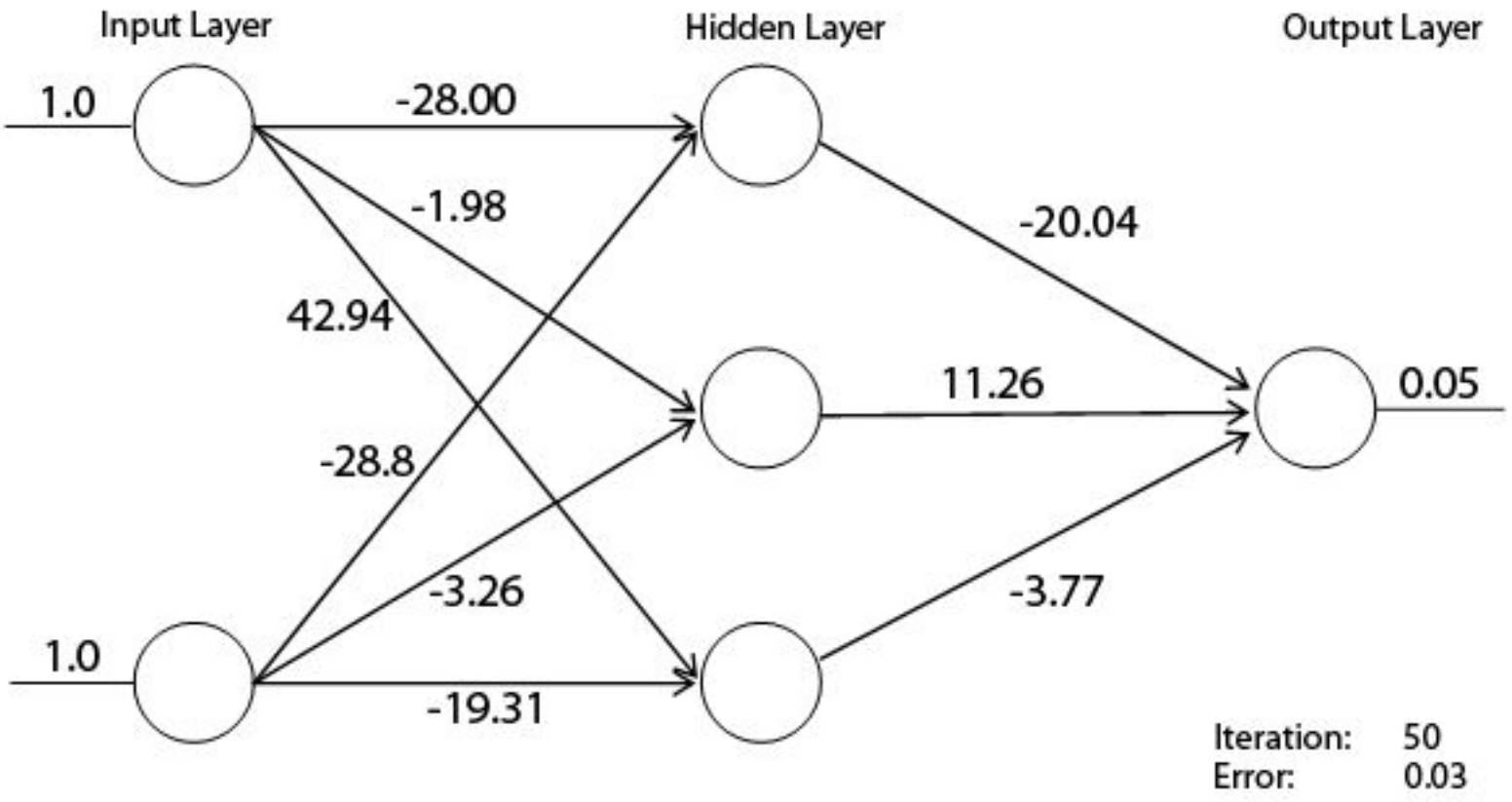
Neural Network Simulation



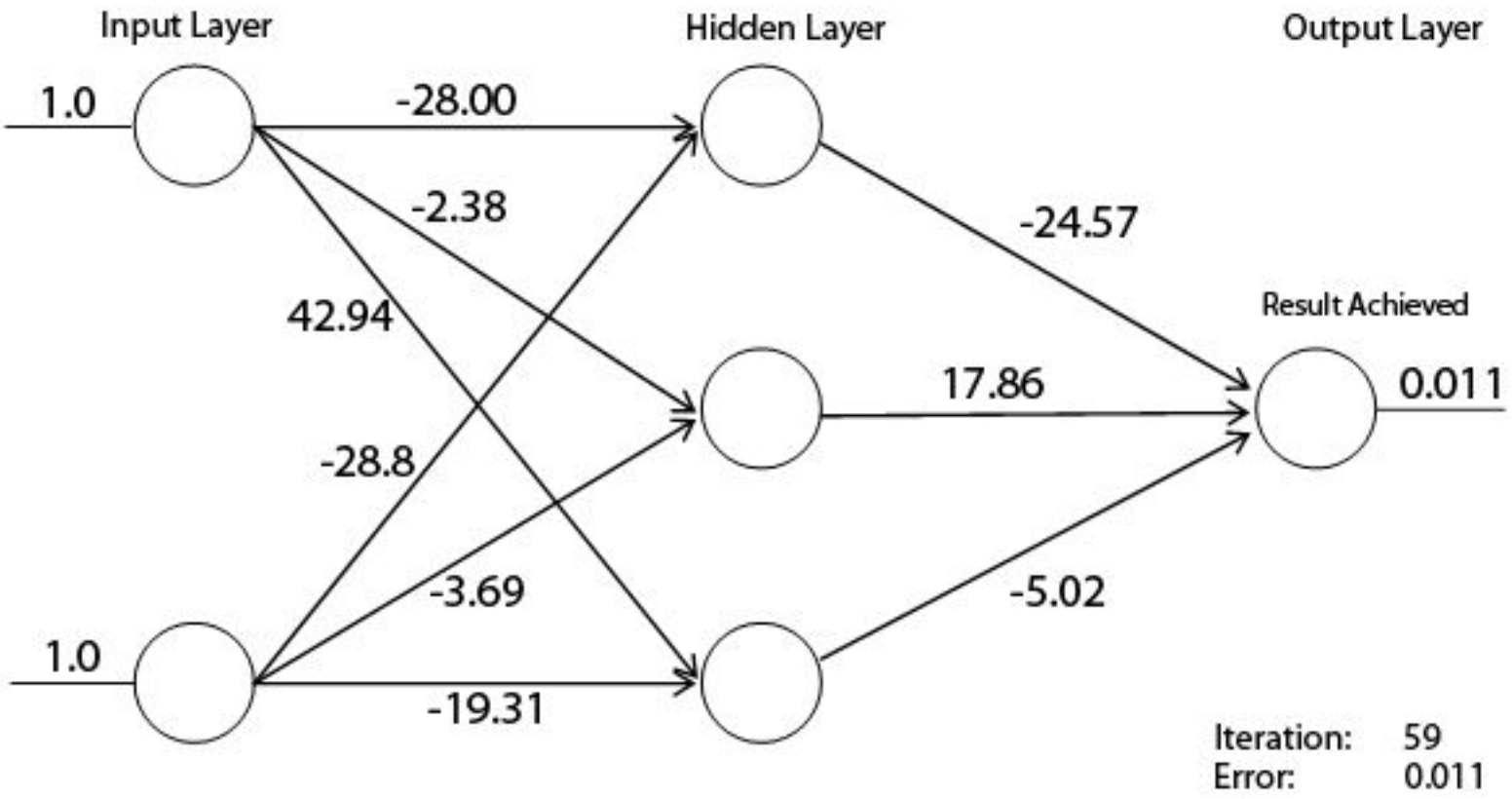
Neural Network Simulation



Neural Network Simulation

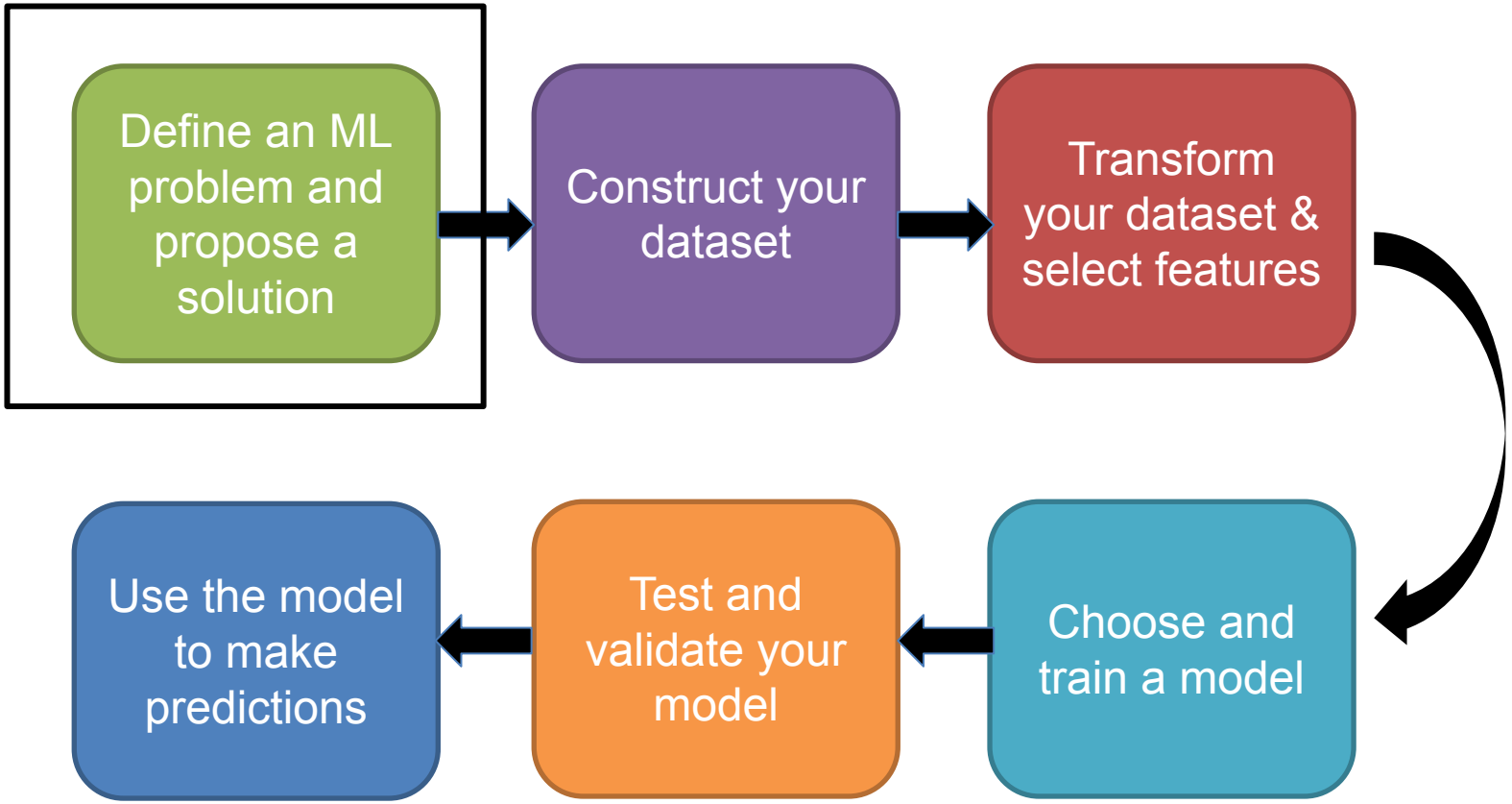


Neural Network Simulation



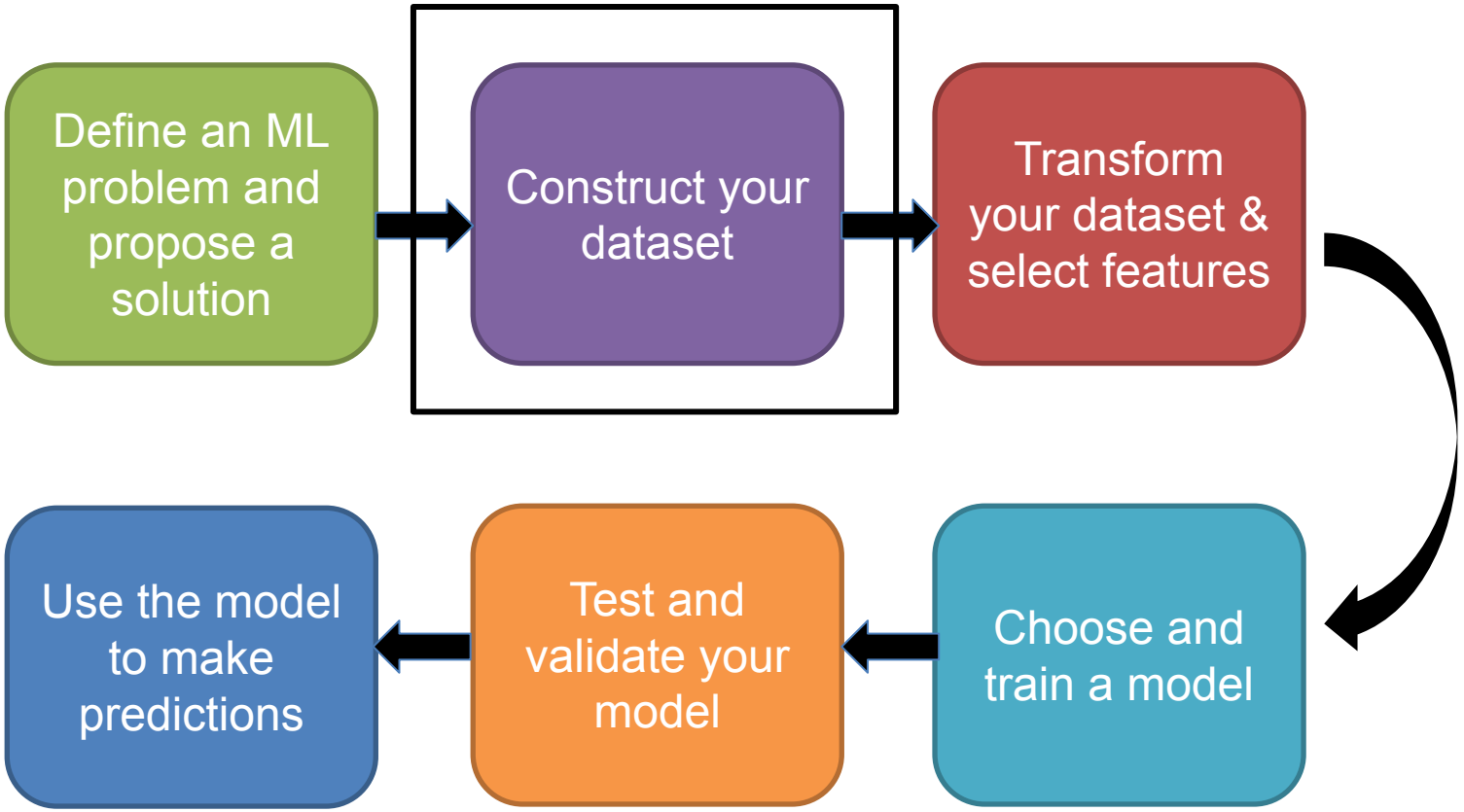
Let’s Try A Real Example

Machine Learning Workflow



How do I classify Iris flowers in my area based on their floral dimensions?

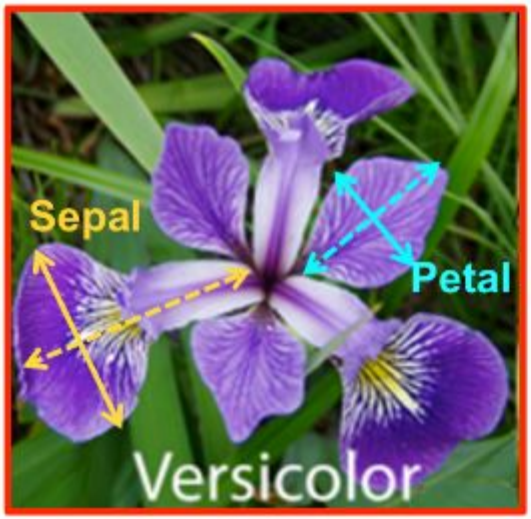
Machine Learning Workflow



Find a good training/testing data set

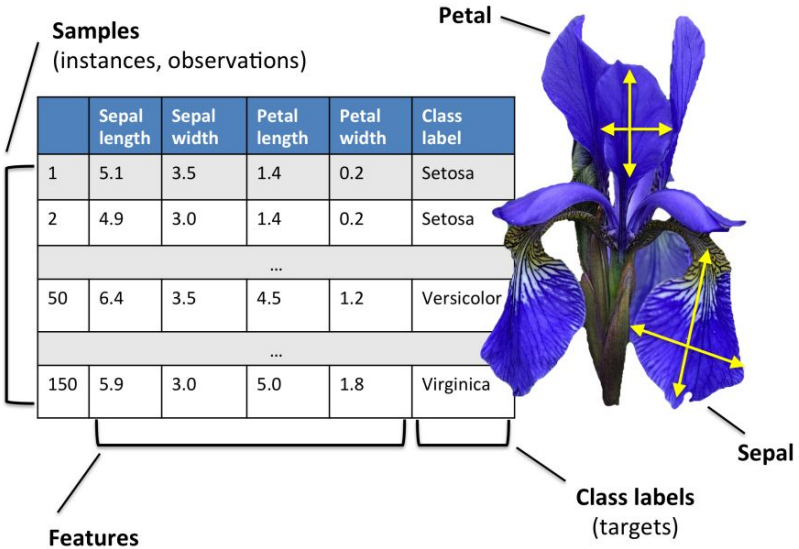
Iris Flower Data Set

- Iris flower data set introduced in 1936 by Ronald Fisher (Statistician) to demonstrate linear discriminant analysis
- Consists of 50 samples each of the Iris species: *Iris setosa, Iris virginica, Iris versicolor*
- Four features measured from each sample, length and width of sepals and petals in cm
- Different iris species can be differentiated by their petal/sepal dimensions



