Neural Network from Scratch

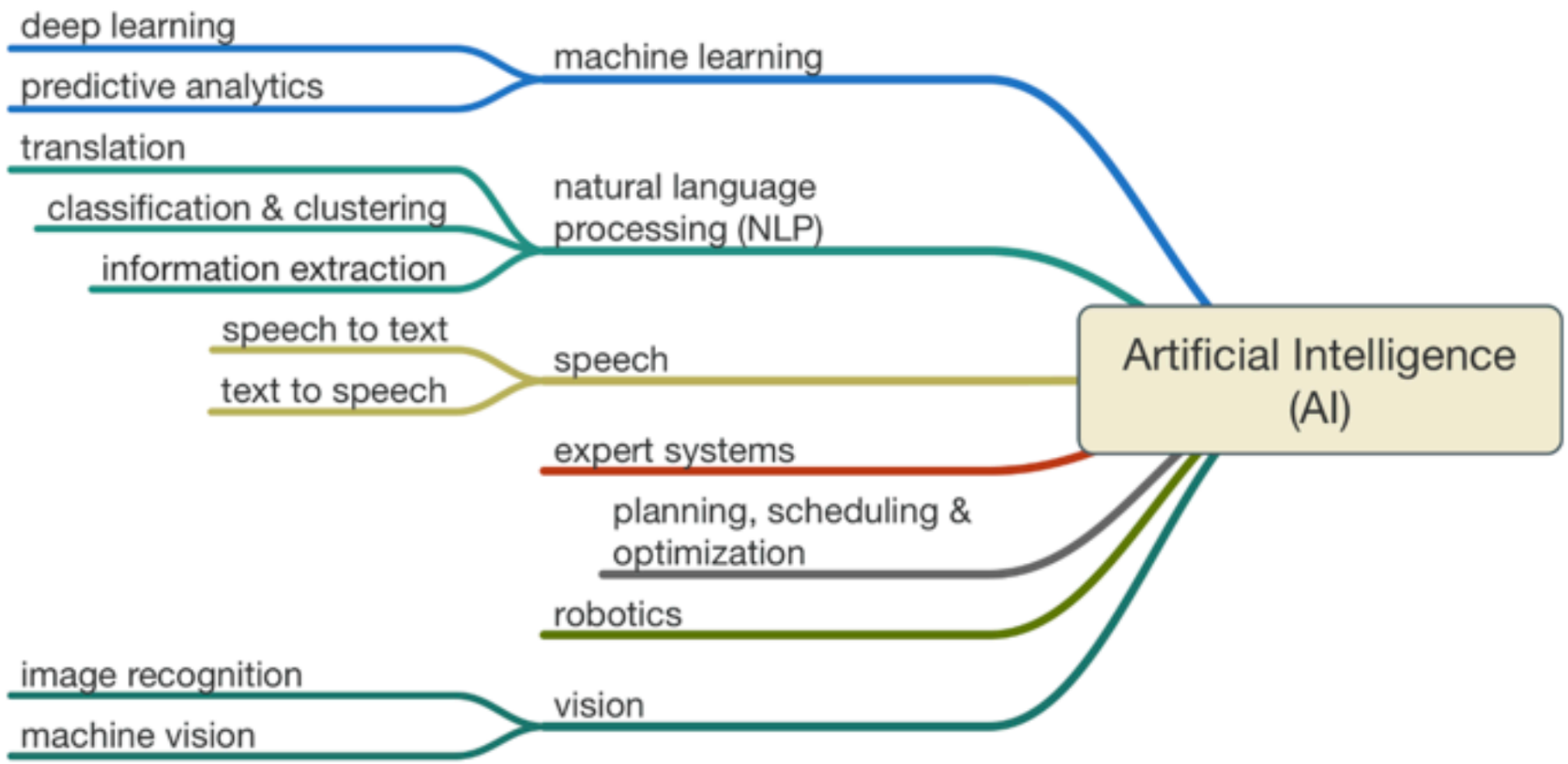
Subash Gandyer
Data Scientist, HealthChain Inc

Agenda

1. Introduction to AI, ML and DL
2. Machine Learning Vs Deep Learning
3. First Neural Network
4. Neural Network Theory
5. Structure of a Neuron
6. Coding a Neuron
7. Structure of a Neural Network
8. Coding a Neural Network
9. Loss function
10. Coding the loss function
11. Computation Graph
12. Calculus Theory
13. Coding the partial derivatives
14. Optimization Algorithm - Stochastic Gradient Descent
15. Coding the Optimization algorithm
16. Putting all things together
17. Deep Neural Network
18. More examples

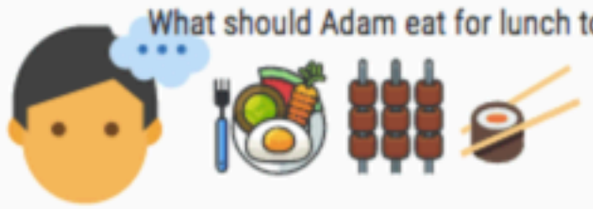
AI, ML, DL

Artificial Intelligence



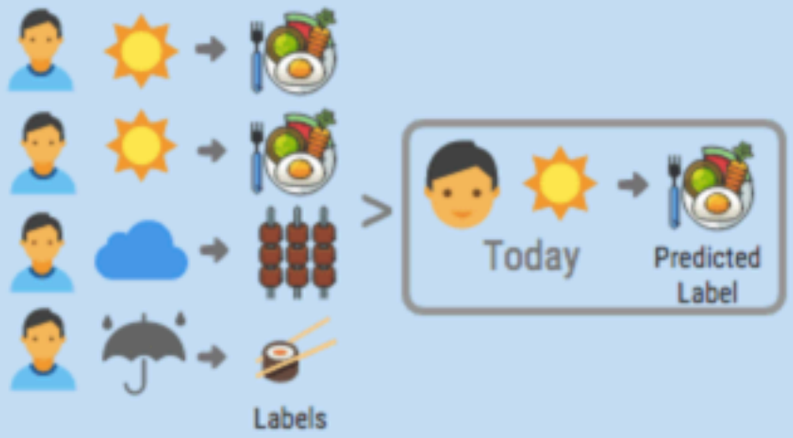
Decrypting the machine learning black box

What's for lunch?



Source: https://www.linkedin.com/in/pohwanting/

Supervised Learning



Learning: Machine is trained to associate input data with their labels
Output: Predict label based on data

Unsupervised Learning

Learning: Machine is trained to understand which data points are similar
Output: Which group is data point most similar to

Reinforcement Learning

Learning: Machine is trained through feedbacks and rewards
Output: Predict output that will get the most reward

Machine Learning

- Machine Learning
- Classification
- Regression
- Clustering
- Association
- Decision Trees
- Support Vector Machines

Representation	Evaluation	Optimization
Instances	Accuracy/Error rate	Combinatorial optimization
K-nearest neighbor	Precision and recall	Greedy search
Support vector machines	Squared error	Beam search
Hyperplanes	Likelihood	Branch-and-bound
Naive Bayes	Posterior probability	Continuous optimization
Logistic regression	Information gain	Unconstrained
Decision trees	K-L divergence	Gradient descent
Sets of rules	Cost/Utility	Conjugate gradient
Propositional rules	Margin	Quasi-Newton methods
Logic programs		Constrained
Neural networks		Linear programming
Graphical models		Quadratic programming
Bayesian networks		
Conditional random fields		

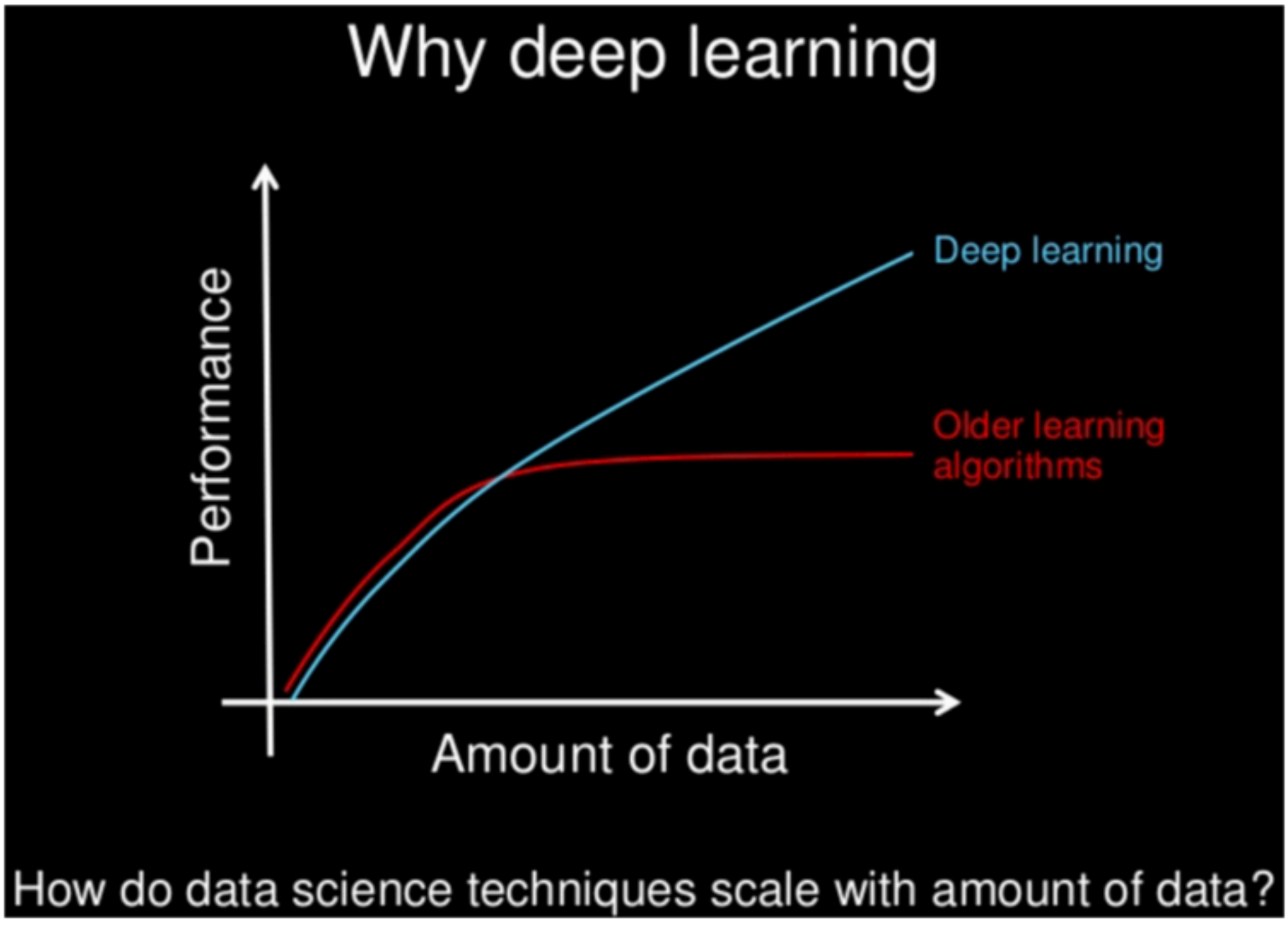
https://github.com/dformoso/machine-learning-mindmap