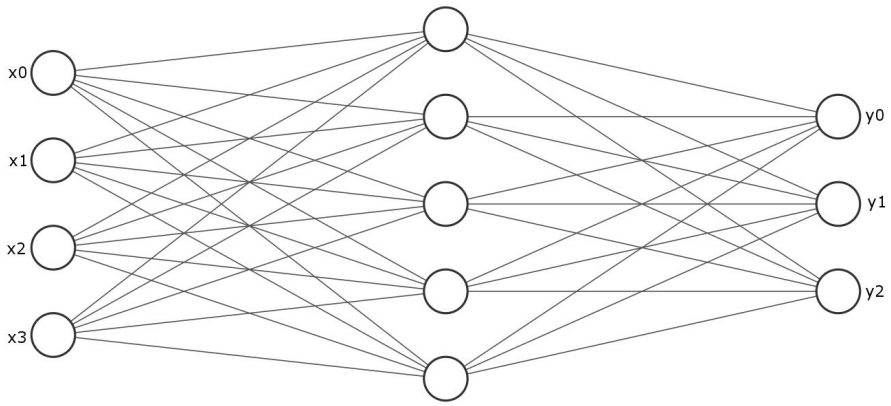
Iris Flower Neural Network

- **We will construct a neural network that will take as input these four features, and predict which category the flower belongs to**
- **The neural network will be trained on a subset of the 150 samples (training set), and will be tested on another subset (test set)**



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

https://sebastianraschka.com/Articles/2015_pca_in_3_steps.html



Format Data/Select Features

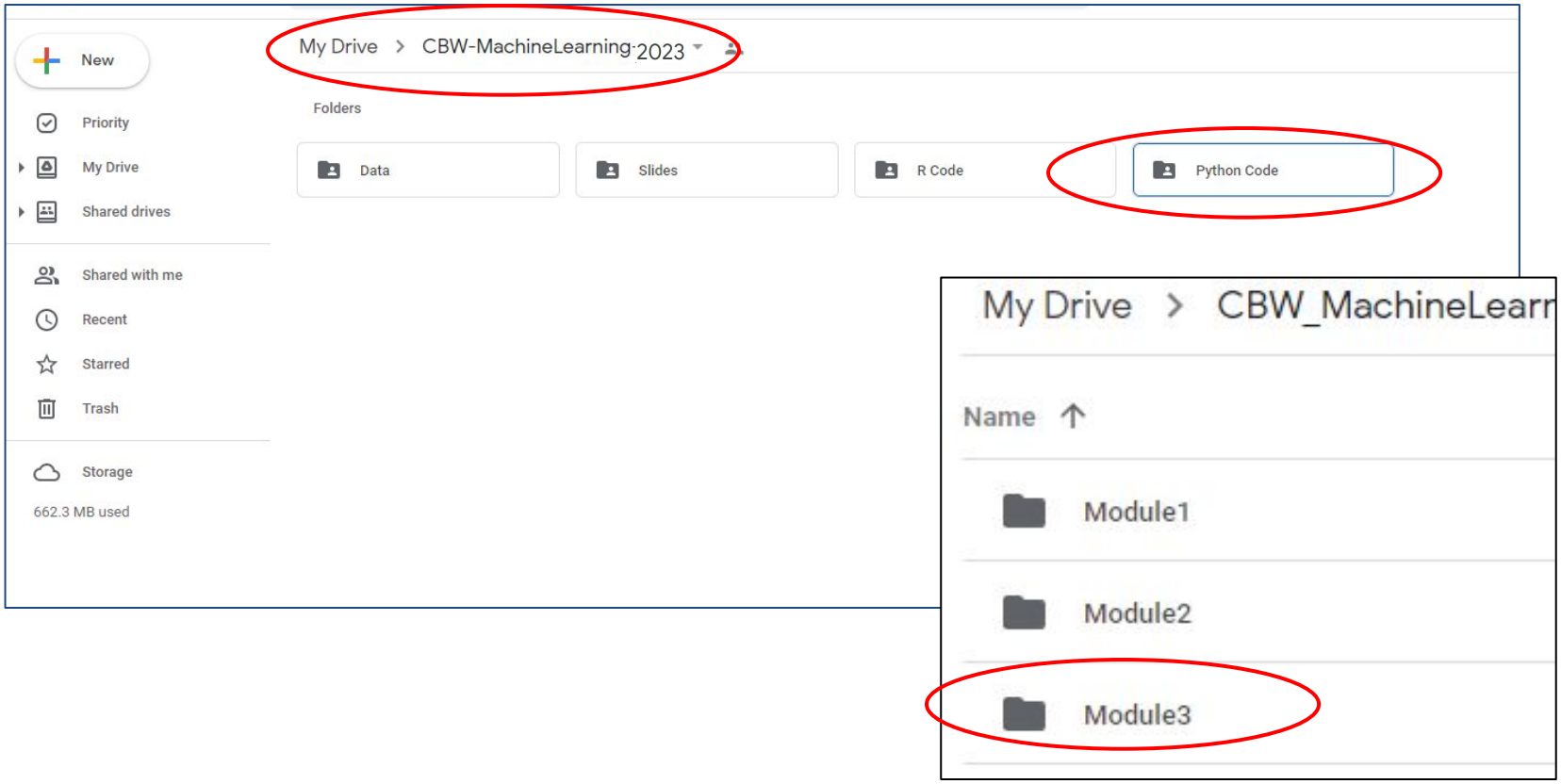
Sepal_Length	Sepal_Width	Petal_Length	Petal_Width	Species
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
7	3.2	4.7	1.4	Iris-versicolor
6.4	3.2	4.5	1.5	Iris-versicolor
6.9	3.1	4.9	1.5	Iris-versicolor
5.5	2.3	4	1.3	Iris-versicolor
6.5	2.8	4.6	1.5	Iris-versicolor
6.3	3.3	6	2.5	Iris-virginica
5.8	2.7	5.1	1.9	Iris-virginica
7.1	3	5.9	2.1	Iris-virginica
6.3	2.9	5.6	1.8	Iris-virginica
6.5	3	5.8	2.2	Iris-virginica

Let's Try Programming an Artificial Neural Network to Classify Iris Flowers

IrisANN.ipynb

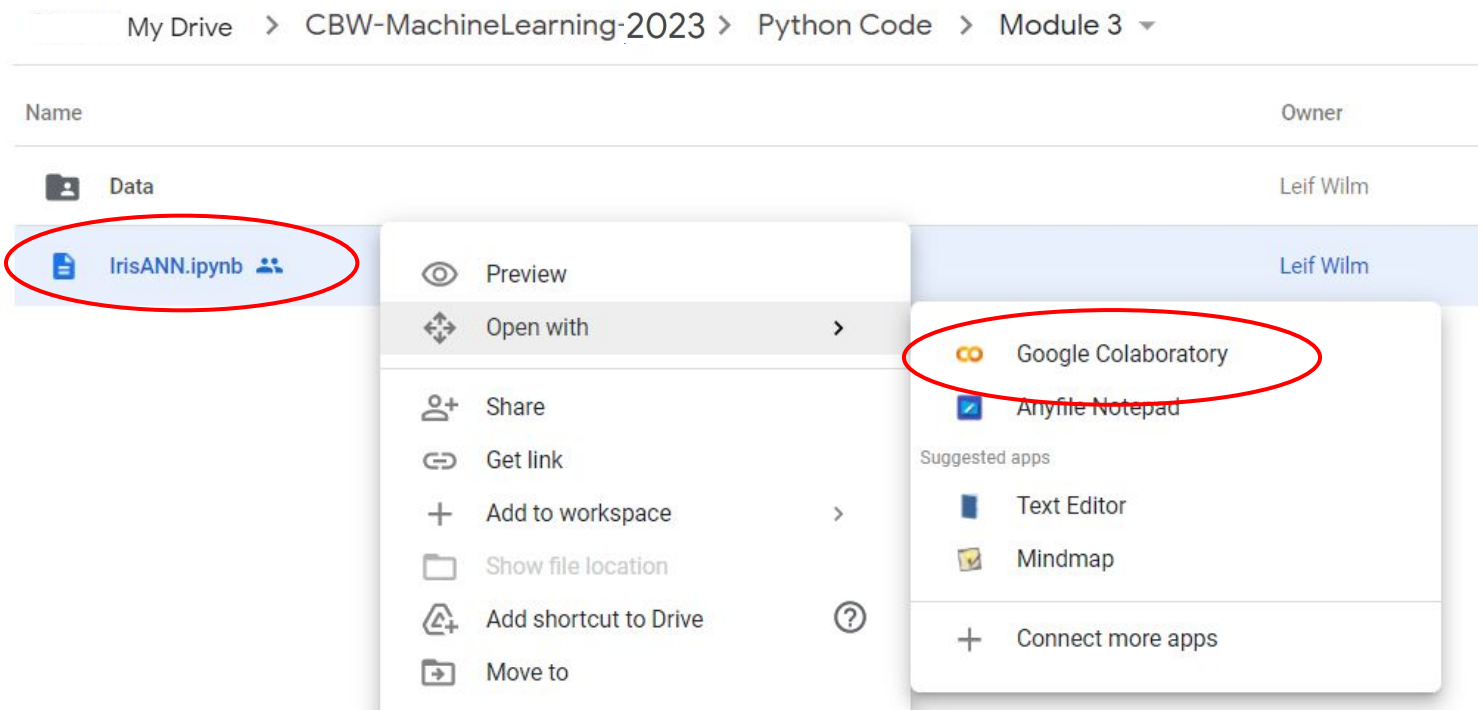
Open the Colab File

- **Find the Module 3 folder under Python Code in folder CBW-MachineLearning-2023 in your Google Drive**



Open the Colab File *cont...*

- **Right click on 'IrisANN.ipynb' and select open with Google Colaboratory**



General Algorithm

- Read data
- Check data
- Create training/testing data sets
- Create a one hot encoding function (to_one_hot)
- Create a normalization function (normalize)
- Normalize the data
- Perform label encoding and one hot encoding
- Define activation function (sigmoid or softmax)
- Initialize weights and biases
- Determine number of batches
- Perform forward propagation
- Calculate the errors
- Perform back propagation
- Update weights and biases

Import Functions for Python Math

- Numpy allows for mathematical operations and array handling
- Pandas is used to read data and for providing dataframe capabilities
- Seaborn and matplotlib are used for data visualization

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
```

Code for Reading Data In

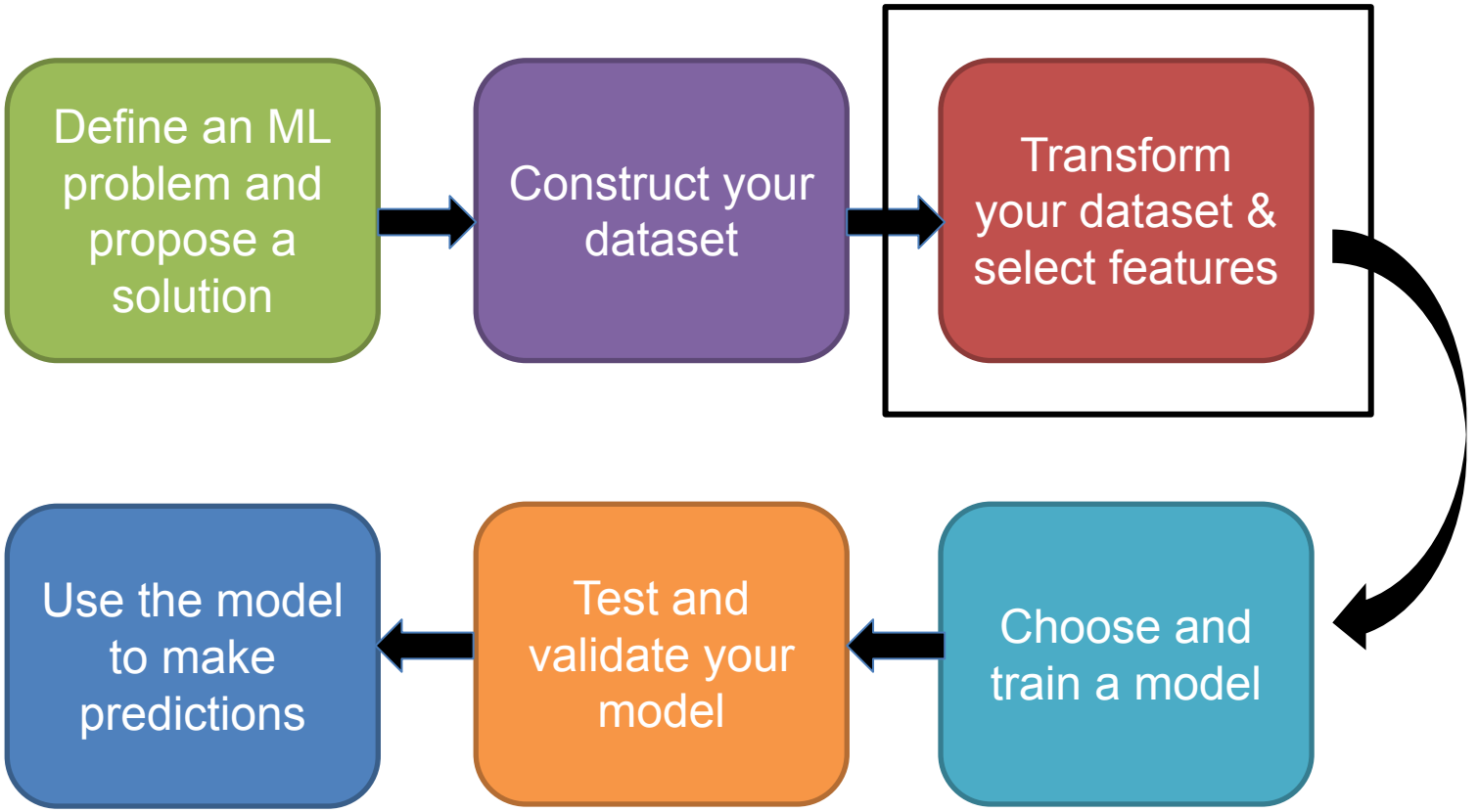
```
data = pd.read_csv('data1.csv')
#data.head()
data.loc[np.r_[0:3, 51:53, 101:103], :]
```

	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width	Species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
51	6.4	3.2	4.5	1.5	Iris-versicolor
52	6.9	3.1	4.9	1.5	Iris-versicolor
101	5.8	2.7	5.1	1.9	Iris-virginica
102	7.1	3.0	5.9	2.1	Iris-virginica

Data Check (Missing Values, Label Check)

```
#Check the dataset to make sure no data is missing and check
the class labels - need to define "verify_dataset" function
def verify_dataset(data):
#Use data_found as a dummy variable to determine if to print
missing value information
  data_found = 1
  for each_column in data.columns:
    if data[each_column].isnull().any():
            print("Data missing in Column " + each_column)
            data_found = 0
            quit()
    if data_found == 1:
        print("Dataset is complete. No missing value")
    return
#Call verify_dataset and check data
verify_dataset(data)
```

Machine Learning Workflow



Create a Training (70%) & Testing Set (30%)

- **The data set of 150 flowers is divided into 2/3 for training and 1/3 for testing (3-fold cross validation)**

```
##Splitting The Database in training and testing
def split_dataset_test_train(data):
    training_data = data.iloc[:int(0.7 * len(data))].reset_
    index(drop=True)
##Determine the integer location (iloc) from beginning of
array (:) to 0.7*150 and do a "cleanup" with a reset call
    testing_data = data.iloc[int(0.7 * len(data)):].reset_
    index(drop=True)
##Determine the integer location (iloc) from 0.7*150 to end
of array (: ) and do a "cleanup" with a reset call
    return [training_data, testing_data]

testtrain = split_dataset_test_train(data)
print(testtrain)
```

Transforming Data by One-Hot Encoding

- ANNs require input data to be binary (one-hot) encoded so that the input can be properly manipulated by the matrix calculations
- Need to change categorical or nominal variables to 1s and 0s

	target
1	Iris-setosa
2	Iris-virginica
3	Iris-setosa
4	Iris-versicolor
5	Iris-virginica

Apply One-hot Encoding →

	Iris-setosa	Iris-versicolor	Iris-virginica
1	1	0	0
2	0	0	1
3	1	0	0
4	0	1	0
5	0	0	1

Create a One-Hot Encoding Function