Machine Learning
vs
Deep Learning

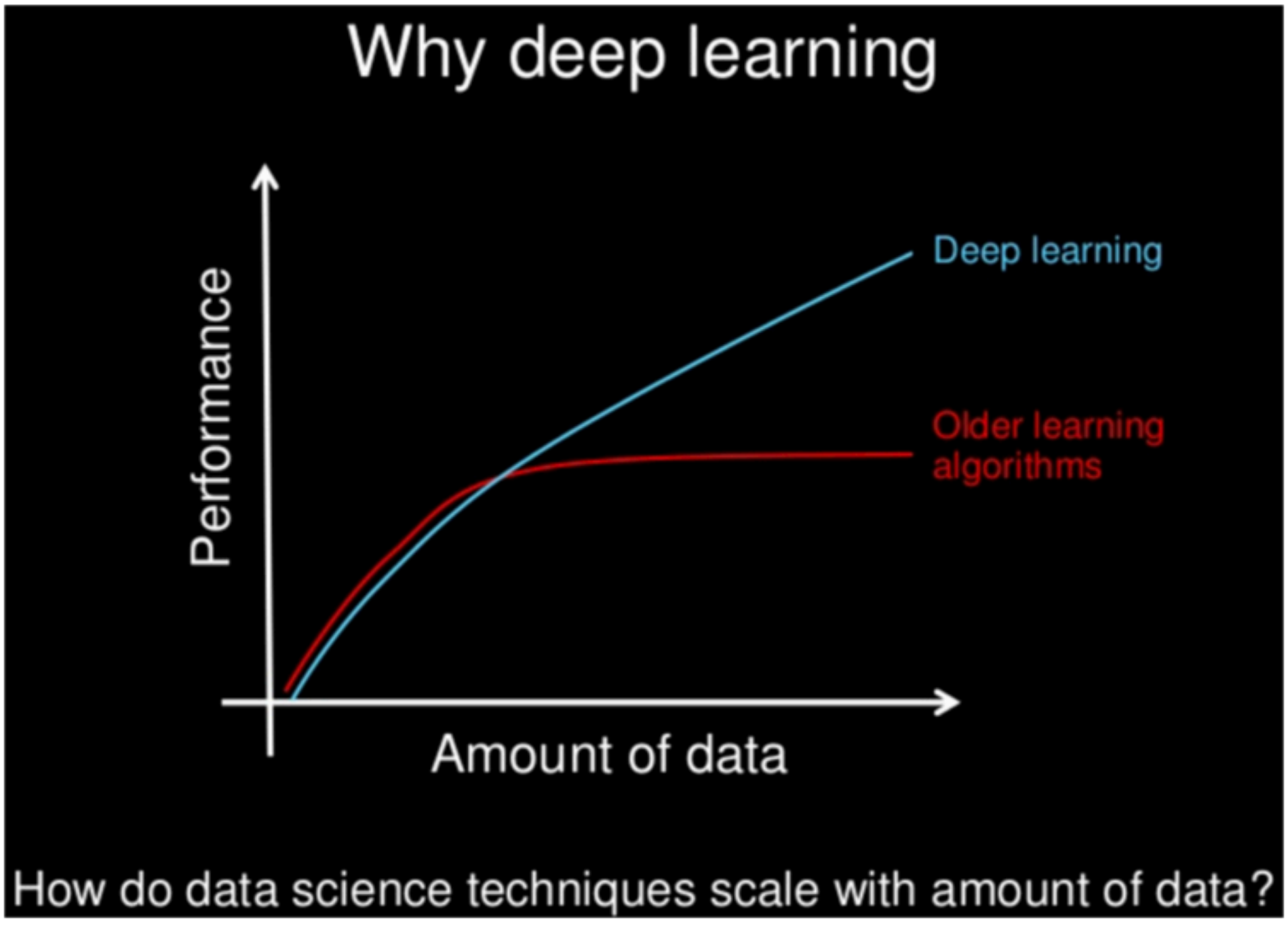
ML vs DL

- Data Dependencies
- Hardware Dependencies
- Feature Engineering
- Problem Solving Approach
- Execution Time
- Interpretability