```
#This function accepts an array of categorical variables and
returns the one hot encoding
def to_one_hot(Y):
#Calculate number of columns
    n_col = np.amax(Y) + 1
#Create an array of size (len(Y) x n_col) and fill it with
0s
    binarized = np.zeros((len(Y), n_col))
    for i in range(len(Y)):
#Populate array with 1's at indicated array positions
        binarized[i, Y[i]] = 1.
    return binarized
```

Create a Normalization Function

- A key step in preparing the data is normalization, also known as feature scaling – we are using L2 normalization so that for each row in the data set, the sum of squares adds up to one
- The numerical input data (flower measurements) exist in different ranges
- If this input data is not scaled, the model will assume that features with higher values are more important when predicting flower classes, which may not be the case

```
#Normalize array - linalg.norm is a function that
linearly normalizes, expand_dims  is a function that
expands the shape of an array, atleast_1d is a function
that converts inputs into arrays of at least 1D
def normalize(X, axis=-1, order=2):
    L2 = np.atleast_1d(np.linalg.norm(X, order, axis))
    L2[L2 == 0] = 1
    return X / np.expand_dims(L2, axis)
```

Separate the Dataset into Input and Output & Normalize

- The input data "x" is then extracted from the original dataset, and normalized by calling the "normalize" function

```
#Specifying the columns that belong to input data "x"
columns = ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm',
'PetalWidthCm']
x = pd.DataFrame(iris, columns=columns)
#Calling the normalize function
x = normalize(x.as_matrix())
#Calls normalize function on the 4 x 150 iris data array
```

Encode the Categorical Variables

- **Before one hot encoding the output (categorical) variables, the labels are first replaced by 0,1,2 for *setosa, virginica*, and *versicolor* respectively**
- **This is known as label encoding**

```
#Replace the species with 0, 1, or 2 as appropriate
iris['Species'].replace(['Iris-setosa',
'Iris-virginica', 'Iris-versicolor'], [0, 1, 2],
   inplace=True)
```

Encoding the Categorical Variables *cont...*

- **Next, the y values (flower species) are extracted from the iris dataset and then one-hot encoded by calling the "to_one_hot" function**

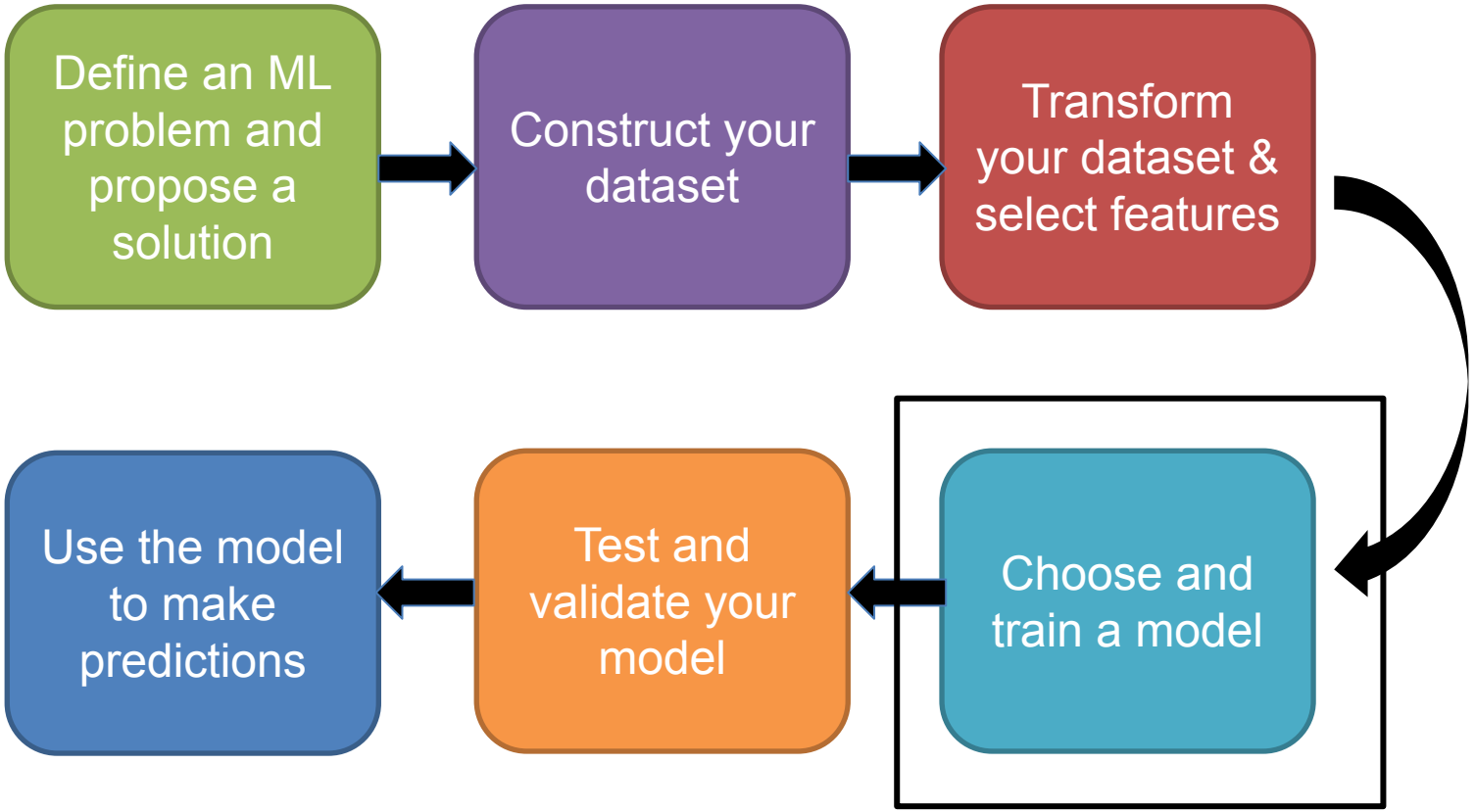
```
#Get Output, flatten and encode to one-hot
columns = ['Species']
y = pd.DataFrame(iris, columns=columns)
#Convert y to a Numpy readable array
y = y.as_matrix()
#Fix format of array so dim=2 array becomes dim=1
y = y.flatten()
#Call one-hot conversion function for species
y = to_one_hot(y)
```

Feature Selection

- Features can be turned on and off to determine their impact on the performance of the model
- This will be used in exercises later

```
'''Change the values below'''
sepal_length = True
sepal_width = True
petal_length = True
petal_width = True
'''Change the values above'''

feature_list = [sepal_length,sepal_width,petal_length,petal_width]
```

Machine Learning Workflow



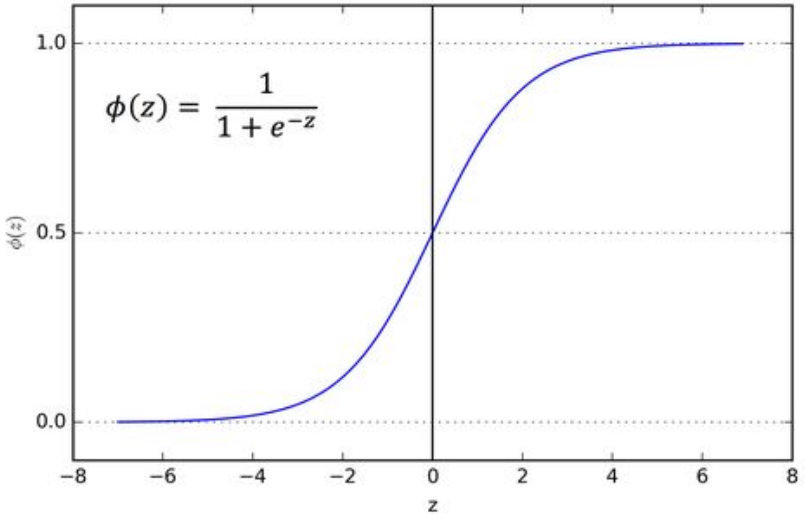
We have chosen an Artificial Neural Network Model

Define Activation Function

- The sigmoid function is used for the activation function, as it outputs values between 0 and 1
- This is useful because this model predicts probabilities as an output
- Additionally, this function is differentiable
- This allows for gradient descent optimization, which is used to minimize the error between the predicted and actual output (during training)

```
#sigmoid and its derivative
def sigmoid(x):
    return 1/(1+np.exp(-x))

def sigmoid_deriv(x):
    return sigmoid(x)*(1 -
sigmoid(x))
```

Sigmoid function used for layer 1



Some Math Reminders

Sigmoid Function

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

SoftMax Function

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

$$\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \qquad \frac{d}{dx}e^x = e^x \\
&= \frac{d}{dx}\left(1+e^{-x}\right)^{-1} \\
&= -(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x})-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) \\
&= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x) \cdot (1-\sigma(x))
\end{aligned}$$

$$\vec{z} = \begin{bmatrix} 8 \\ 5 \\ 0 \end{bmatrix} \qquad \begin{aligned} e^{z_1} &= e^8 = 2981.0 \\ e^{z_2} &= e^5 = 148.4 \\ e^{z_3} &= e^0 = 1.0 \end{aligned}$$

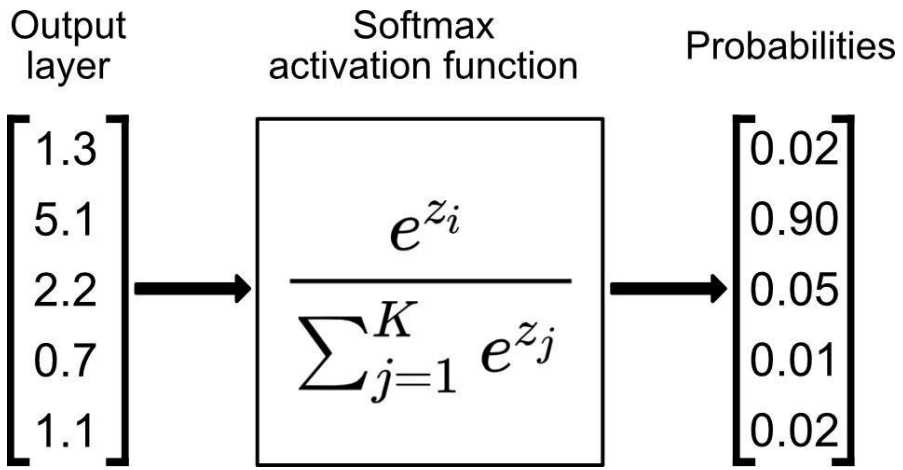
$$\begin{aligned}
\sum_{j=1}^{K} e^{z_j} &= e^{z_1} + e^{z_2} + e^{z_3} \\
&= 2981.0 + 148.4 + 1.0 = 3130.4
\end{aligned}$$

$$\begin{aligned}
\sigma(\vec{z})_1 &= \frac{2981.0}{3130.4} = 0.9523 \\
\sigma(\vec{z})_2 &= \frac{148.4}{3130.4} = 0.0474 \\
\sigma(\vec{z})_3 &= \frac{1.0}{3130.4} = 0.0003
\end{aligned}$$

Define Activation Function

- For multiclass problems, the output layer often uses the softmax activation function
- This function has the benefit of outputting values that sum exactly to 1

```
def user_softmax(A):
    expA = np.exp(A)
    return expA / expA.sum(axis=1,
    keepdims=True)
```



Softmax function used for layer 2

Initialize Weights and Biases

- Before training the model, the weights and biases are initialized as random values between 0 and 1
  - w0 is the set of weights for connections between the input and hidden layer
  - w1 is the set of weights for connections between the hidden and output layer
- The biases bh and bo are for the hidden and output layers respectively

```
#random.random returns number from 0 to 1 while random.randn
#returns Gaussian random number from -1 to 1, with mean of 0

#Weight initialization
w0 = 2*np.random.random((X_train.shape[1], hidden_size)) - 1
w1 = 2*np.random.random((hidden_size, 3)) - 1

bh = np.random.randn(hidden_size)
bo = np.random.randn(3)
```

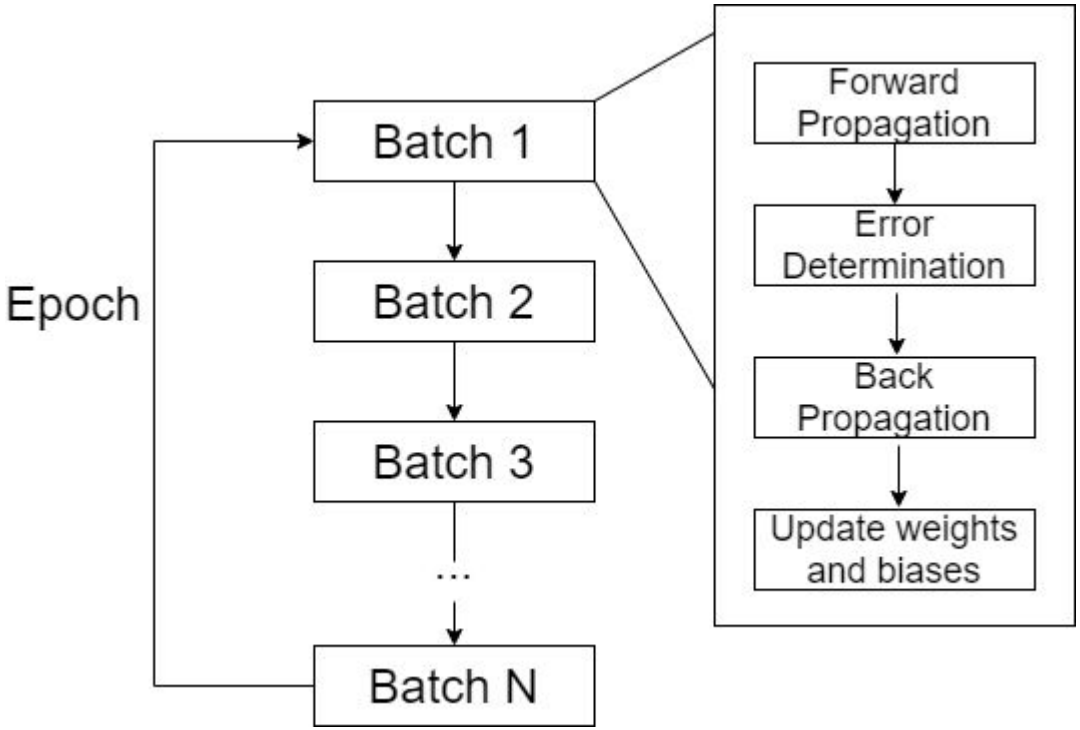
Determine Number of Batches

- The number of batches ‘num_batch’ is calculated from the user-determined batch size and training size
- The number of batches is corrected in case the training set size is not a multiple of the batch size

```
#Recall that % returns remainder (modulus)
#Recall that // returns largest divisor

  rem = len(X_train) % batch_size
  num_batch = len(X_train)//batch_size
  if rem != 0:
    num_batch += 1
```

The ANN Training Loop

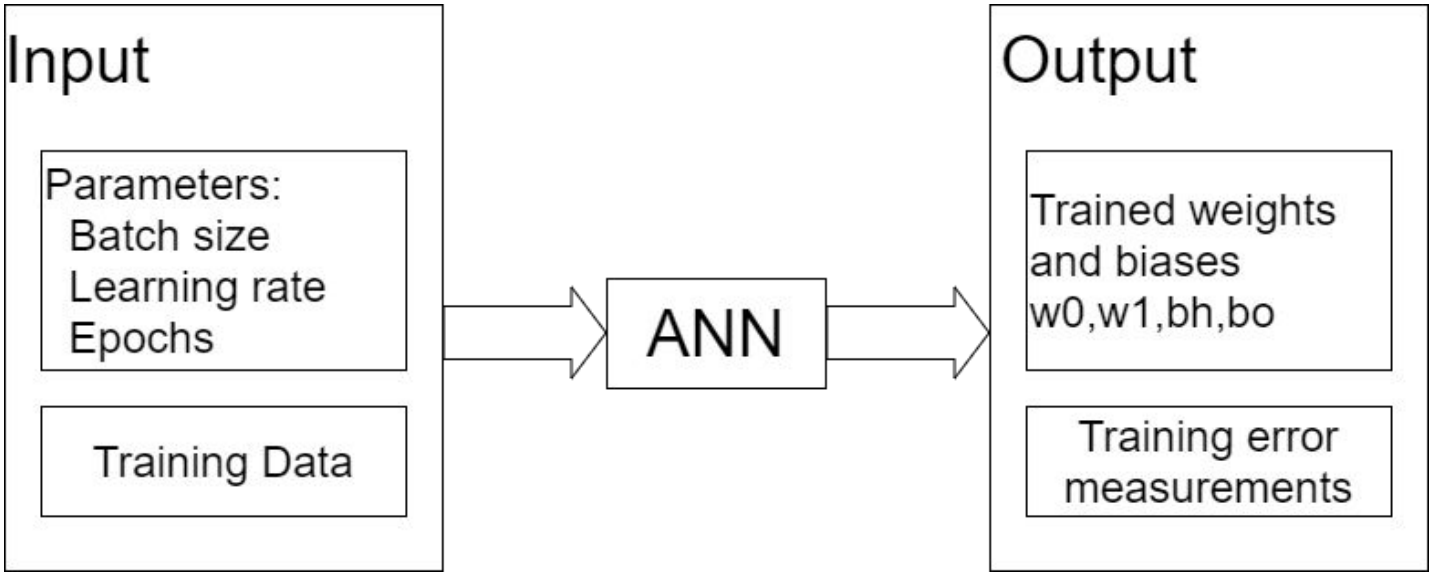
- Forward propagation, error determination, back propagation, and weight/bias updating are performed for every batch of input data
- This process is repeated for multiple epochs



ANN Input and Output

- The model takes learning rate, batch size, epochs, and the training set as input
- The model will output the weights and biases for the input and hidden layer, as well as training error measurements

```
def training(lr, batch_size, epochs): ⟶ return [w0,bh,w1,bo], error, errors
```



Forward Propagation

- The current batch of data is extracted from the total training set
- The input is propagated through the layers by multiplying by weight matrices, adding biases, and running through the activation functions

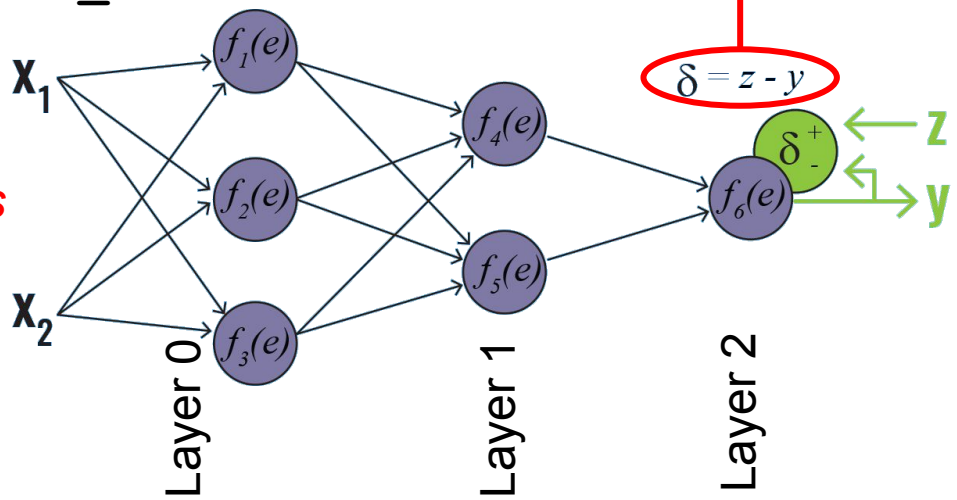
```
# Finding the current batch
batch_start = curr_batch * batch_size
batch_end = batch_start + batch_size
input_batch = X_train[batch_start:batch_end]
#First layer propagation This equation calculates dot
#product of input layer vector (batch size x 4) with weight
#matrix w0 (4 x hidden layer size)
zh = np.dot(input_batch, w0) + bh
layer1 = sigmoid(zh)
#Second layer propagation
zo = np.dot(layer1, w1) + bo
layer2 = user_softmax(zo)
```

Determine the Error

- The labels (true output) corresponding to the current batch of inputs is found
- These are compared to the forward-propagated output of the model 'layer2' (δ is derivative of the cost function)
- This subtraction represents the cost function which, through gradient descent, will be minimized

```
labels_batch = y_train[batch_start:batch_end]
layer2_error = layer2 - labels_batch
```

The "complicated" partial derivatives actually simplify to simple differences



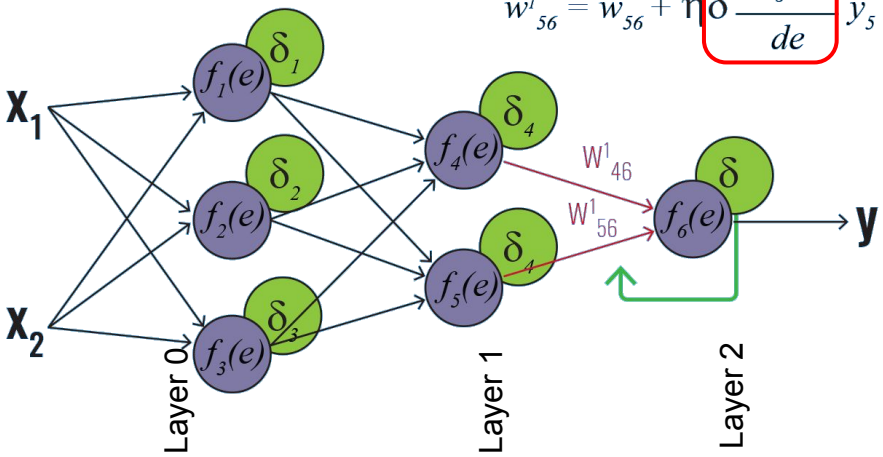
Backpropagation (Layer 2 □ 1)

- Using the gradient descent algorithm, the derivative of the cost with respect to weights for layer 2 is calculated
- This is called “layer2w_delta” as this will later be subtracted from the layer2 weights to find the new set of weights
- The change in layer2 bias is simply the layer2 error