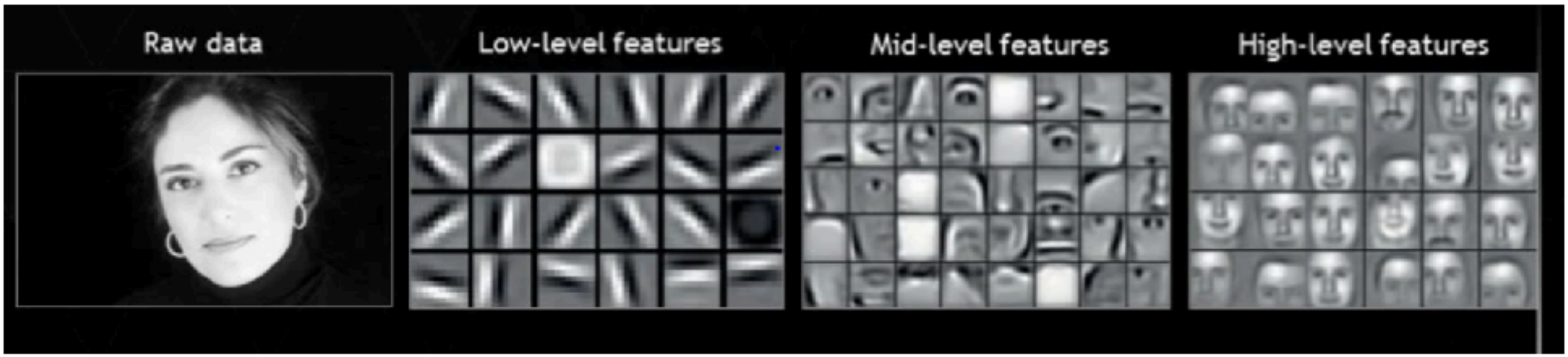
1. Data Dependencies



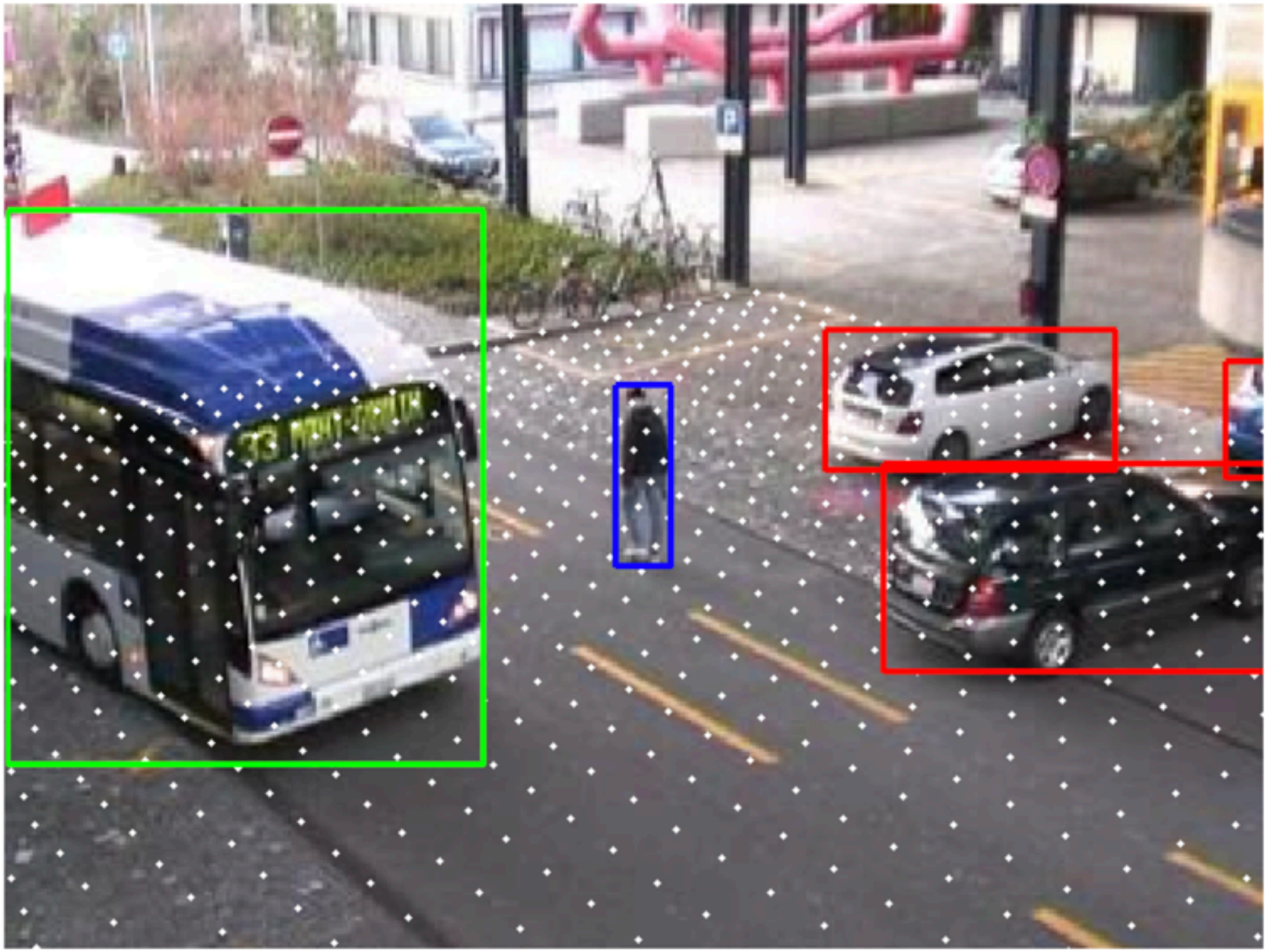
2. Hardware Dependencies



3. Feature Engineering



4. Problem Solving Approach



5. Execution Time

Training Time
Testing Time

Usually, a deep learning algorithm takes a long time to train. This is because there are so many parameters in a deep learning algorithm that training them takes longer than usual. State of the art deep learning algorithm ResNet takes about two weeks to train completely from scratch. Whereas machine learning comparatively takes much less time to train, ranging from a few seconds to a few hours.

This is turn is completely reversed on testing time. At test time, deep learning algorithm takes much less time to run. Whereas, if you compare it with k-nearest neighbors (a type of machine learning algorithm), test time increases on increasing the size of data. Although this is not applicable on all machine learning algorithms, as some of them have small testing times too.

6. Interpretability

Suppose we use deep learning to give automated scoring to essays. The performance it gives in scoring is quite excellent and is near human performance. But there's is an issue. It does not reveal why it has given that score. Indeed mathematically you can find out which nodes of a deep neural network were activated, but we don't know what there neurons were supposed to model and what these layers of neurons were doing collectively. So we fail to interpret the results.

On the other hand, machine learning algorithms like decision trees give us crisp rules as to why it chose what it chose, so it is particularly easy to interpret the reasoning behind it. Therefore, algorithms like decision trees and linear/logistic regression are primarily used in industry for interpretability.

First Neural Network

MNIST

```python
import numpy as np
import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical

train_images = mnist.train_images()
train_labels = mnist.train_labels()
test_images = mnist.test_images()
test_labels = mnist.test_labels()

# Normalize the images.
train_images = (train_images / 255) - 0.5
test_images = (test_images / 255) - 0.5

# Flatten the images.
train_images = train_images.reshape((-1, 784))
test_images = test_images.reshape((-1, 784))

# Build the model.
model = Sequential([
  Dense(64, activation='relu', input_shape=(784,)),
  Dense(64, activation='relu'),
  Dense(10, activation='softmax'),
])

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

# Train the model.
model.fit(
  train_images,
  to_categorical(train_labels),
  epochs=5,
  batch_size=32,
)

# Evaluate the model.
model.evaluate(
  test_images,
  to_categorical(test_labels)
)

# Predict on the first 5 test images.
predictions = model.predict(test_images[:5])

# Print our model's predictions.
print(np.argmax(predictions, axis=1)) # [7, 2, 1, 0, 4]

# Check our predictions against the ground truths.
print(test_labels[:5]) # [7, 2, 1, 0, 4]
```

Let’s code our first Neural Network

00 - First Neural Network

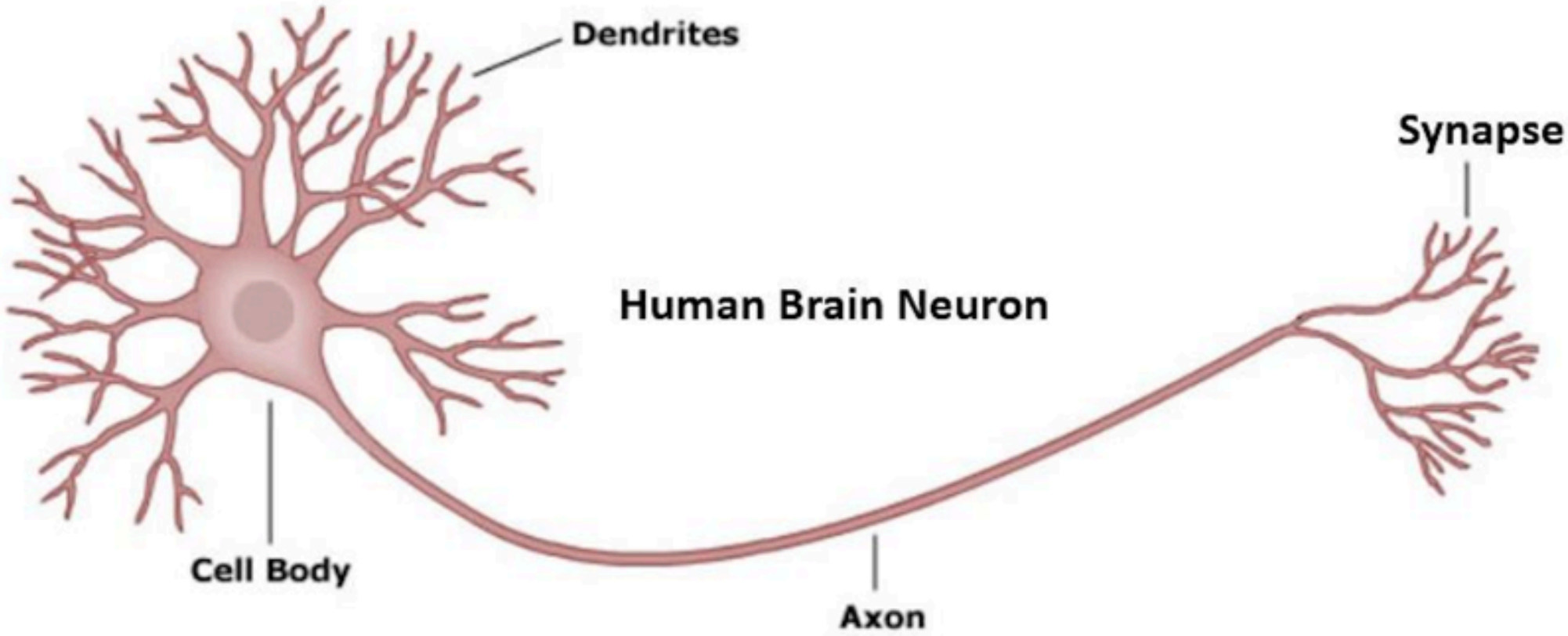
Neural Network Theory

NN Theory

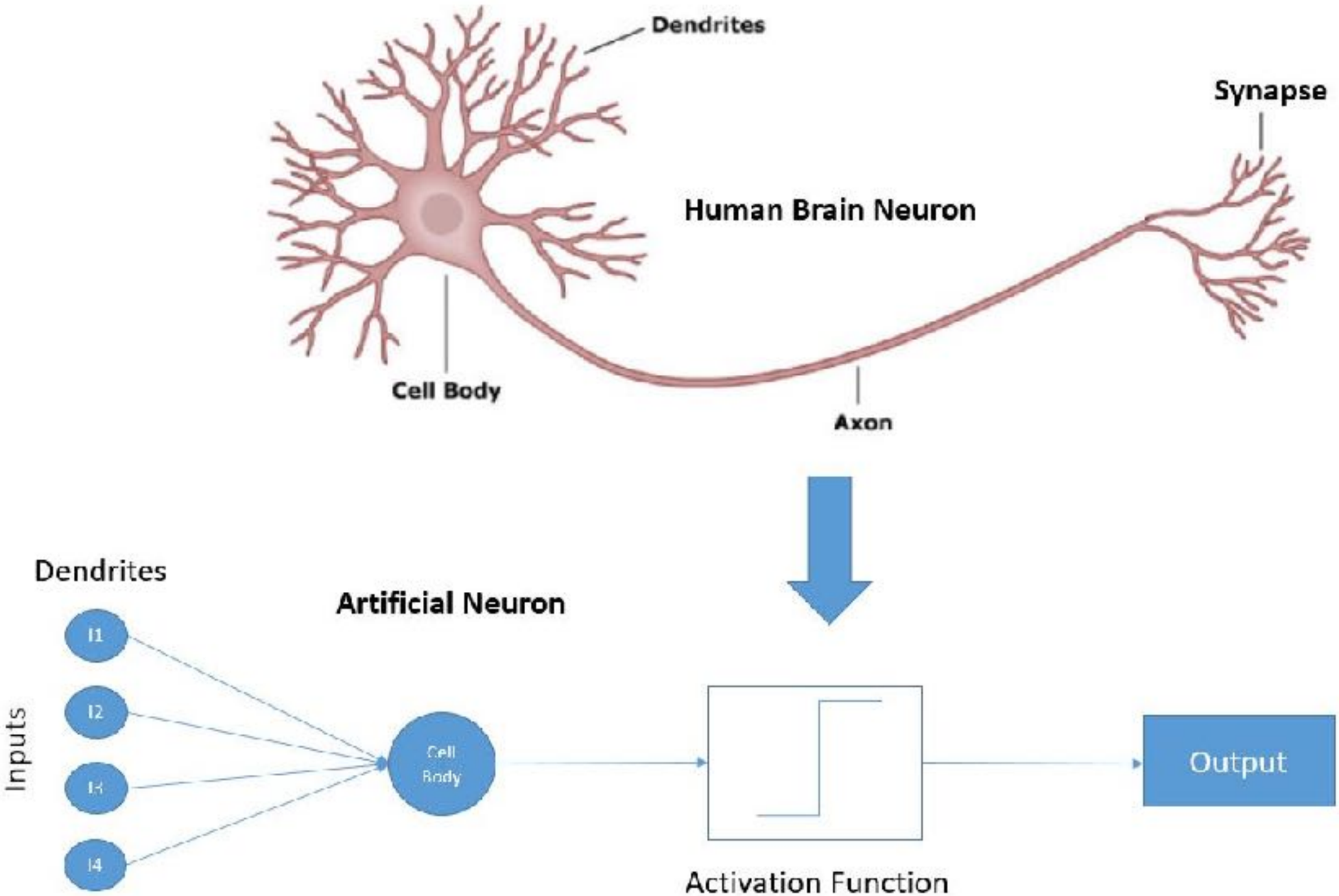
A neural network is a supervised learning algorithm which means that we provide it the input data containing the independent variables and the output data that contains the dependent variable. For instance, in our example our independent variables are smoking, obesity and exercise. The dependent variable is whether a person is diabetic or not.