```
layer2w_delta = np.dot(layer1.T, layer2_error)
layer2b_delta = layer2_error
```

$$w^1_{46} = w_{46} + \eta\delta \frac{df_6(e)}{de} y_4$$

$$w^1_{56} = w_{56} + \eta\delta \frac{df_6(e)}{de} y_5$$

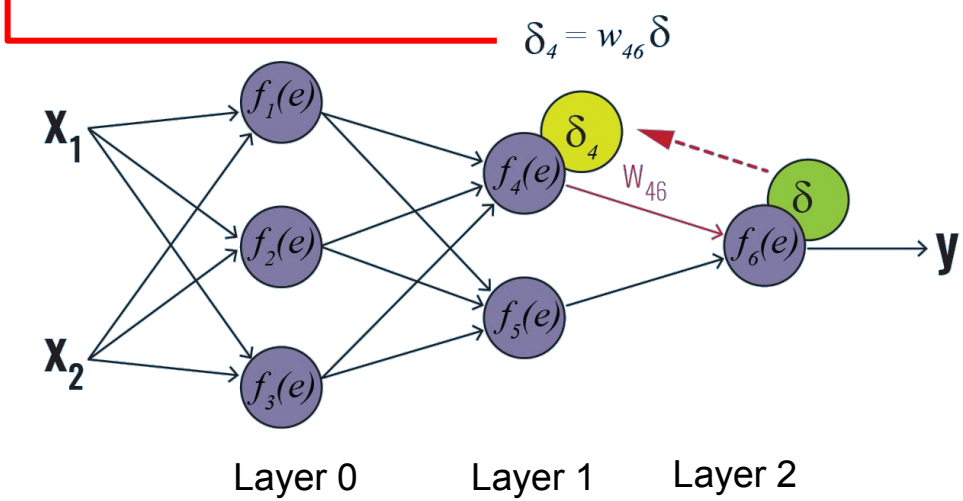


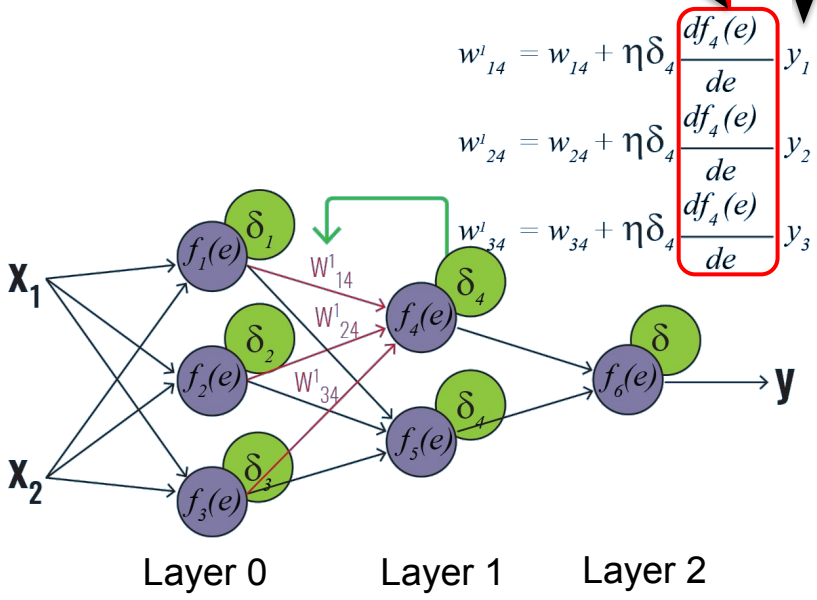
Backpropagation (Layer 1 □ 0)

- The cost function must be differentiated with respect to the hidden layer weights
- The hidden layer partial derivatives are first found

```
#Propogation of error to layer1
dcost_dah = np.dot(layer2_error , w1.T)
dah_dzh = sigmoid_deriv(zh)
```

ah = output of hidden layer (y here)
zh = weighted sum (e here)
dah/dzh = dy/de
dcost/dah = δ





Backpropagation (Layer 1 □ 0)

- **The partial derivatives are then multiplied to get 'layer1w_delta', the number used to change the layer1 weights**

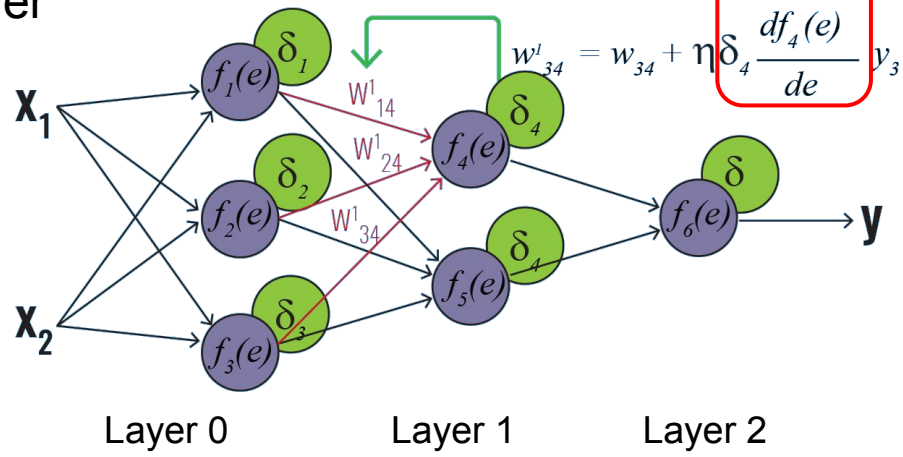
```
#Propogation of error to layer1

layer1_error = dah_dzh * dcost_dah
layer1w_delta = np.dot(input_batch.T, layer1_error)
layer1b_delta = layer1_error
```

Propagating from output layer to hidden layer

$$w^1_{14} = w_{14} + \eta\delta_4 \frac{df_4(e)}{de} y_1$$

$$w^1_{24} = w_{24} + \eta\delta_4 \frac{df_4(e)}{de} y_2$$

$$w^1_{34} = w_{34} + \eta\delta_4 \frac{df_4(e)}{de} y_3$$

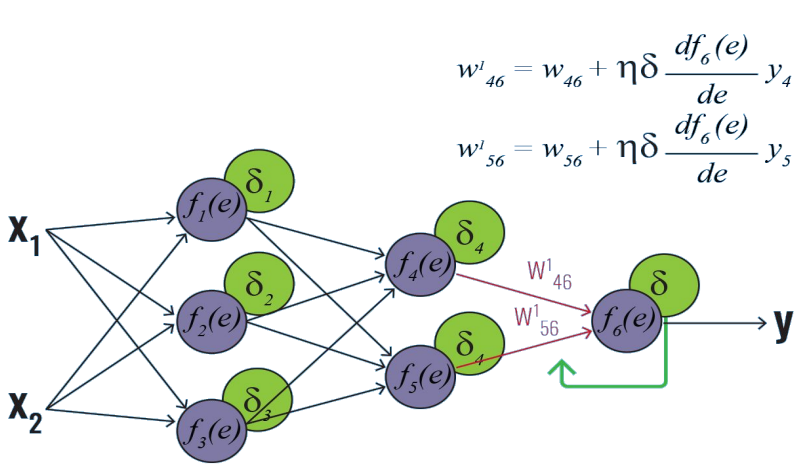


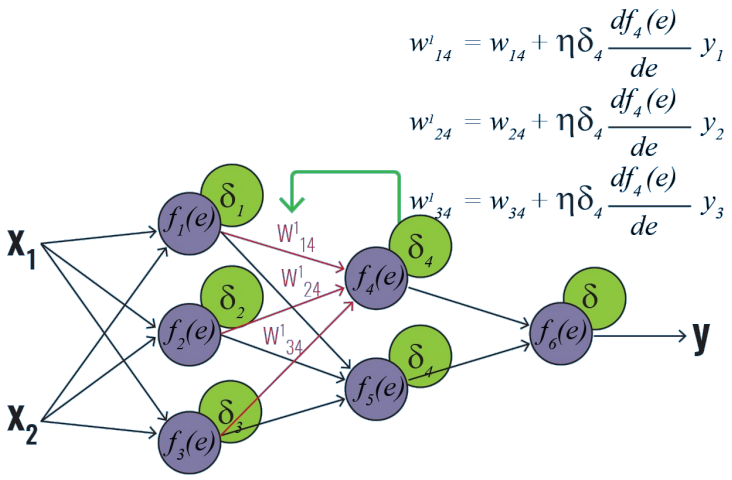
Update Weights

- **Using the products found earlier, the weight matrices can now be updated**
- **The respective delta matrices are multiplied by the learning rate and subtracted from the current weight values**

```
#Update weights with learning rate n
w0 -= lr * layer1w_delta
w1 -= lr * layer2w_delta
```

*wo = wo – lr*layer1w_delta*





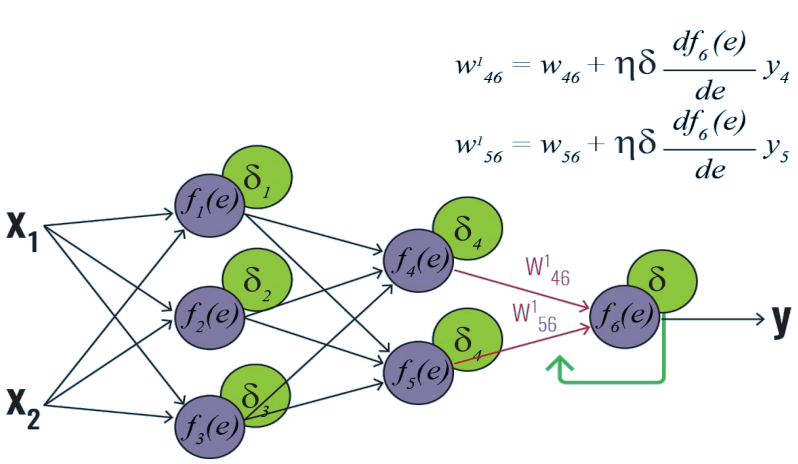
Update Biases

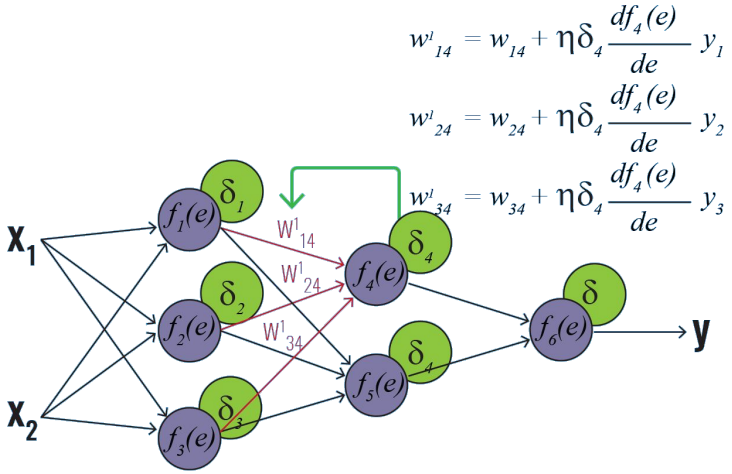
- **Similarly, the bias deltas are subtracted from the current bias values**

```
#Update biases with learning rate n
bh -= lr * layer1b_delta.sum(axis=0)
bo -= lr * layer2b_delta.sum(axis=0)
```

*bh = bh – lr*layer1b_delta*

Sums all elements in 2D array to put into 1D vector





Complete Training Process

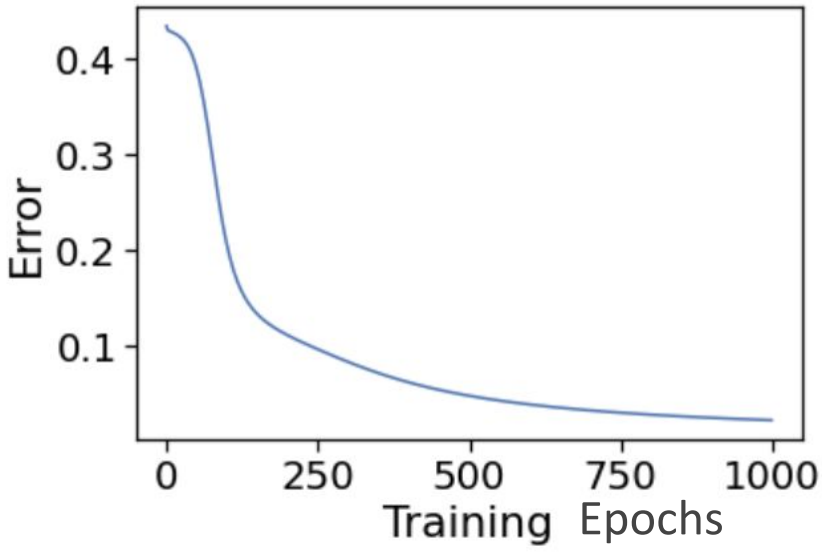
- **The process is repeated for a number of epochs**

```
for epoch in range(epochs):
    for curr_batch in range(num_batch):
        batch_start = curr_batch * batch_size    # Finding the current batch
        batch_end = batch_start + batch_size
        input_batch = X_train[batch_start:batch_end]

        zh = np.dot(input_batch, w0) + bh        # First layer propagation
        layer1 = sigmoid(zh)
        zo = np.dot(layer1, w1) + bo         # Second layer propagation
        layer2 = user_softmax(zo)

        labels_batch = y_train[batch_start:batch_end]
        layer2_error = layer2 - labels_batch           # Layer 2 error calculation
        layer2w_delta = np.dot(layer1.T, layer2_error) # Layer 2 derivative calculations
        layer2b_delta = layer2_error

        dcost_dah = np.dot(layer2_error , w1.T)        # Layer 1 derivative calculations
        dah_dzh = sigmoid_deriv(zh)
        layer1_error = dah_dzh * dcost_dah       # Layer 1 error calculation
        layer1w_delta = np.dot(input_batch.T, layer1_error)

        layer1b_delta = layer1_error

        w0 -= lr * layer1w_delta         # Weight and bias updating
        bh -= lr * layer1b_delta.sum(axis=0)

        w1 -= lr * layer2w_delta
        bo -= lr * layer2b_delta.sum(axis=0)
```

Training Output

- **In addition to outputting the trained parameters, the training function will also output all the layer2 errors found during training**
- **This results in one error measurement per epoch, which can then be plotted**

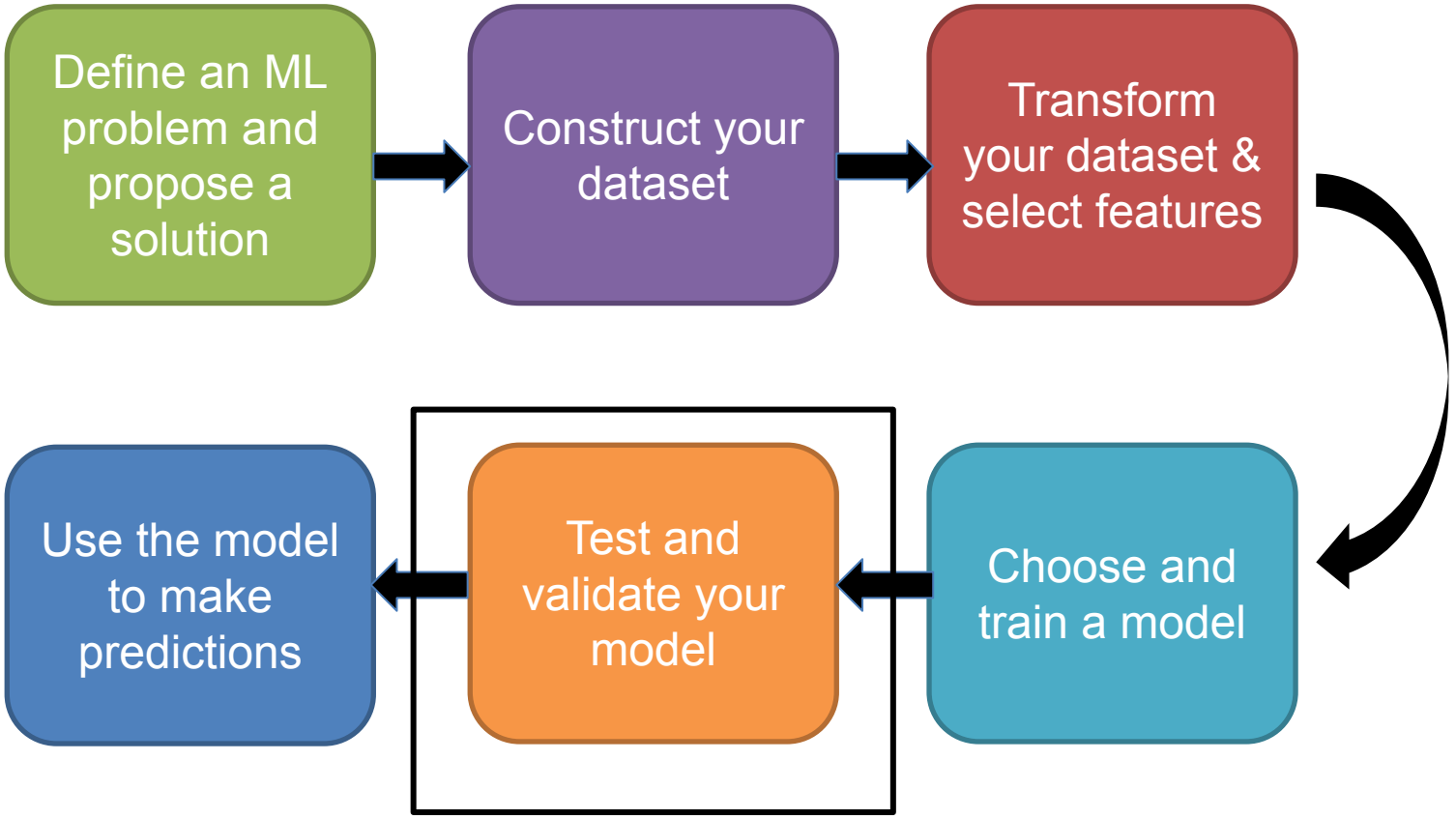
Error is difference between output and true label



Program Statistics

- Total number of lines: 215
- Total number of comment lines: 30
- Total number of coding lines: 185
- Time to train on training data: 1.5 sec
- Typical time to train and perform test run with program: 2.5 sec
- This program actually trains every time you run it although you can choose to run certain cells to use the trained ANN with new input data (<0.5 sec)

Machine Learning Workflow



Forward Propagation (Evaluation) of the Test Set

- Now that the model is trained, the test set can be evaluated
- This function will take the trained parameters (training function output) and the test set as input

```
def evaluation(params,tst_set):
  w0 = params[0]
  bh = params[1]
  w1 = params[2]
  bo = params[3]
```

Forward Propagation (Evaluation) of the Test Set *cont...*

- Using these parameters, the test set is propagated forward through the model
- The final output 'layer2' is the model's predictions on the test set

```
# First layer propagation
zh = np.dot(tst_set, w0) + bh
layer1 = sigmoid(zh)