In the beginning, the neural network makes some random predictions, these predictions are matched with the correct output and the error or the difference between the predicted values and the actual values is calculated. The function that finds the difference between the actual value and the propagated values is called the cost function. The cost here refers to the error. Our objective is to minimize the cost function. Training a neural network basically refers to minimizing the cost function. We will see how we can perform this task.

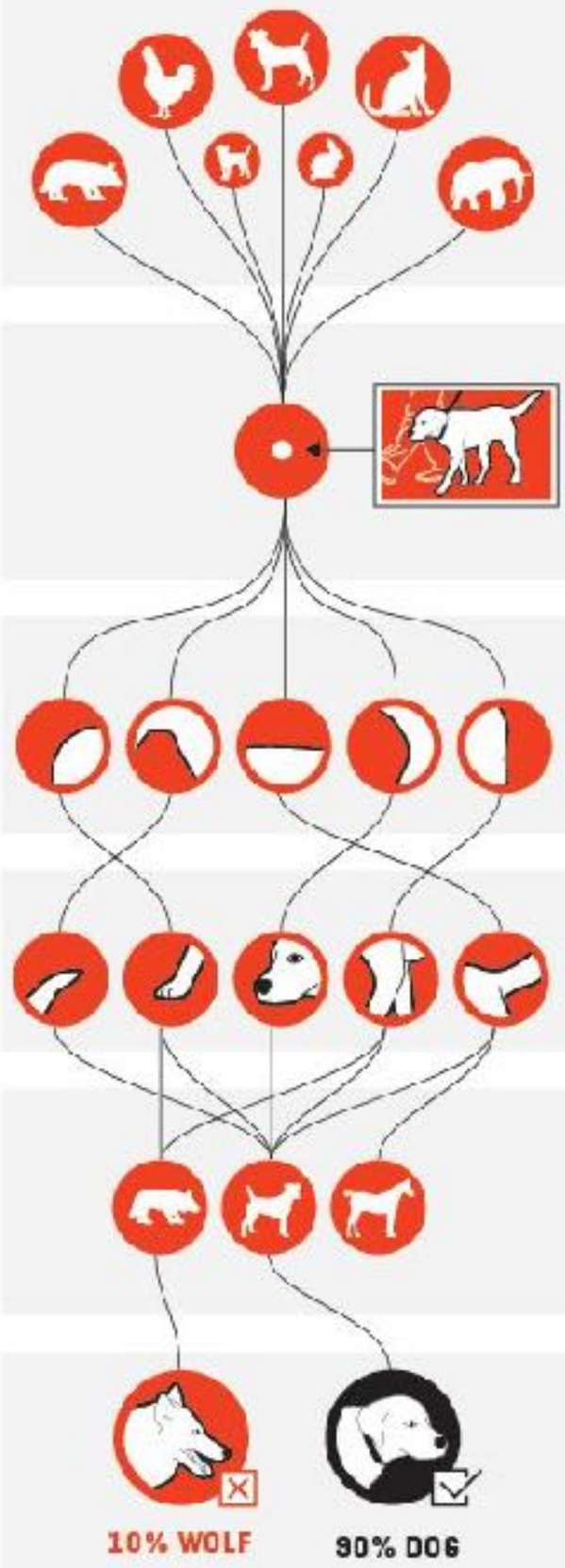
Neuron



Biologically Inspired Artificial Neuron



HOW NEURAL NETWORKS RECOGNIZE A DOG IN A PHOTO



TRAINING
During the training phase, a neural network is fed thousands of labeled images of various animals, learning to classify them.

INPUT
An unlabeled image is shown to the pretrained network.

FIRST LAYER
The neurons respond to different simple shapes, like edges.

HIGHER LAYER
Neurons respond to more complex structures.

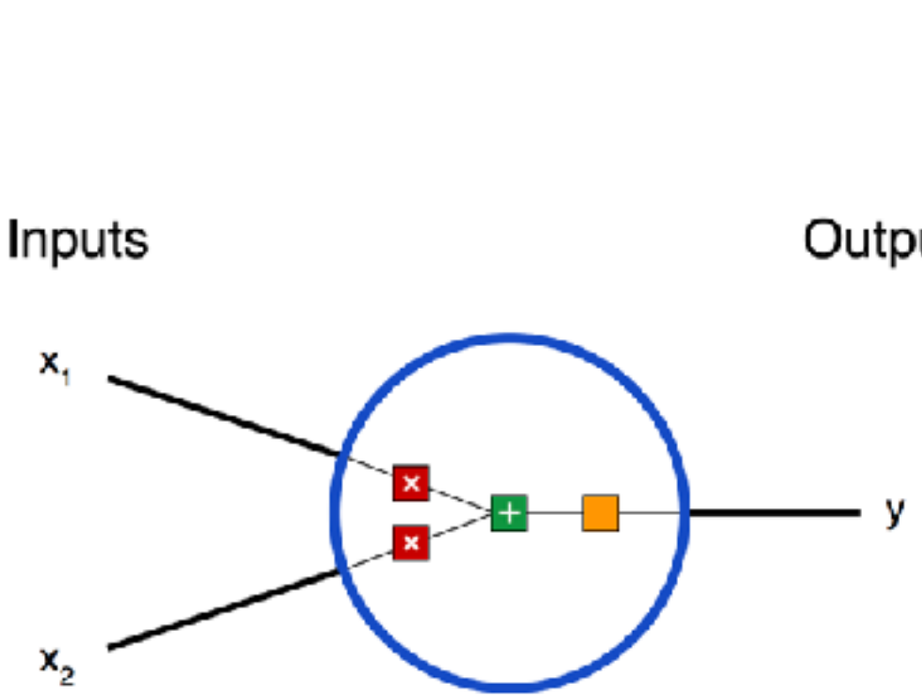
TOP LAYER
Neurons respond to highly complex, abstract concepts that we would identify as different animals.

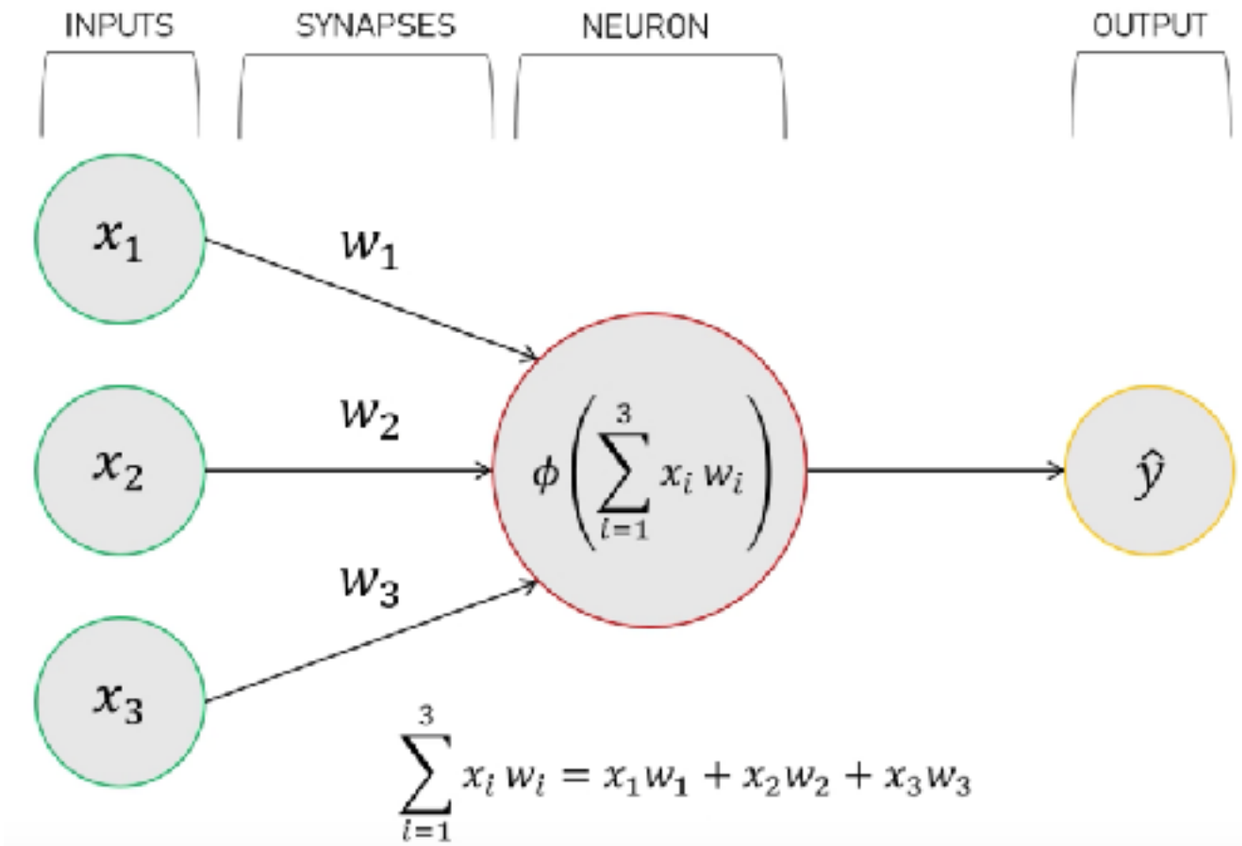
OUTPUT
The network predicts what the object most likely is, based on its training.

Building Blocks

- Neuron
- Activation function
- Loss function
- Cost function
- Computation Graphs
- Back Propagation
- Optimization algorithm

Structure of a Neuron





Formal definition of a Neuron

Neuron

```
class Neuron:
  def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias
```

```
def feedforward(self, inputs):
    # Weight inputs, add bias, then use the activation function
    total = np.dot(self.weights, inputs) + self.bias
    return total
```

Activation function

Activation function

Example

```
def sigmoid(x):
  # Our activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))
```

Let’s code activation function

01 - Sigmoid function

Coding of a Neuron

```python
import numpy as np

def sigmoid(x):
  # Our activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))

class Neuron:
  def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias

  def feedforward(self, inputs):
    # Weight inputs, add bias, then use the activation function
    total = np.dot(self.weights, inputs) + self.bias
    return sigmoid(total)

weights = np.array([0, 1]) # w1 = 0, w2 = 1
bias = 4                   # b = 4
n = Neuron(weights, bias)

x = np.array([2, 3])       # x1 = 2, x2 = 3
print(n.feedforward(x))    # 0.9990889488055994
```

Let’s code a neuron

02 - Neuron

Neural Network

Feed Forward

In the feed-forward part of a neural network, predictions are made based on the values in the input nodes and the weights. If you look at the neural network in the above figure, you will see that we have three features in the dataset: smoking, obesity, and exercise, therefore we have three nodes in the first layer, also known as the input layer. We have replaced our feature names with the variable x, for generality in the figure above.

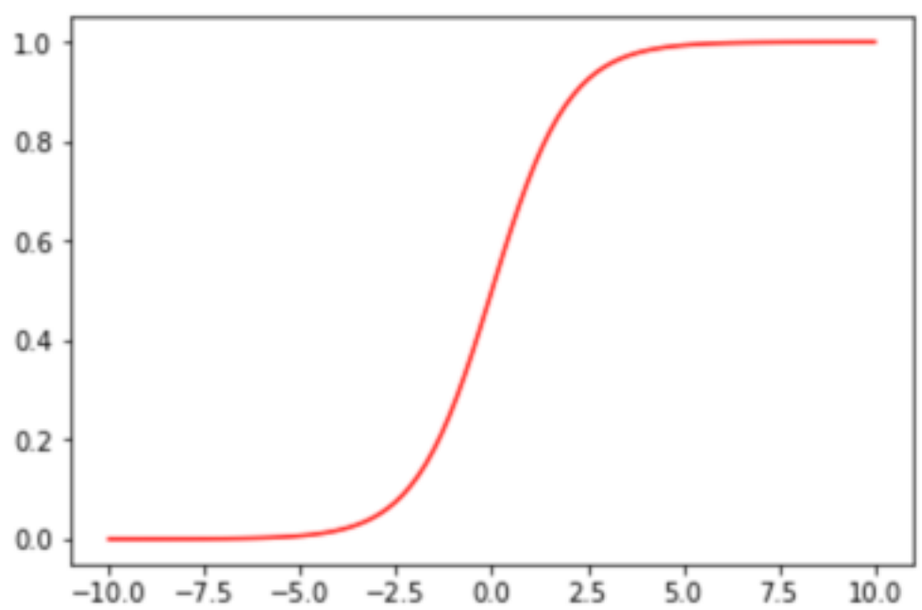
The weights of a neural network are basically the strings that we have to adjust in order to be able to correctly predict our output. For now, just remember that for each input feature, we have one weight.

1. Calculate the dot product between inputs and weights

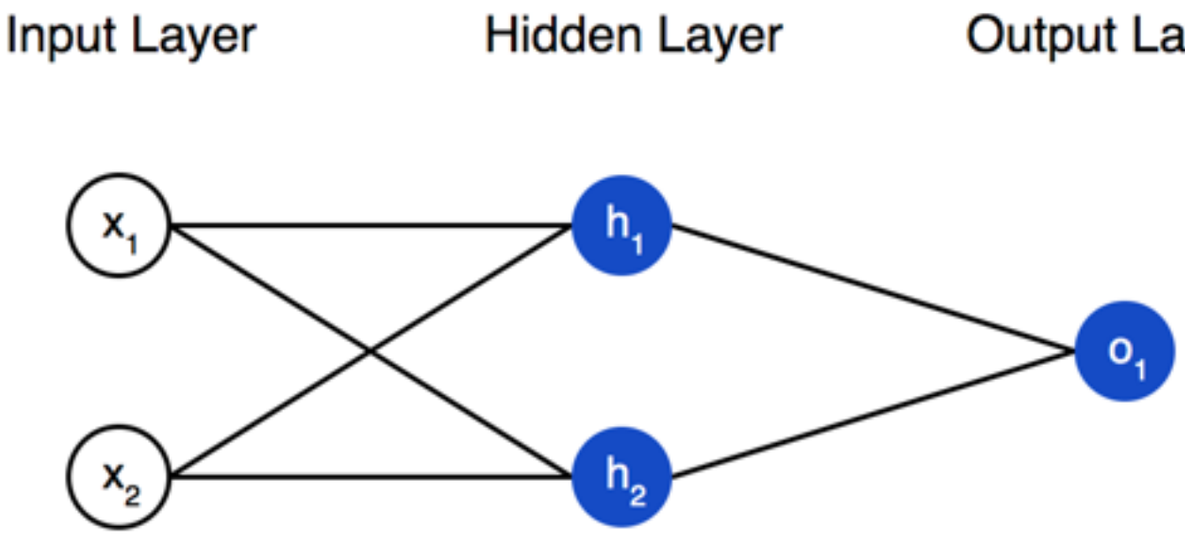
2. Pass the result through an activation function

Forward propagation

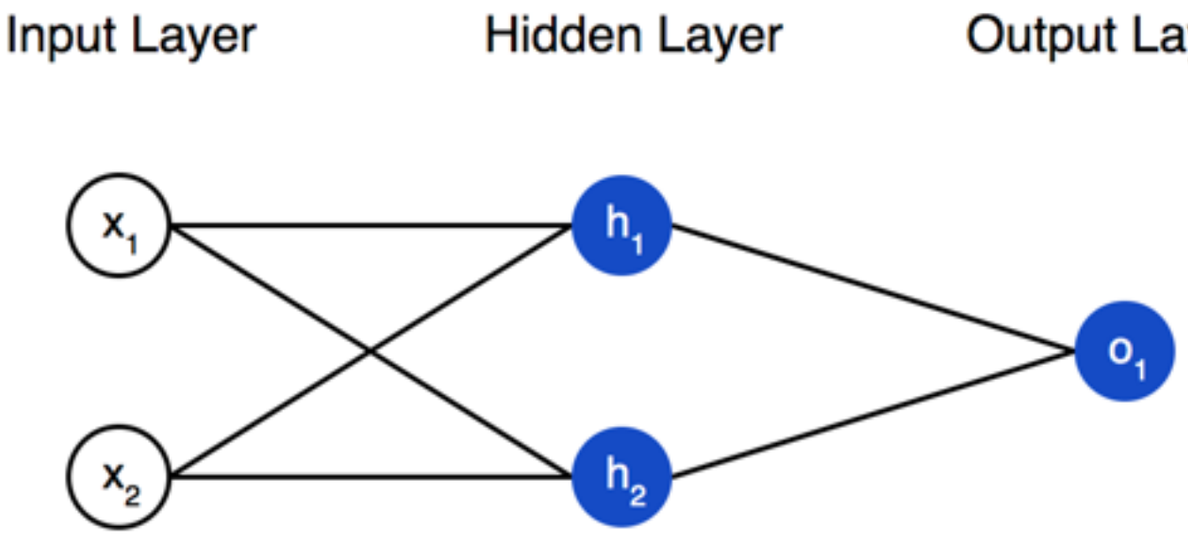
```
input = np.linspace(-10, 10, 100)

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

from matplotlib import pyplot as plt
plt.plot(input, sigmoid(input), c="r")
```



Feed forward Calculation



w = [0, 1]
b = 0
x = [2, 3]
O_1 = ?



w = [0, 1]
b = 0
x = [2, 3]
o_1 = 0.7216

$$
\begin{aligned}
h_1 = h_2 &= f(w \cdot x + b) \\
&= f((0 * 2) + (1 * 3) + 0) \\
&= f(3) \\
&= 0.9526
\end{aligned}
$$

$$
\begin{aligned}
o_1 &= f(w \cdot [h_1, h_2] + b) \\
&= f((0 * h_1) + (1 * h_2) + 0) \\
&= f(0.9526) \\
&= \boxed{0.7216}
\end{aligned}
$$

Let’s Code a Feedforward Calculation

03 - Feedforward Calculation

Coding a Feedforward NN

```
import numpy as np

# Class Neuron code goes here…

class OurNeuralNetwork:
  def __init__(self):
    weights = np.array([0, 1])
    bias = 0

    # The Neuron class here is from the previous section
    self.h1 = Neuron(weights, bias)
    self.h2 = Neuron(weights, bias)
    self.o1 = Neuron(weights, bias)

  def feedforward(self, x):
    out_h1 = self.h1.feedforward(x)
    out_h2 = self.h2.feedforward(x)

    # The inputs for o1 are the outputs from h1 and h2
    out_o1 = self.o1.feedforward(np.array([out_h1, out_h2]))

    return out_o1

network = OurNeuralNetwork()
x = np.array([2, 3])
print(network.feedforward(x)) # 0.7216325609518421
```

Let’s Code a Feedforward Neural Network

04 - Feedforward Neural Network

Training a Neural Network

Mean shifting

Name	Weight (lb)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1

Loss function

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$

Example: Loss function

Name	y_{true}	y_{pred}	$(y_{true} - y_{pred})^2$
Alice	1	0	1
Bob	0	0	0
Charlie	0	0	0
Diana	1	0	1

$$\text{MSE} = \frac{1}{4}(1 + 0 + 0 + 1) = \boxed{0.5}$$

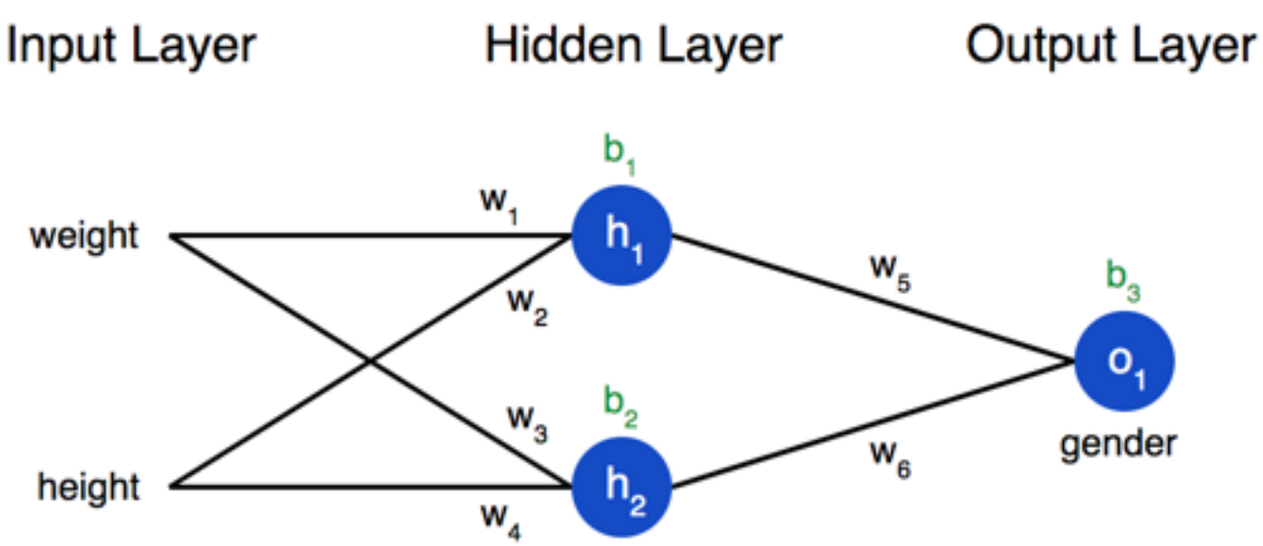
Coding a loss function

```python
import numpy as np

def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()

y_true = np.array([1, 0, 0, 1])
y_pred = np.array([0, 0, 0, 0])

print(mse_loss(y_true, y_pred)) # 0.5
```

Let’s code a Loss function

05 - Loss function

Cost Function



$$L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$$

$$L = (1 - y_{pred})^2$$

How would Loss **L** change if we change or tweak $\mathbf{w_1}$?

Calculus Revisited

How would Loss **L** change if we change or tweak $\mathbf{w_1}$?

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1}$$

Chain Rule

$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial (1 - y_{pred})^2}{\partial y_{pred}} = \boxed{-2(1 - y_{pred})}$$

$$y_{pred} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$

$$\frac{\partial y_{pred}}{\partial w_1} = \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial y_{pred}}{\partial h_1} = \boxed{w_5 * f'(w_5 h_1 + w_6 h_2 + b_3)}$$

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = \boxed{x_1 * f'(w_1 x_1 + w_2 x_2 + b_1)}$$

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

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x) * (1 - f(x))$$

$$\boxed{\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}}$$

Example: Calculation

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$
$$= f(-2 + -1 + 0)$$
$$= 0.0474$$

$$h_2 = f(w_3 x_1 + w_4 x_2 + b_2) = 0.0474$$

$$o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$
$$= f(0.0474 + 0.0474 + 0)$$
$$= 0.524$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial L}{\partial y_{pred}} = -2(1 - y_{pred})$$
$$= -2(1 - 0.524)$$
$$= -0.952$$

$$\frac{\partial y_{pred}}{\partial h_1} = w_5 * f'(w_5 h_1 + w_6 h_2 + b_3)$$
$$= 1 * f'(0.0474 + 0.0474 + 0)$$
$$= f(0.0948) * (1 - f(0.0948))$$
$$= 0.249$$

$$\frac{\partial h_1}{\partial w_1} = x_1 * f'(w_1 x_1 + w_2 x_2 + b_1)$$
$$= -2 * f'(-2 + -1 + 0)$$
$$= -2 * f(-3) * (1 - f(-3))$$
$$= -0.0904$$

$$\frac{\partial L}{\partial w_1} = -0.952 * 0.249 * -0.0904$$
$$= \boxed{0.0214}$$

Let’s hand compute partial derivatives

Coding Partial derivatives

```
# --- Calculate partial derivatives.
        # --- Naming: d_L_d_w1 represents "partial L / partial
w1"
        d_L_d_ypred = -2 * (y_true - y_pred)

        # Neuron o1
        d_ypred_d_w5 = h1 * deriv_sigmoid(sum_o1)
        d_ypred_d_w6 = h2 * deriv_sigmoid(sum_o1)
        d_ypred_d_b3 = deriv_sigmoid(sum_o1)

        d_ypred_d_h1 = self.w5 * deriv_sigmoid(sum_o1)
        d_ypred_d_h2 = self.w6 * deriv_sigmoid(sum_o1)

        # Neuron h1
        d_h1_d_w1 = x[0] * deriv_sigmoid(sum_h1)
        d_h1_d_w2 = x[1] * deriv_sigmoid(sum_h1)
        d_h1_d_b1 = deriv_sigmoid(sum_h1)

        # Neuron h2
        d_h2_d_w3 = x[0] * deriv_sigmoid(sum_h2)
        d_h2_d_w4 = x[1] * deriv_sigmoid(sum_h2)
        d_h2_d_b2 = deriv_sigmoid(sum_h2)
```

Let’s compute more examples of partial derivatives

Stochastic Gradient Algorithm

Stochastic Gradient Algorithm

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

η is a constant called the **learning rate** that controls how fast we train.

- If $\frac{\partial L}{\partial w_1}$ is positive, w_1 will decrease, which makes L decrease.
- If $\frac{\partial L}{\partial w_1}$ is negative, w_1 will increase, which makes L decrease.

1. Choose **one** sample from our dataset. This is what makes it *stochastic* gradient descent – we only operate on one sample at a time.
2. Calculate all the partial derivatives of loss with respect to weights or biases (e.g. $\frac{\partial L}{\partial w_1}$, $\frac{\partial L}{\partial w_2}$, etc).
3. Use the update equation to update each weight and bias.
4. Go back to step 1.

```
# --- Update weights and biases
        # Neuron h1
        self.w1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w1
        self.w2 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w2
        self.b1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_b1

        # Neuron h2
        self.w3 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w3
        self.w4 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w4
        self.b2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_b2

        # Neuron o1
        self.w5 -= learn_rate * d_L_d_ypred * d_ypred_d_w5
        self.w6 -= learn_rate * d_L_d_ypred * d_ypred_d_w6
        self.b3 -= learn_rate * d_L_d_ypred * d_ypred_d_b3
```

Let’s code Stochastic Gradient algorithm

Let’s code a Complete Neural Network

100 - Full Neural Network

```python
import numpy as np

def sigmoid(x):
  # Sigmoid activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))

def deriv_sigmoid(x):
  # Derivative of sigmoid: f'(x) = f(x) * (1 - f(x))
  fx = sigmoid(x)
  return fx * (1 - fx)

def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()
```

```python
class OurNeuralNetwork:
  '''
  A neural network with:
    - 2 inputs
    - a hidden layer with 2 neurons (h1, h2)
    - an output layer with 1 neuron (o1)

  *** DISCLAIMER ***:
  The code below is intended to be simple and educational, NOT optimal.
  Real neural net code looks nothing like this. DO NOT use this code.
  Instead, read/run it to understand how this specific network works.
  '''
  def __init__(self):
    # Weights
    self.w1 = np.random.normal()
    self.w2 = np.random.normal()
    self.w3 = np.random.normal()
    self.w4 = np.random.normal()
    self.w5 = np.random.normal()
    self.w6 = np.random.normal()

    # Biases
    self.b1 = np.random.normal()
    self.b2 = np.random.normal()
    self.b3 = np.random.normal()

  def feedforward(self, x):
    # x is a numpy array with 2 elements.
    h1 = sigmoid(self.w1 * x[0] + self.w2 * x[1] + self.b1)
    h2 = sigmoid(self.w3 * x[0] + self.w4 * x[1] + self.b2)
    o1 = sigmoid(self.w5 * h1 + self.w6 * h2 + self.b3)
    return o1
```

```python
def train(self, data, all_y_trues):
    '''
    - data is a (n x 2) numpy array, n = # of samples in the dataset.
    - all_y_trues is a numpy array with n elements.
      Elements in all_y_trues correspond to those in data.
    '''
    learn_rate = 0.1
    epochs = 1000 # number of times to loop through the entire dataset

    for epoch in range(epochs):
      for x, y_true in zip(data, all_y_trues):
        # --- Do a feedforward (we'll need these values later)
        sum_h1 = self.w1 * x[0] + self.w2 * x[1] + self.b1
        h1 = sigmoid(sum_h1)

        sum_h2 = self.w3 * x[0] + self.w4 * x[1] + self.b2
        h2 = sigmoid(sum_h2)

        sum_o1 = self.w5 * h1 + self.w6 * h2 + self.b3
        o1 = sigmoid(sum_o1)
        y_pred = o1

        # --- Calculate partial derivatives.
        # --- Naming: d_L_d_w1 represents "partial L / partial w1"
        d_L_d_ypred = -2 * (y_true - y_pred)

        # Neuron o1
        d_ypred_d_w5 = h1 * deriv_sigmoid(sum_o1)
        d_ypred_d_w6 = h2 * deriv_sigmoid(sum_o1)
        d_ypred_d_b3 = deriv_sigmoid(sum_o1)

        d_ypred_d_h1 = self.w5 * deriv_sigmoid(sum_o1)
        d_ypred_d_h2 = self.w6 * deriv_sigmoid(sum_o1)
        # Neuron h1
        d_h1_d_w1 = x[0] * deriv_sigmoid(sum_h1)
        d_h1_d_w2 = x[1] * deriv_sigmoid(sum_h1)
        d_h1_d_b1 = deriv_sigmoid(sum_h1)
        # Neuron h2
        d_h2_d_w3 = x[0] * deriv_sigmoid(sum_h2)
        d_h2_d_w4 = x[1] * deriv_sigmoid(sum_h2)
        d_h2_d_b2 = deriv_sigmoid(sum_h2)

        # --- Update weights and biases
        # Neuron h1
        self.w1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w1
        self.w2 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w2
        self.b1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_b1

        # Neuron h2
        self.w3 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w3
        self.w4 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w4
        self.b2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_b2

        # Neuron o1
        self.w5 -= learn_rate * d_L_d_ypred * d_ypred_d_w5
        self.w6 -= learn_rate * d_L_d_ypred * d_ypred_d_w6
        self.b3 -= learn_rate * d_L_d_ypred * d_ypred_d_b3

      # --- Calculate total loss at the end of each epoch
      if epoch % 10 == 0:
        y_preds = np.apply_along_axis(self.feedforward, 1, data)
        loss = mse_loss(all_y_trues, y_preds)
        print("Epoch %d loss: %.3f" % (epoch, loss))

# Define dataset
data = np.array([
  [-2, -1],  # Alice
  [25, 6],   # Bob
  [17, 4],   # Charlie
  [-15, -6], # Diana
])
all_y_trues = np.array([
  1, # Alice
  0, # Bob
  0, # Charlie
  1, # Diana
])

# Train our neural network!
network = OurNeuralNetwork()
network.train(data, all_y_trues)
```

```
# Make some predictions
emily = np.array([-7, -3]) # 128 pounds, 63 inches
frank = np.array([20, 2])  # 155 pounds, 68 inches
print("Emily: %.3f" % network.feedforward(emily)) # 0.951 - F
print("Frank: %.3f" % network.feedforward(frank)) # 0.039 - M
```

MNIST

MNIST

Wikipedia

https://en.wikipedia.org/wiki/MNIST_database

Dataset

http://yann.lecun.com/exdb/mnist/

MNIST - Simple Case
Preparing the dataset

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
X, y = mnist["data"], mnist["target"]
```

```
X = X / 255
```

```
import numpy as np

y_new = np.zeros(y.shape)
y_new[np.where(y == 0.0)[0]] = 1
y = y_new
```

```
m = 60000
m_test = X.shape[0] - m

X_train, X_test = X[:m].T, X[m:].T
y_train, y_test = y[:m].reshape(1,m), y[m:].reshape(1,m_test)
```

```
np.random.seed(138)
shuffle_index = np.random.permutation(m)
X_train, y_train = X_train[:,shuffle_index], y_train[:,shuffle_index]
```

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

i = 3
plt.imshow(X_train[:,i].reshape(28,28), cmap = matplotlib.cm.binary)
plt.axis("off")
plt.show()
print(y_train[:,i])
```

Forward Propagation

$$\hat{y} = \sigma(w^T x + b)$$

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

```
def neuron(W, X, b):
    return np.matmul(W.T, X) + b
```

```
def sigmoid(z):
    s = 1 / (1 + np.exp(-z))
    return s
```

Another Efficient Loss Function

Cost Function

$$L(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

$$L(Y, \hat{Y}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right)$$

```
def compute_loss(Y, Y_hat):

    m = Y.shape[1]
    L = -(1./m) * ( np.sum( np.multiply(np.log(Y_hat),Y) ) +
np.sum( np.multiply(np.log(1-Y_hat),(1-Y)) ) )

    return L
```

Backward Propagation

$$
\begin{aligned}
z &= w^T x + b, \\
\hat{y} &= \sigma(z), \\
L(y, \hat{y}) &= -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})
\end{aligned}
$$

$$
\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_j}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}} \left(-y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}) \right) \\
&= -y \frac{\partial}{\partial \hat{y}} \log(\hat{y}) - (1 - y) \frac{\partial}{\partial \hat{y}} \log(1 - \hat{y}) \\
&= \frac{-y}{\hat{y}} + \frac{(1 - y)}{1 - \hat{y}} \\
&= \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}.
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial}{\partial z}\sigma(z) &= \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right) \\
&= -\frac{1}{(1+e^{-z})^2}\frac{\partial}{\partial z}(1+e^{-z}) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}} \\
&= \sigma(z)\frac{e^{-z}}{1+e^{-z}} \\
&= \sigma(z)\left(1-\frac{1}{1+e^{-z}}\right) \\
&= \sigma(z)\left(1-\sigma(z)\right) \\
&= \hat{y}(1-\hat{y}).
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial}{\partial w_j}(w^T x + b) &= \frac{\partial}{\partial w_j}(w_0 x_0 + \ldots + w_n x_n + b) \\
&= w_j.
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial w_j} &= \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w_j} \\
&= \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}\hat{y}(1-\hat{y})w_j \\
&= (\hat{y}-y)w_j.
\end{aligned}
$$

$$
\frac{\partial L}{\partial w} = \frac{1}{m}X(\hat{y}-y)^T
$$

```
dW = (1/m) * np.matmul(X, (A-Y).T)
```

$$
\frac{\partial L}{\partial b} = (\hat{y}-y)
$$

$$
\frac{\partial L}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)}-y^{(i)})
$$

```
db = (1/m) * np.sum(A-Y, axis=1, keepdims=True)
```

```
learning_rate = 1

X = X_train
Y = y_train

n_x = X.shape[0]
m = X.shape[1]

W = np.random.randn(n_x, 1) * 0.01
b = np.zeros((1, 1))

for i in range(2000):
    Z = np.matmul(W.T, X) + b
    A = sigmoid(Z)

    cost = compute_loss(Y, A)

    dW = (1/m) * np.matmul(X, (A-Y).T)
    db = (1/m) * np.sum(A-Y, axis=1, keepdims=True)

    W = W - learning_rate * dW
    b = b - learning_rate * db

    if (i % 100 == 0):
        print("Epoch", i, "cost: ", cost)

print("Final cost:", cost)
```

```
from sklearn.metrics import classification_report, confusion_matrix

Z = np.matmul(W.T, X_test) + b
A = sigmoid(Z)

predictions = (A>.5)[0,:]
labels = (y_test == 1)[0,:]

print(confusion_matrix(predictions, labels))
```

```
[[8980   33]
 [  40  947]]
```

```
print(classification_report(predictions, labels))
```

```
             precision    recall  f1-score   support

      False       1.00      1.00      1.00      9013
       True       0.97      0.96      0.96       987

avg / total       0.99      0.99      0.99     10000
```

MNIST - One hidden layer

```
X = X_train
Y = y_train

n_x = X.shape[0]
n_h = 64
learning_rate = 1

W1 = np.random.randn(n_h, n_x)
b1 = np.zeros((n_h, 1))
W2 = np.random.randn(1, n_h)
b2 = np.zeros((1, 1))

for i in range(2000):

    Z1 = np.matmul(W1, X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.matmul(W2, A1) + b2
    A2 = sigmoid(Z2)

    cost = compute_loss(Y, A2)
```

```
    dZ2 = A2-Y
    dW2 = (1./m) * np.matmul(dZ2, A1.T)
    db2 = (1./m) * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.matmul(W2.T, dZ2)
    dZ1 = dA1 * sigmoid(Z1) * (1 - sigmoid(Z1))
    dW1 = (1./m) * np.matmul(dZ1, X.T)
    db1 = (1./m) * np.sum(dZ1, axis=1, keepdims=True)

    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1

    if i % 100 == 0:
        print("Epoch", i, "cost: ", cost)

print("Final cost:", cost)
```

Confusion Matrix

```
Z1 = np.matmul(W1, X_test) + b1
A1 = sigmoid(Z1)
Z2 = np.matmul(W2, A1) + b2
A2 = sigmoid(Z2)

predictions = (A2>.5)[0,:]
labels = (y_test == 1)[0,:]

print(confusion_matrix(predictions, labels))
print(classification_report(predictions, labels))
```

```
[[8984   36]
 [  36  944]]
             precision    recall  f1-score   support

      False       1.00      1.00      1.00      9020
       True       0.96      0.96      0.96       980

avg / total       0.99      0.99      0.99     10000
```

MNIST
Multi-label Classifier

```
mnist = fetch_mldata('MNIST original')
X, y = mnist["data"], mnist["target"]

X = X / 255
```

```
digits = 10
examples = y.shape[0]

y = y.reshape(1, examples)

Y_new = np.eye(digits)[y.astype('int32')]
Y_new = Y_new.T.reshape(digits, examples)
```

```
m = 60000
m_test = X.shape[0] - m

X_train, X_test = X[:m].T, X[m:].T
Y_train, Y_test = Y_new[:,:m], Y_new[:,m:]

shuffle_index = np.random.permutation(m)
X_train, Y_train = X_train[:, shuffle_index],
Y_train[:, shuffle_index]
```

Forward Propagation

Only the last layer of our network is changing.

Replace our lone, final node with a 10 unit layer.

Its final activations are the exponentials of its z-values, normalized across all ten such exponentials.

So instead of just computing σ(z),

compute the activation for each unit i :

$$\frac{e^{z_i}}{\sum_{j=0}^{9} e^{z_j}}$$

```
A2 = np.exp(Z2) / np.sum(np.exp(Z2), axis=0)
```

Cost Function

$$L(y, \hat{y}) = -\sum_{i=0}^{n} y_i \log(\hat{y}_i)$$

$$L(Y, \hat{Y}) = -\frac{1}{m} \sum_{j=0}^{m} \sum_{i=0}^{n} y_i^{(j)} \log(\hat{y}_i^{(j)})$$

```python
def compute_multiclass_loss(Y, Y_hat):

    L_sum = np.sum(np.multiply(Y, np.log(Y_hat)))
    m = Y.shape[1]
    L = -(1/m) * L_sum

    return L
```

Back Propagation

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i$$

MNIST
Multi-Class Classifier

```
n_x = X_train.shape[0]
n_h = 64
learning_rate = 1

W1 = np.random.randn(n_h, n_x)
b1 = np.zeros((n_h, 1))
W2 = np.random.randn(digits, n_h)
b2 = np.zeros((digits, 1))

X = X_train
Y = Y_train

for i in range(2000):

    Z1 = np.matmul(W1,X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.matmul(W2,A1) + b2
    A2 = np.exp(Z2) / np.sum(np.exp(Z2), axis=0)

    cost = compute_multiclass_loss(Y, A2)
```

```
    dZ2 = A2-Y
    dW2 = (1./m) * np.matmul(dZ2, A1.T)
    db2 = (1./m) * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.matmul(W2.T, dZ2)
    dZ1 = dA1 * sigmoid(Z1) * (1 - sigmoid(Z1))
    dW1 = (1./m) * np.matmul(dZ1, X.T)
    db1 = (1./m) * np.sum(dZ1, axis=1, keepdims=True)

    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1

    if (i % 100 == 0):
        print("Epoch", i, "cost: ", cost)

print("Final cost:", cost)
```

```
Z1 = np.matmul(W1, X_test) + b1
A1 = sigmoid(Z1)
Z2 = np.matmul(W2, A1) + b2
A2 = np.exp(Z2) / np.sum(np.exp(Z2), axis=0)

predictions = np.argmax(A2, axis=0)
labels = np.argmax(Y_test, axis=0)

print(confusion_matrix(predictions, labels))
print(classification_report(predictions, labels))
```

```
[[ 946    0   14    3    3   10   12    2    9    4]
 [   0 1112    3    2    1    1    2    8    3    4]
 [   3    4  937   24   10    7    8   18    8    3]
 [   4    2   17  924    1   39    4   13   26    9]
 [   0    1   10    0  905    9   11    9   10   40]
 [  12    5    2   26    3  786   15    3   24   14]
 [   8    1   19    2    9   10  902    1    9    1]
 [   2    1   13   14    3    5    1  946    9   25]
 [   5    9   16   11    5   18    3    5  868    9]
 [   0    0    1    4   42    7    0   23    8  900]]
              precision    recall  f1-score   support

           0       0.97      0.94      0.95      1003
           1       0.98      0.98      0.98      1136
           2       0.91      0.92      0.91      1022
           3       0.91      0.89      0.90      1039
           4       0.92      0.91      0.92       995
           5       0.88      0.88      0.88       890
           6       0.94      0.94      0.94       962
           7       0.92      0.93      0.92      1019
           8       0.89      0.91      0.90       949
           9       0.89      0.91      0.90       985

avg / total       0.92      0.92      0.92     10000
```

Let’s code a MNIST Neural Network

150 - MNIST Multi-class Neural Network

Improved MNIST

```
from sklearn.datasets import fetch_mldata
from sklearn.metrics import classification_report, confusion_matrix
import numpy as np

# import
mnist = fetch_mldata('MNIST original')
X, y = mnist["data"], mnist["target"]

# scale
X = X / 255

# one-hot encode labels
digits = 10
examples = y.shape[0]
y = y.reshape(1, examples)
Y_new = np.eye(digits)[y.astype('int32')]
Y_new = Y_new.T.reshape(digits, examples)

# split, reshape, shuffle
m = 60000
m_test = X.shape[0] - m
X_train, X_test = X[:m].T, X[m:].T
Y_train, Y_test = Y_new[:,:m], Y_new[:,m:]
shuffle_index = np.random.permutation(m)
X_train, Y_train = X_train[:, shuffle_index], Y_train[:, shuffle_index
```

```python
def sigmoid(z):
    s = 1. / (1. + np.exp(-z))
    return s

def compute_loss(Y, Y_hat):

    L_sum = np.sum(np.multiply(Y, np.log(Y_hat)))
    m = Y.shape[1]
    L = -(1./m) * L_sum

    return L
```

```python
def feed_forward(X, params):

    cache = {}

    cache["Z1"] = np.matmul(params["W1"], X) + params["b1"]
    cache["A1"] = sigmoid(cache["Z1"])
    cache["Z2"] = np.matmul(params["W2"], cache["A1"]) + params["b2"]
    cache["A2"] = np.exp(cache["Z2"]) / np.sum(np.exp(cache["Z2"]), axis=0)

    return cache
```

```python
def back_propagate(X, Y, params, cache):

    dZ2 = cache["A2"] - Y
    dW2 = (1./m_batch) * np.matmul(dZ2, cache["A1"].T)
    db2 = (1./m_batch) * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.matmul(params["W2"].T, dZ2)
    dZ1 = dA1 * sigmoid(cache["Z1"]) * (1 - sigmoid(cache["Z1"]))
    dW1 = (1./m_batch) * np.matmul(dZ1, X.T)
    db1 = (1./m_batch) * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2}

    return grads
```

Improvements

1. Mini-batch gradient descent
2. Momentum
3. Weights Initialization

#1 Mini-batch gradient descent

Add another for loop inside the pass through each epoch.

At each pass randomly shuffle the training set, then iterate through it in chunks of batch_size, which will be arbitrarily set to 128.

#2 Momentum

To add momentum, keep a moving average of gradients.

So instead of updating our parameters like this below :

```
params["W1"] = params["W1"] - learning_rate * grads["dW1"]
```

Do this :

```
V_dW1 = (beta * V_dW1 + (1. - beta) * grads["dW1"])
params["W1"] = params["W1"] - learning_rate * V_dW1
```

#3 Weights Initialization

Shrink the variance of the weights in each layer.

Set the variance for each layer to 1 / n

`np.random.randn()` function draws from the standard normal distribution.

So to adjust the variance to 1 / n, divide by sqrt(n)

```
np.random.randn(n_h, n_x) * np.sqrt(1. / n_x)
```

Putting all things together

```
np.random.seed(138)

# hyperparameters
n_x = X_train.shape[0]
n_h = 64
learning_rate = 4
beta = .9
batch_size = 128
batches = -(-m // batch_size)

# initialization
params = { "W1": np.random.randn(n_h, n_x) * np.sqrt(1. / n_x),
           "b1": np.zeros((n_h, 1)) * np.sqrt(1. / n_x),
           "W2": np.random.randn(digits, n_h) * np.sqrt(1. / n_h),
           "b2": np.zeros((digits, 1)) * np.sqrt(1. / n_h) }

V_dW1 = np.zeros(params["W1"].shape)
V_db1 = np.zeros(params["b1"].shape)
V_dW2 = np.zeros(params["W2"].shape)
V_db2 = np.zeros(params["b2"].shape)
```

```
# train
for i in range(9):

    permutation = np.random.permutation(X_train.shape[1])
    X_train_shuffled = X_train[:, permutation]
    Y_train_shuffled = Y_train[:, permutation]

    for j in range(batches):

        begin = j * batch_size
        end = min(begin + batch_size, X_train.shape[1] - 1)
        X = X_train_shuffled[:, begin:end]
        Y = Y_train_shuffled[:, begin:end]
        m_batch = end - begin

        cache = feed_forward(X, params)
        grads = back_propagate(X, Y, params, cache)

        V_dW1 = (beta * V_dW1 + (1. - beta) * grads["dW1"])
        V_db1 = (beta * V_db1 + (1. - beta) * grads["db1"])
        V_dW2 = (beta * V_dW2 + (1. - beta) * grads["dW2"])
        V_db2 = (beta * V_db2 + (1. - beta) * grads["db2"])

        params["W1"] = params["W1"] - learning_rate * V_dW1
        params["b1"] = params["b1"] - learning_rate * V_db1
        params["W2"] = params["W2"] - learning_rate * V_dW2
        params["b2"] = params["b2"] - learning_rate * V_db2
```

```
    cache = feed_forward(X_train, params)
    train_cost = compute_loss(Y_train, cache["A2"])
    cache = feed_forward(X_test, params)
    test_cost = compute_loss(Y_test, cache["A2"])
    print("Epoch {}: training cost = {}, test cost = {}".format(i+1 ,train_cost, test_cost))

print("Done.")
```

98% accuracy

```
cache = feed_forward(X_test, params)
predictions = np.argmax(cache["A2"], axis=0)
labels = np.argmax(Y_test, axis=0)

print(classification_report(predictions, labels))
```

```
             precision    recall  f1-score   support

          0       0.99      0.98      0.98       984
          1       0.99      0.99      0.99      1136
          2       0.98      0.98      0.98      1037
          3       0.97      0.98      0.97      1004
          4       0.97      0.98      0.98       970
          5       0.97      0.96      0.97       900
          6       0.97      0.99      0.98       942
          7       0.97      0.97      0.97      1026
          8       0.97      0.96      0.97       982
          9       0.97      0.96      0.97      1019

avg / total       0.98      0.98      0.98     10000
```

Let’s code Improved MNIST Neural Network

200 - Improved MNIST Neural Network

Deep Neural Network

CNN