# Second layer propagation
zo = np.dot(layer1, w1) + bo
layer2 = user_softmax(zo)

return layer2
```

Prediction Accuracy on Training & Test Sets (Confusion Matrices)

Training Set

	Setosa (Observed)	Virginica (Observed)	Versicolor (Observed)
Setosa (Predicted)	1	0	0
Virginica (Predicted)	0	0.97	0.03
Versicolor (Predicted)	0	0	1

Testing Set

	Setosa (Observed)	Virginica (Observed)	Versicolor (Observed)
Setosa (Predicted)	1	0	0
Virginica (Predicted)	0	1	0
Versicolor (Predicted)	0	0.05	0.95

Normally you want training & testing sets to be within 5-6% of each other
If they are much more than this, you have likely over-trained the system

Performance of IrisDT vs. IrisANN (Confusion Matrices)

IrisDT Script

	Setosa (Observed)	Virginica (Observed)	Versicolor (Observed)
Setosa (Predicted)	1	0	0
Virginica (Predicted)	0	1	0
Versicolor (Predicted)	0	0.07	0.93

IrisANN Script

	Setosa (Observed)	Virginica (Observed)	Versicolor (Observed)
Setosa (Predicted)	1	0	0
Virginica (Predicted)	0	1	0
Versicolor (Predicted)	0	0.05	0.95

Program Statistics

	Python	R
Packages used	numpy, pandas	N/A
Lines of code	205	252
Average Running time (seconds)	4.35	5.10

Summary

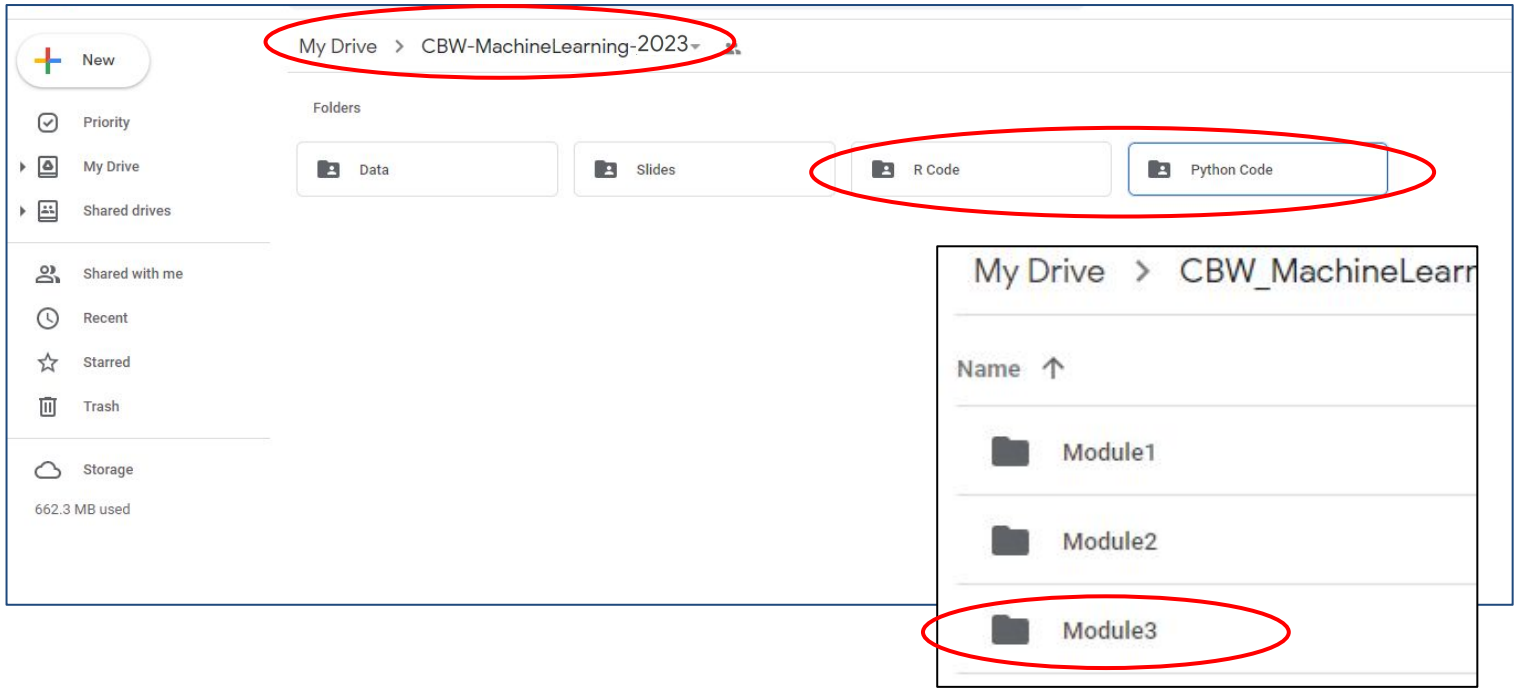
- We now have a neural network program written in "pure" python that predicts iris flower classes
- It has been thoroughly trained on a training set of 105 flowers and tested on a test set of 45
- This is fairly generic code so you could use or adapt this code for many other kinds of classification problems such as patients (cases or controls) with different levels of gene, protein or metabolite expression
- In the next section we will explore the code and run it on a few examples
- If you would like to work with in R rather than Python, follow the steps indicated in the Lab to find the corresponding scripts

Module 3 - Lab

Let's Take a Look at the Program

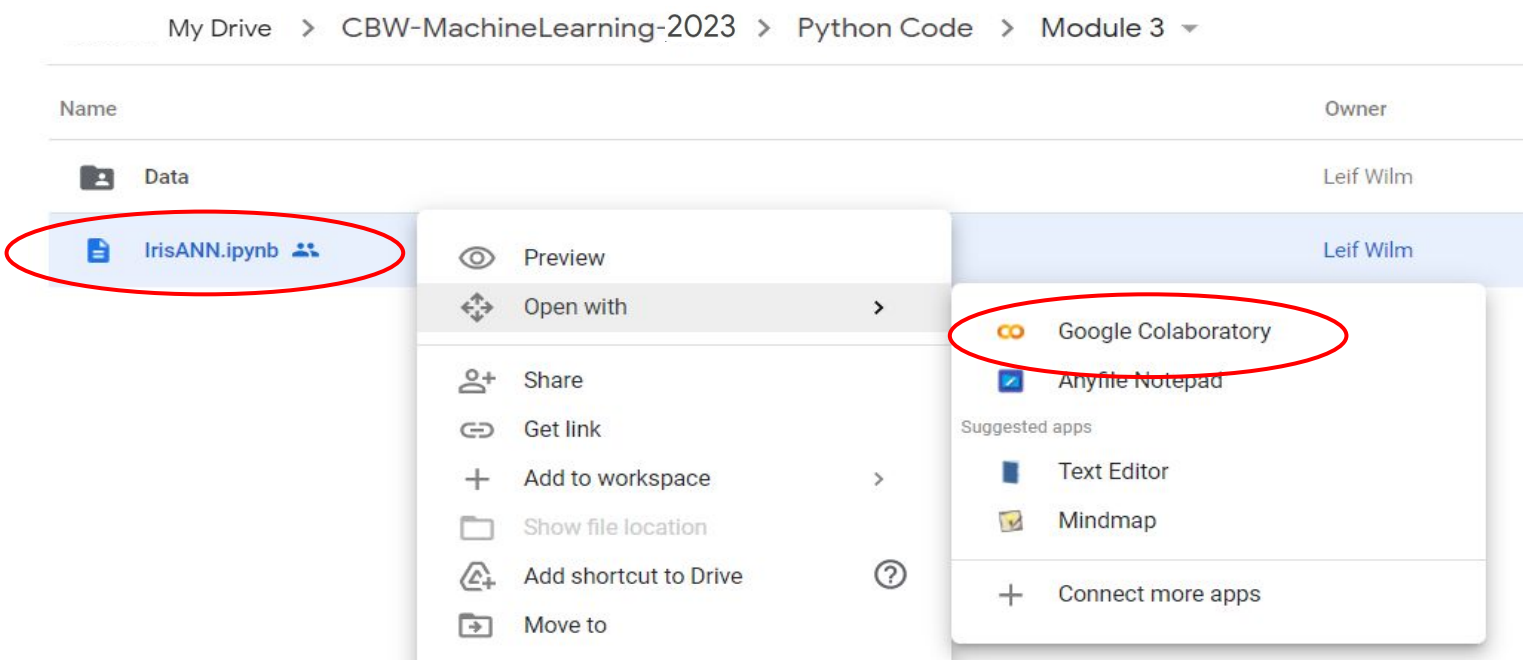
Open the Colab File in the Language of Your Choosing

- The python script 'IrisANN.ipynb', covered during lecture, will be used for the Lab portion
- However, you have the option of running the Lab in R
- To do so, find the R code 'IrisANN_R.ipynb' in the R Code folder under Module 3



Open the Colab File in the Language of Your Choosing *cont...*

- **Right click on 'IrisANN.ipynb' (or alternatively IrisANN_R.ipynb in the R Code folder under Module 3) and select open with Google Colaboratory**

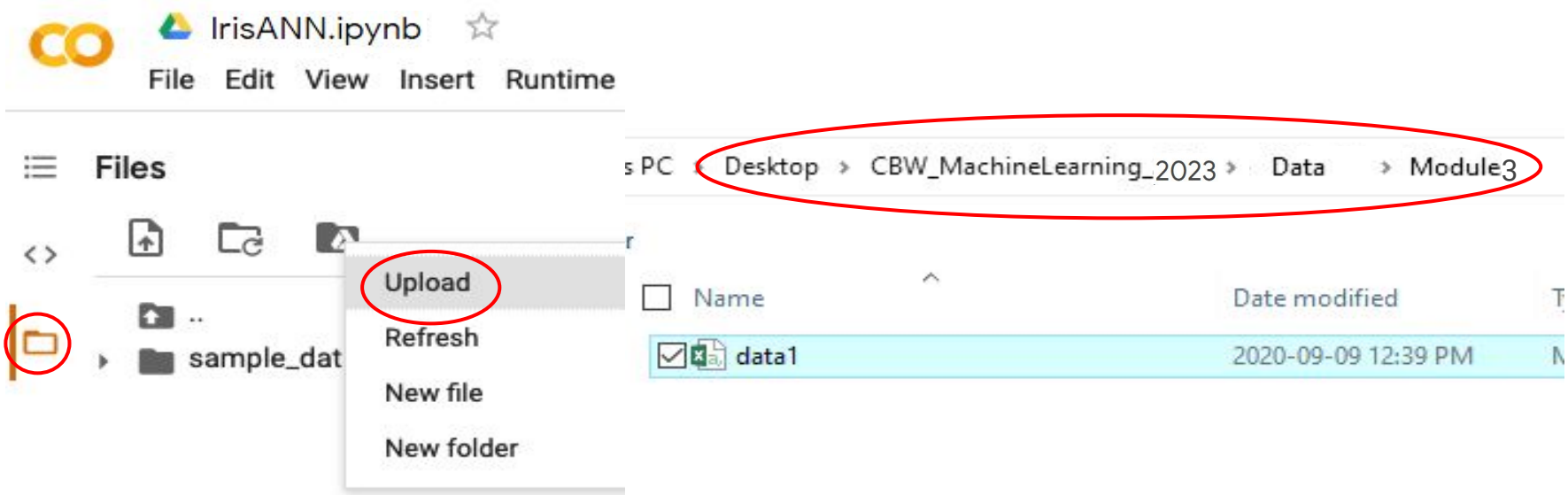


Study the Program

- Spend a few minutes studying the IrisANN.ipynb (or IrisANN_R.ipynb) program
- See if you can understand the logic of the program
- Ask questions of the TA's if you are unclear about some of the functions or Python code
- After studying the code, try running it

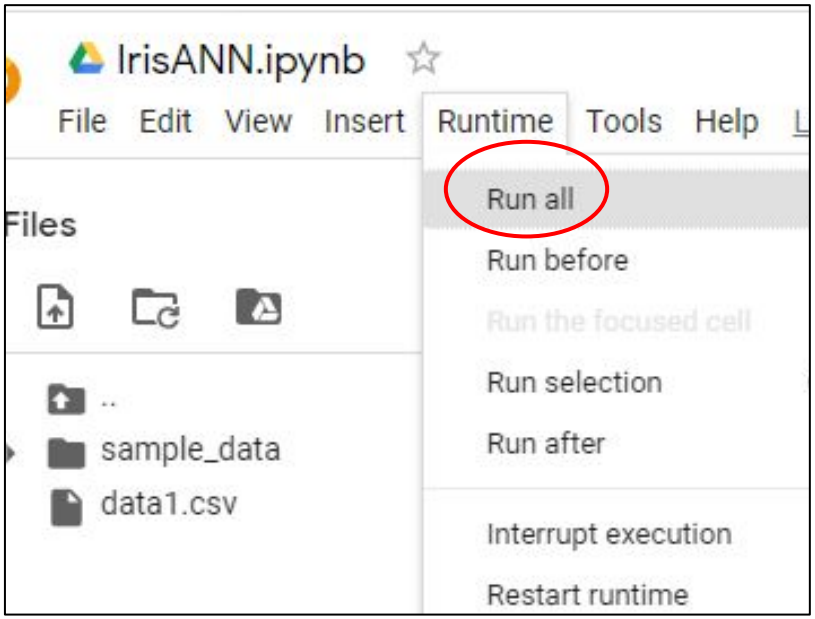
To Run: Upload the Data

- Once the Colab file is open, click the folder icon to the right
- When the panel opens, right click inside the panel and click upload
- Then find the CBW_MachineLearning_2023 folder on your desktop
- Open the Data folder followed by the Module 3 folder, and open the corresponding data file named 'data1.csv'



Exercise 3.1- Run The Program

- **Select ‘Run all’ from the ‘Runtime’ menu**



- **This will train the program (on the training set of 105) and test the program (on the test set of 45) and produce 1 confusion matrix on the test data**
- **Check to see if your numbers match with the slide’s numbers**

Exercise 3.1 – Change Epoch Numbers

- **Go to cell1 3 and change the number of epochs from 1000 to 100**

Exercise 3.1

Cell 13 Selecting parameters and running the training function

```
'''Change these values'''
learning_rate = 0.01
batch_size = 10
hidden_size = 5
epochs = 1000

trained_params, error, errors = training(learning_rate, batch_size, epochs)
```

Exercise 3.1 – Optimize Epoch Numbers

- Run cells 13 through 16 and observe the confusion matrix
- The model will misclassify many versicolor samples
- Try changing the number of epochs to other numbers
- What is the lowest number of epochs that produces desirable results?
- What combination of parameters yields the highest training accuracy?

Conclusion

- This concludes the “formal” section on ANNs for iris classification
- Feel free to explore the code, modify it if you wish or to review the slides and explanations – talk to the TAs

We are on a Coffee Break & Networking Session

Workshop Sponsors: