Ch. 5. Neural Network and Deep Learning

Yongjae Yoo, Ph. D.
Assistant Professor
Department of Artificial Intelligence,
Hanyang University ERICA

한양대학교 ERICA
Education Research Industry Cluster @ Ansan

So Far,

- We have learnt how your computer understand the data and make decisions.
- (In the simplest ways.)

- Based on the data, the machine can identify
  - The trends of data
  - Which group the new input belongs to

What If

- The question goes “deep,” i.e.,
  - Questions gets sophisticated – multiple times we should ask.
  - Questions have many possibilities
  - New point (input) has many dimensions
  - New point (input) has totally new one.

- Then, the basic approach goes impossible.

- In most cases in real life, we have such issues.

Examples

- Coke vs. Pepsi
  - So many ingredients, answers diverge.

- Conditional/confounded plans
  - “It depends on whether my friends’ opinions.”
  - We eat 빵. Turkish eat Ekmek. Chinese eat Mian Bao.
  - What will you do after this class?
  - Etc.

Dealing With Many Questions

- Nested If statements

```
x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

- With and/or/not (binary conditional operators)

Binary Conditional Operators

- And, and OR

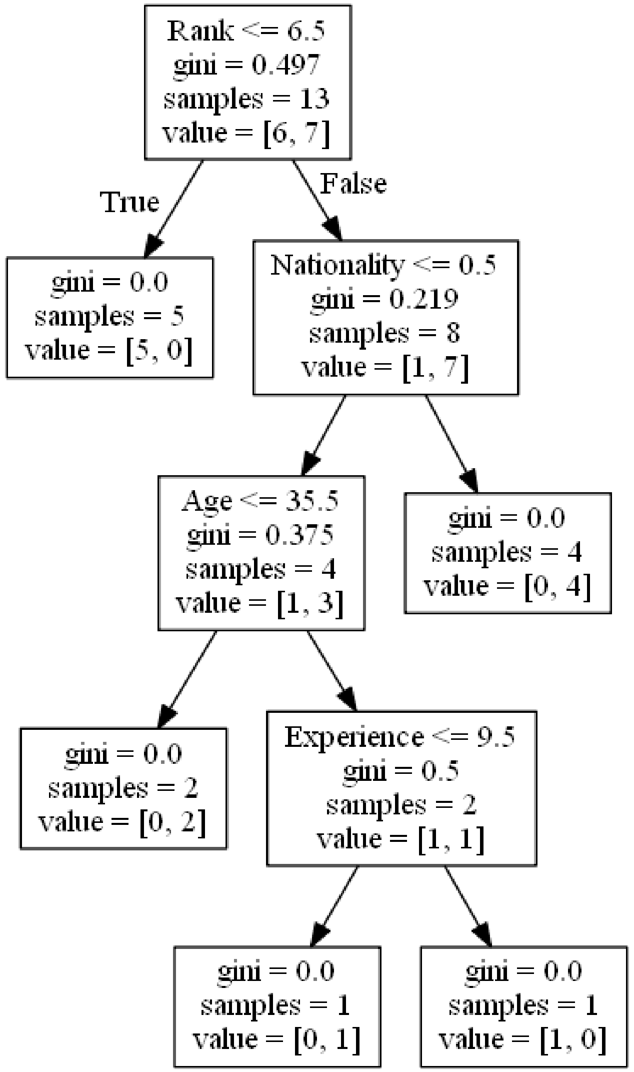
```python
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

```python
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

Decision Tree

- Dealing with multiple conditions in a systematic way.

- A series of true-false questions, and the answers will be decided by the sequential inputs.



Decision Tree

- Example:
  - “I am trying to decide if I should go to a comedy show or not.”
- Fortunately, I know the comedian’s experience, age, nationality, rank, and my preference (Yay or Nay).

Age	Experience	Rank	Nationality	Go
36	10	9	UK	NO
42	12	4	USA	NO
23	4	6	N	NO
52	4	4	USA	NO
43	21	8	USA	YES
44	14	5	UK	NO
66	3	7	N	YES
35	14	9	UK	YES
52	13	7	N	YES
35	5	9	N	YES
24	3	5	USA	NO
18	3	7	UK	YES
45	9	9	UK	YES

Python Codes

- We will use a new tool, Pandas in this example.

```
import pandas

df = pandas.read_csv("data.csv")

print(df)
```

- Then, we need to convert the text data into numbers.
- We will use Pandas' map() method.

```
{'UK': 0, 'USA': 1, 'N': 2}
```

Python Codes

- Mapping texts into numbers:

```
d = {'UK': 0, 'USA': 1, 'N': 2}
df['Nationality'] = df['Nationality'].map(d)
d = {'YES': 1, 'NO': 0}
df['Go'] = df['Go'].map(d)

print(df)
```

- Then,

Python Codes

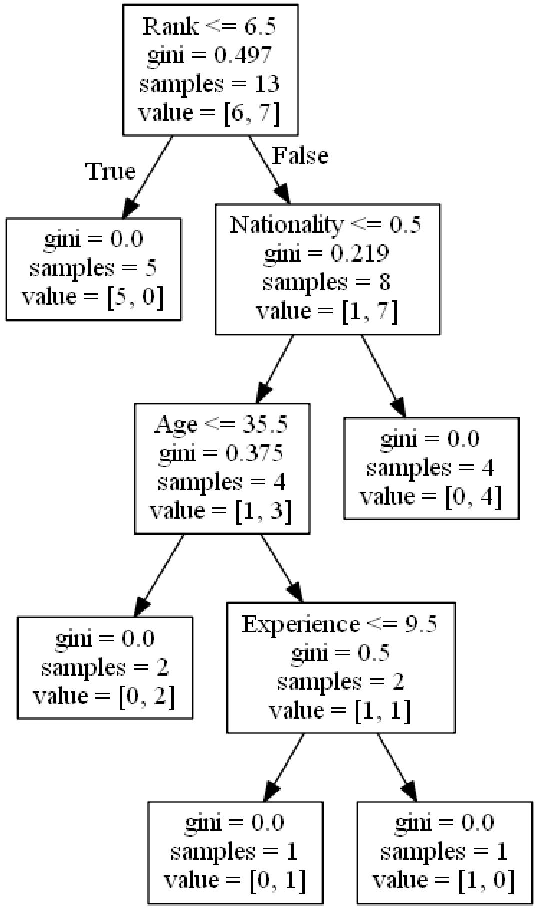
- Then, we will extract "*features*" to make decisions

```
features = 
['Age', 'Experience', 'Rank', 'Nationality']

X = df[features]
y = df['Go']

print(X)
print(y)
```

Final Decision Tree Here:

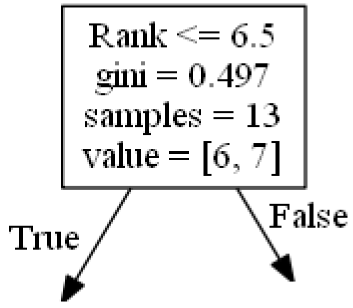
```
import pandas
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

df = pandas.read_csv("data.csv")
d = {'UK': 0, 'USA': 1, 'N': 2}
df['Nationality'] = df['Nationality'].map(d)
d = {'YES': 1, 'NO': 0}
df['Go'] = df['Go'].map(d)

features = ['Age', 'Experience', 'Rank', 'Nationality']
X = df[features]
y = df['Go']

dtree = DecisionTreeClassifier()
dtree = dtree.fit(X, y)
tree.plot_tree(dtree, feature_names=features)
```

Results


Rank <= 6.5
gini = 0.497
samples = 13
value = [6, 7]
True
False
gini = 0.0
samples = 5
value = [5, 0]
Nationality <= 0.5
gini = 0.219
samples = 8
value = [1, 7]
Age <= 35.5
gini = 0.375
samples = 4
value = [1, 3]
gini = 0.0
samples = 4
value = [0, 4]
gini = 0.0
samples = 2
value = [0, 2]
Experience <= 9.5
gini = 0.5
samples = 2
value = [1, 1]
gini = 0.0
samples = 1
value = [0, 1]
gini = 0.0
samples = 1
value = [1, 0]

Explaining Each Level

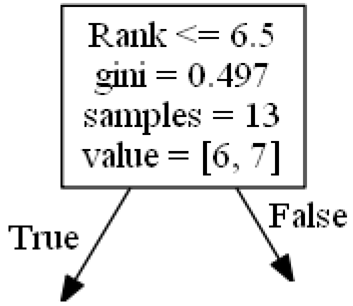
- It shows the criteria of decision on each
- First, see “rank.”
  - Rank <= 6.5 means that every comedian with a rank of 6.5 or lower will follow the True arrow (to the left), and the re st will follow the False arrow (to the right).



- Gini is the split quality – between 0 – 0.5
  - $Gini = 1 - (x/n)^2 - (y/n)^2$
    Where x is the number of positive answers("GO") , n is the number of samples, and y is the number of negative answers ("NO"), which gives us this calcula tion: $1 - (7 / 13)^2$

Explaining Each Level

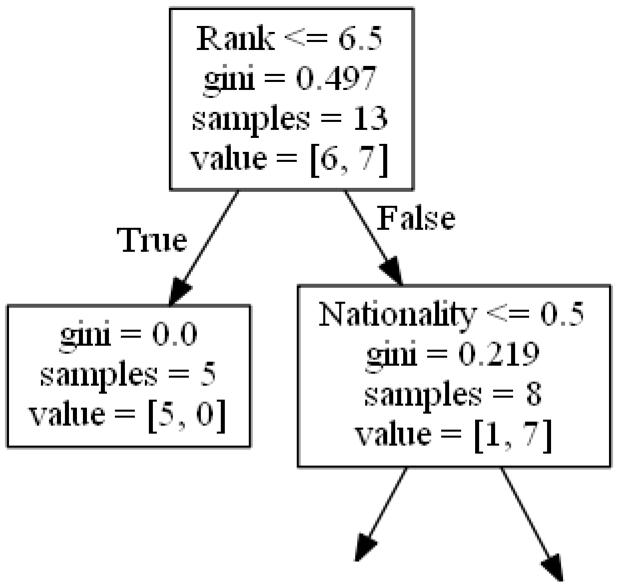
- `samples = 13` means that there are 13 comedians left at this point in the decision, which is all of them since this is the first step.
- `value = [6, 7]` means that of these 13 comedians, 6 will get a "NO", and 7 will get a "GO".



2nd Level

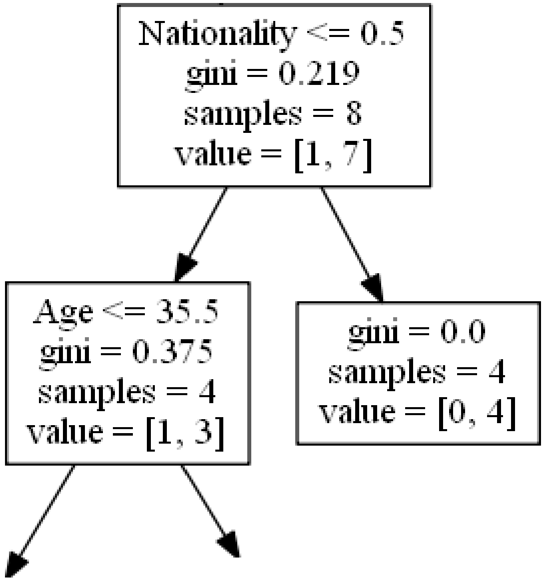
- True: five comedians will end here.
- (five samples, and five “NO”s.

- False:
  - Then the next way is checking nationality.
  - Among 8 samples, 1 No and 7 Yeses.
  - Gini would be 0.219.



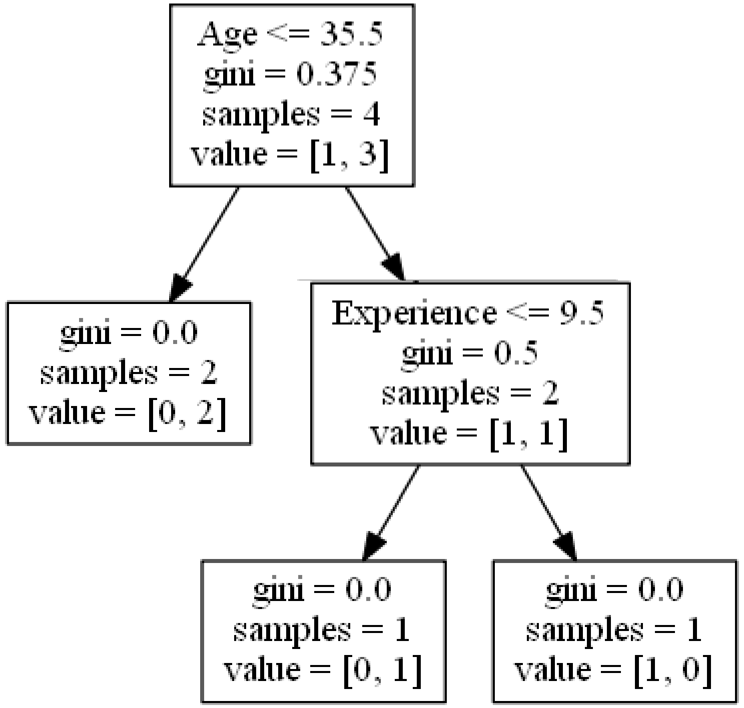
3rd Level

- True - 4 Comedians Continue:
  - `Age <= 35.5` means that comedians at the age of 35.5 or younger will follow the arrow to the left, and the rest will follow the arrow to the right.
  - `gini = 0.375` means that about 37.5% of the samples would go in one direction.
  - `samples = 4` means that there are 4 comedians left in this branch (4 comedians from the UK).
  - `value = [1, 3]` means that of these 4 comedians, 1 will get a "NO" and 3 will get a "GO".
- False - 4 Comedians End Here:
  - `gini = 0.0` means all of the samples got the same result.
  - `samples = 4` means that there are 4 comedians left in this branch (4 comedians not from the UK).
  - `value = [0, 4]` means that of these 4 comedians, 0 will get a "NO" and 4 will get a "GO".



4th and 5th levels

- In 4th level, it checks age.
  - If younger than 35.5, 2 samples will end with "Yes."
  - Otherwise, it will check experience.
    If one's experience is shorter than 9.5 years, it ends up to "No."
    Otherwise, goes to "Yes."



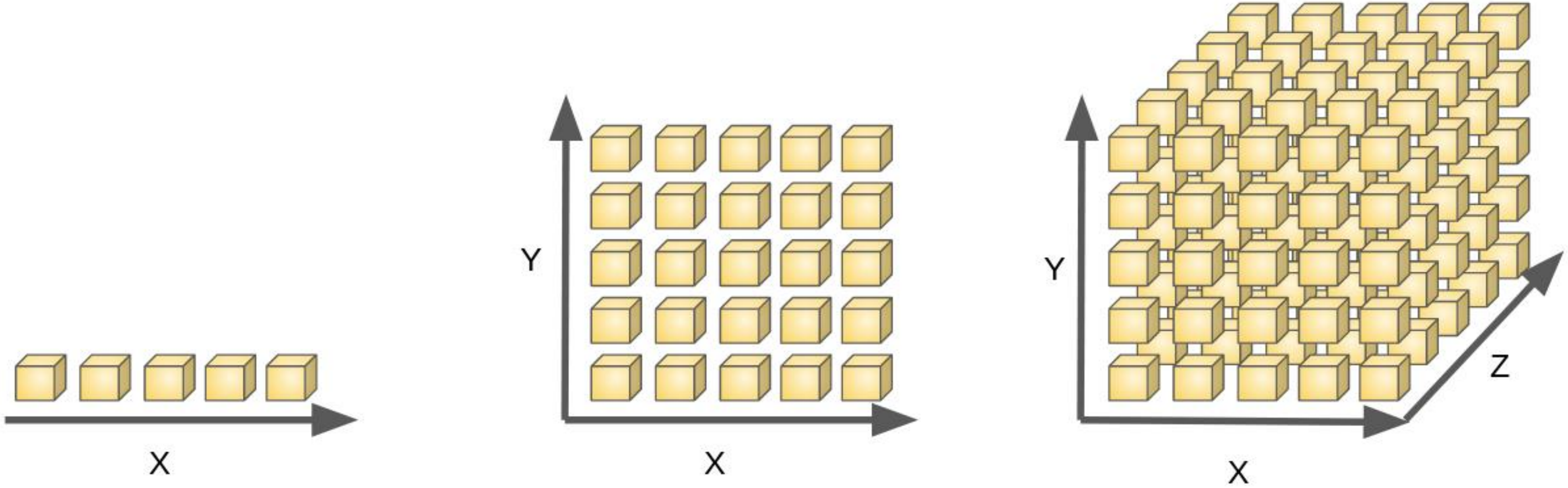
So… Is The Decision Tree Always Works?

- Testing with different samples:
  - `print(dtree.predict([[40, 10, 7, 1]]))`
  - `print(dtree.predict([[40, 10, 6, 1]]))`

- You will see that the Decision Tree gives you different results, even you put in the same data when you try multiple times.
- That is because the Decision Tree is based on the probability of an outcome, and the answer will vary.

Curse of Dimensionality

- If we set multiple domain(s) – variables,
- The possibility and the number of calculation explodes.
- In your cases of



Anyhow Let's Do This

- Try the example.

- If you have any good data, you can try that with your data for your final report.

Dealing with Complexity

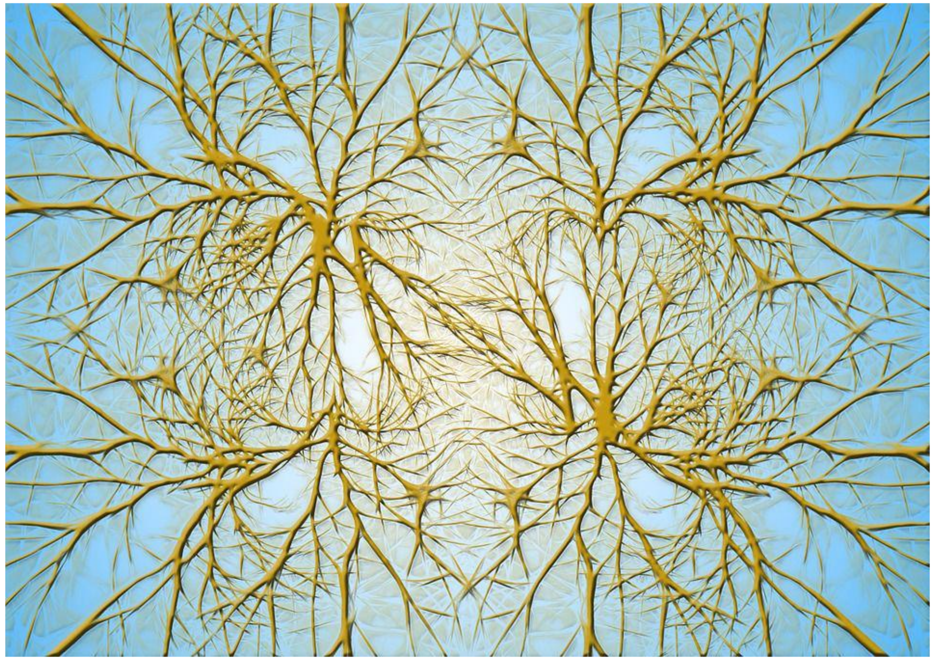
- Once again, computer can deal with yes/no questions.
- In other words, it can answer "data with linearity."

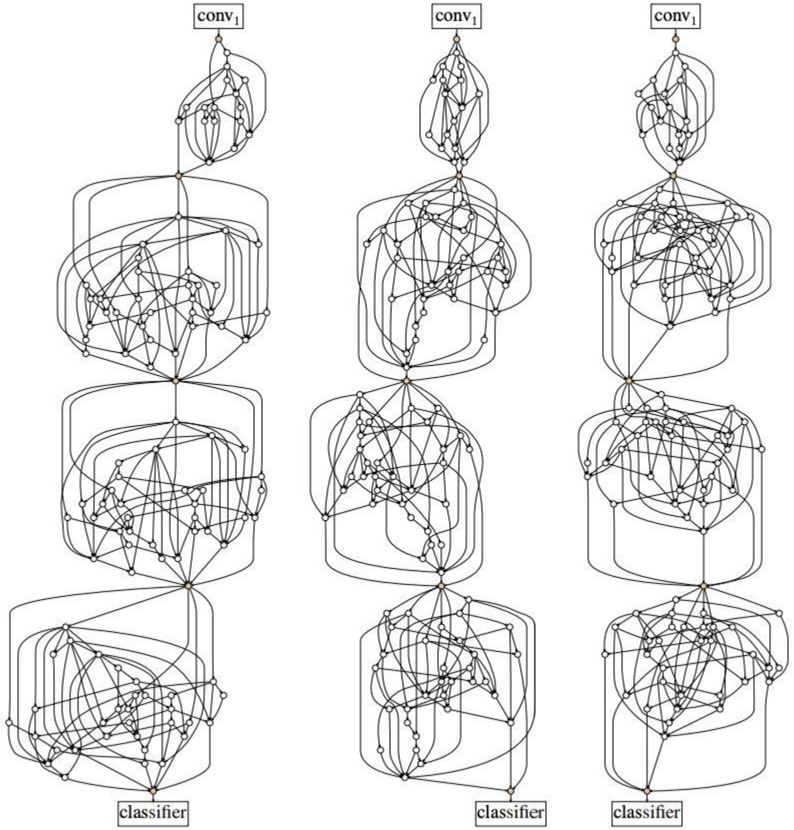
- In theory, we can deal with MANY linear combinations of que stions if we have enough memory. That is so-called Linear score function.

- To resolve this, we will let computer think as like we do.
- Neural network!

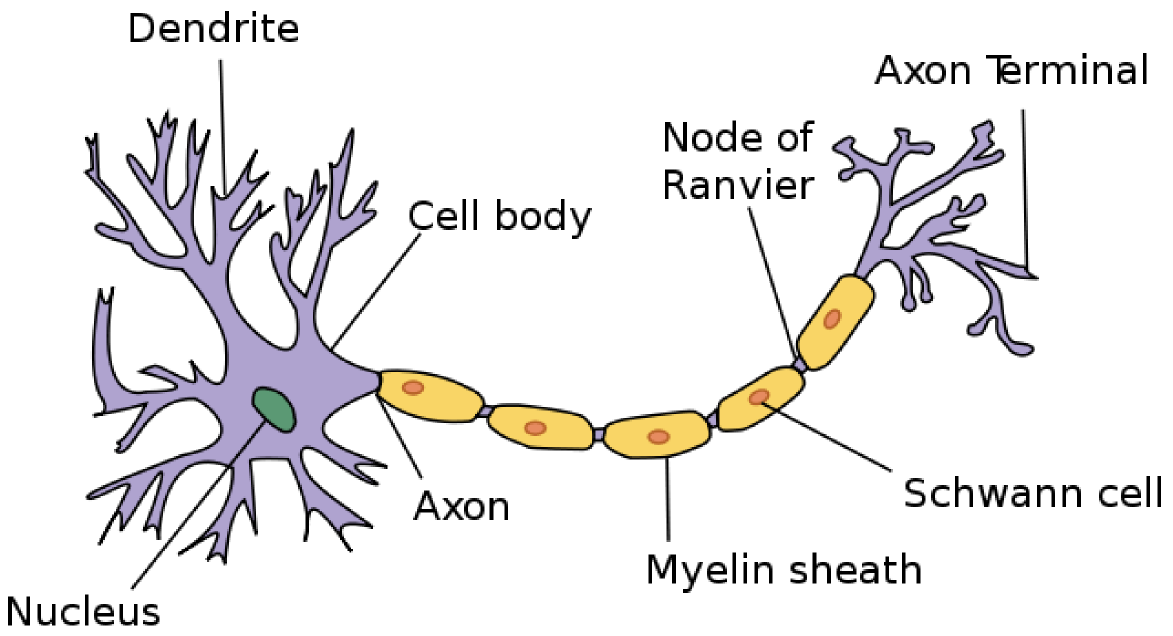
Analogy to (Physiologic) Neurons

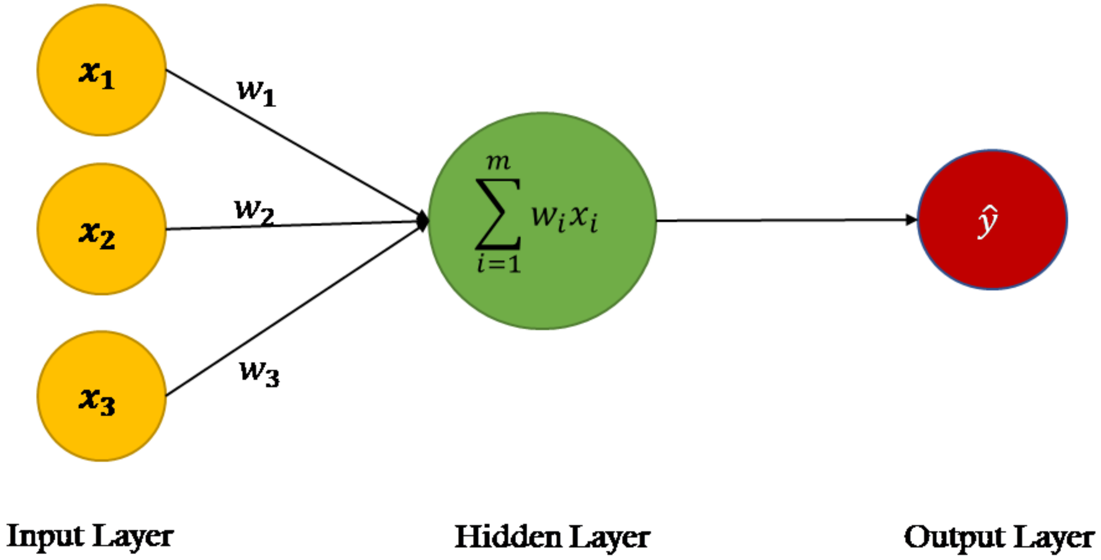
- Physical neural networks vs. computational neural networks.





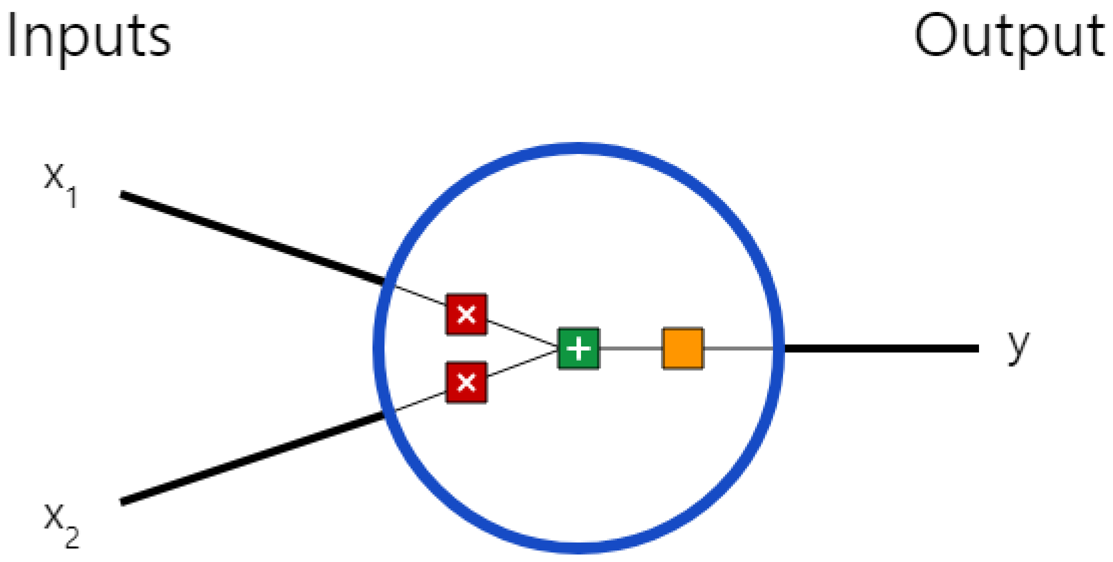
Looking into Single Neuron





Neuron as The Basic Building Block

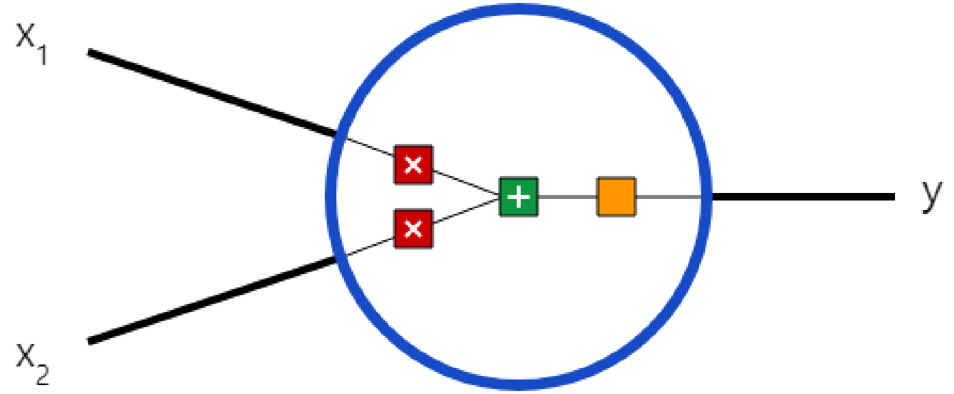
- A neuron takes inputs, does some math with them, and produces one output.
- 2-input neuron:



In a Neuron:

Inputs

Output



3 things are happening here. First, each input is multiplied by a weight:

$$x_1 \rightarrow x_1 * w_1$$

$$x_2 \rightarrow x_2 * w_2$$

Next, all the weighted inputs are added together with a bias b:

$$(x_1 * w_1) + (x_2 * w_2) + b$$

Finally, the sum is passed through an activation function:

$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$

Activation function

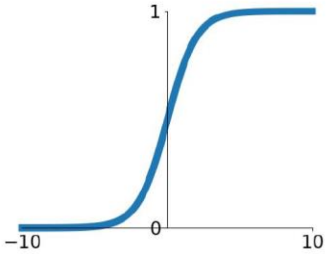
- Works as a switch – it cuts out conditions we do not want.

$$f = W_2 \boxed{\max(0,} W_1 x)$$

Activation Functions

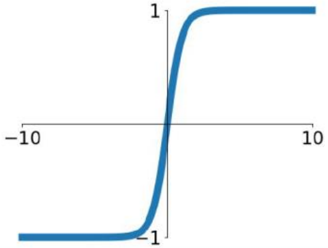
Sigmoid

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



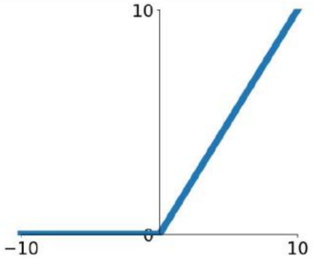
tanh

$\tanh(x)$



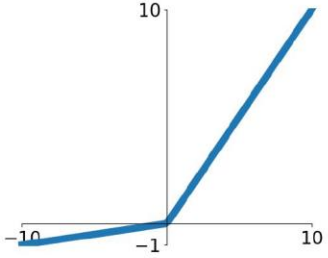
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

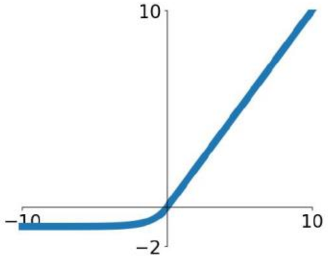


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

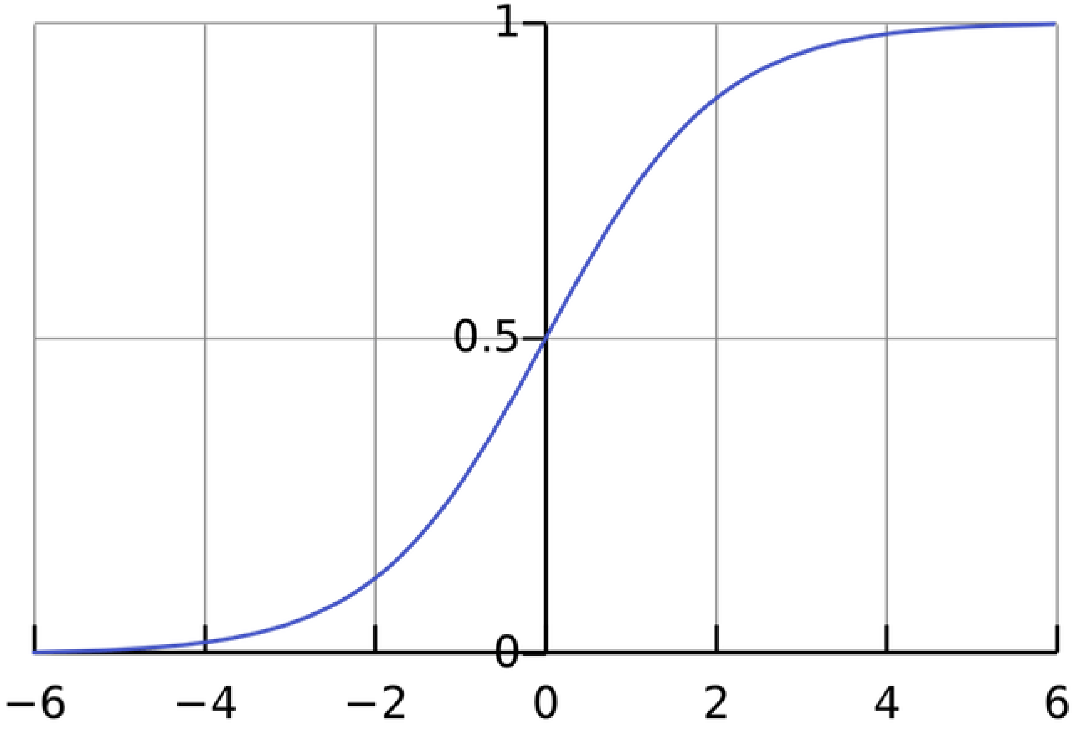
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Rectified Linear Unit (ReLU) often used as Default

Activation Function

- Consider a sigmoid function
  - Converts –inf to +inf to (0, 1)



Example

- Assume we have a 2-input neuron that uses the sigmoid activation function and has the following parameters:

$$w = [0, 1]$$

$$b = 4$$

- $w = [0, 1]$: $w_1 = 0, w_1 = 1$.
- Now give an input $x = [2, 3]$. Applying a dot product:

$$\begin{aligned}(w \cdot x) + b &= ((w_1 * x_1) + (w_2 * x_2)) + b \\ &= 0 * 2 + 1 * 3 + 4 \\ &= 7\end{aligned}$$

$$y = f(w \cdot x + b) = f(7) = \boxed{0.999}$$

A Single Neuron's Output

- Gives you a single output here – 0.999, as a "feedforward" result.
- Feedforward process means:
  passing inputs forward to get an output

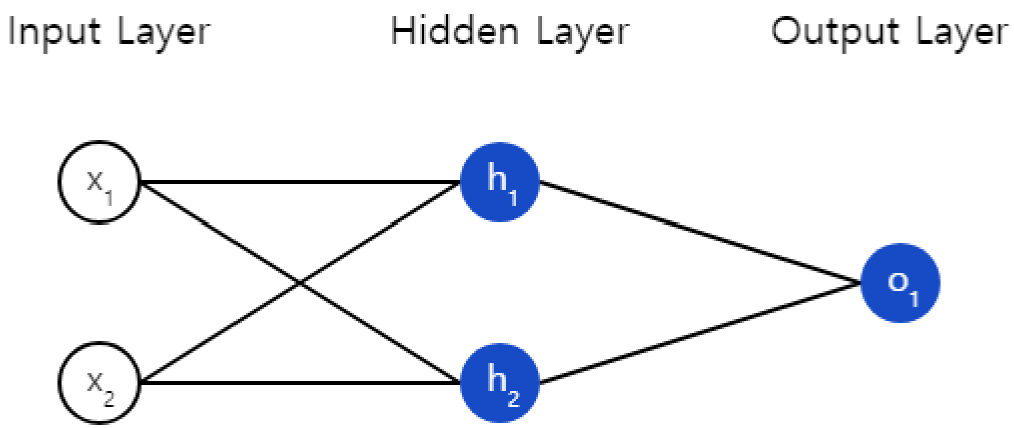
Now, Coding a Neuron Here:

- Here just we implemented example converted to code:

```
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
```

So, What’s The Difference?

- Neural network can be concatenated, like:



- This network has 2 inputs, a hidden layer with 2 neurons h1 and h2, a nd an output layer with 1 neuron (o1).
- Notice that the inputs for o1 are the outputs from $h1$ and $h2$ - that’s what makes this a network.

Feedforwarding the Network

- A neural network can have any number of layers with any number of ne urons in those layers.
- The basic idea stays the same: feed the input(s) forward through the ne urons in the network to get the output(s) at the end.

What happens if we pass in the input $x = [2, 3]$?

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

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

Let’s Implement a 2-Layer Network

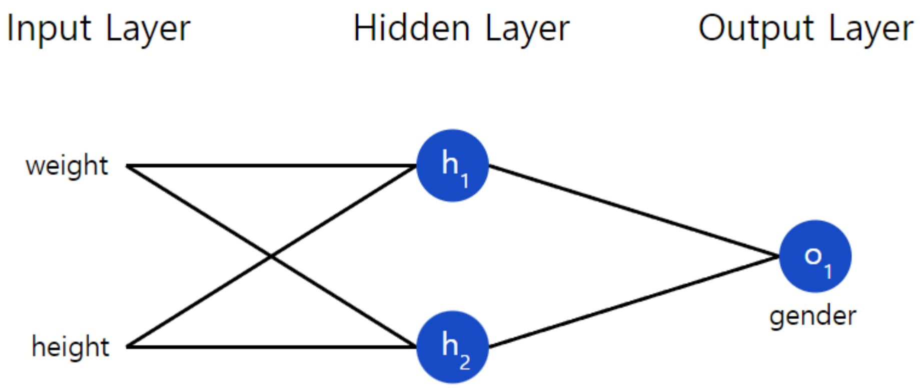
```
import numpy as np # ... code from previous section here

class OurNeuralNetwork:
  ''' A neural network with:
    - 2 inputs
    - a hidden layer with 2 neurons (h1, h2)
    - an output layer with 1 neuron (o1)
    - Each neuron has the same weights and bias:
      - w = [0, 1]
      - b = 0 '''

   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
```

So, More about Training

- Assume we have the following measurements:

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

- Let's train our network to predict someone's gender given th eir weight and height:



So, More about Training

- We'll represent Male with a 0 and Female with a 1, and we'll a lso shift the data to make it easier to use:

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

> *I arbitrarily chose the shift amounts (135 and 66) to make the numbers look nice. Normally, you'd shift by the mean.*

Understanding Loss

- Before we train our network, we first need a way to quantify how "good" it's doing so that it can try to do "better".
- That's what the loss is.

We'll use the **mean squared error** (MSE) loss:

the **squared error**.

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

- n is the number of samples, which is 4 (Alice, Bob, Charlie, Diana).
- y represents the variable being predicted, which is Gender.
- y_{true} is the *true* value of the variable (the "correct answer"). For example, y_{true} for Alice would be 1 (Female).
- y_{pred} is the *predicted* value of the variable. It's whatever our network outputs.

Understanding Loss

- The Keys:
  - Better predictions = Lower loss.
  - Training a network = trying to minimize its loss.

Example

An Example Loss Calculation

Let's say our network always outputs 0 - in other words, it's confident all humans are Male 🤔. What would our loss be?

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

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

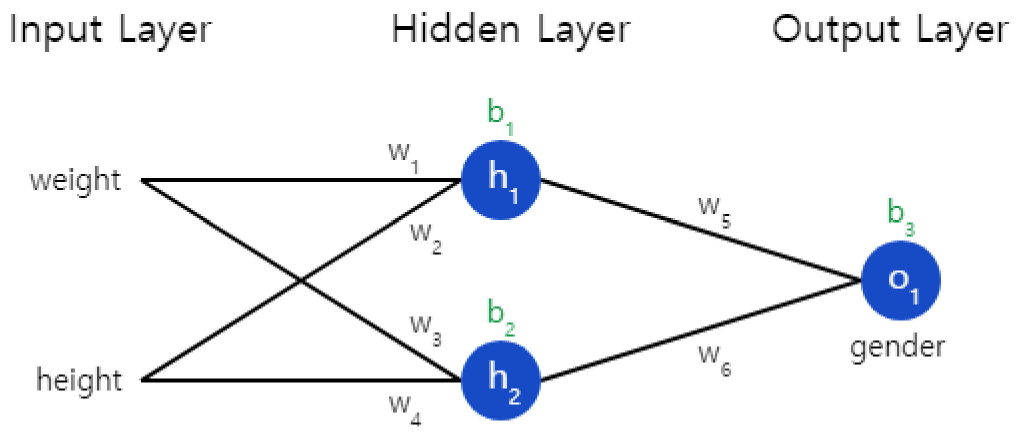
Here MSE Loss Code

- We can convert the procedure as code.

```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
```

Getting A Little Bit More Difficult

- So-called Backprop
- We now have a clear goal: minimize the loss of the neural network.
- We will skip the details of math here, but general way of upda ting the parameters here.



BackProp - Mathematics

To start, let's rewrite the partial derivative in terms of $\frac{\partial y_{pred}}{\partial w_1}$ instead:

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

This works because of the Chain Rule.

We can calculate $\frac{\partial L}{\partial y_{pred}}$ because we computed $L = (1 - y_{pred})^2$ above:

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

Now, let's figure out what to do with $\frac{\partial y_{pred}}{\partial w_1}$. Just like before, let h_1, h_2, o_1 be the outputs of the neurons they represent. Then

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

f is the sigmoid activation function, remember?

BackProp - Mathematics

Since w_1 only affects h_1 (not h_2), we can write

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

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

More Chain Rule.

We do the same thing for $\frac{\partial h_1}{\partial w_1}$:

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

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

You guessed it, Chain Rule.

x_1 here is weight, and x_2 is height. This is the second time we've seen $f'(x)$ (the derivate of the sigmoid function) now! Let's derive it:

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

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

BackProp - Mathematics

We're done! We've managed to break down $\frac{\partial L}{\partial w_1}$ into several parts we can calculate:

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

We will use this derived equation
For actual calculation ☺

This system of calculating partial derivatives by working backwards is known as **backpropagation**, or "backprop".

Training: Stochastic Gradient Descent

- We'll use an optimization algorithm called stochastic gradient descent (SGD) that tells us how to change our weights and biases to minimize loss.
- It's basically just this update equation:

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

- η: learning rate

- 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.

Just Jumping into Calculations

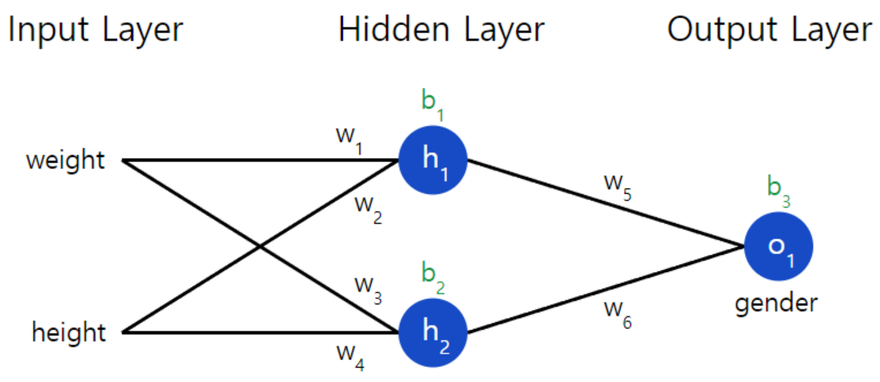
- Choose a single item (stochastic) and try to learn from data.

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.

Let's Make a Complete Implementation

It's *finally* time to implement a complete neural network:

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



Codes (1)

```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()
```

Codes (2)

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

Codes (3)

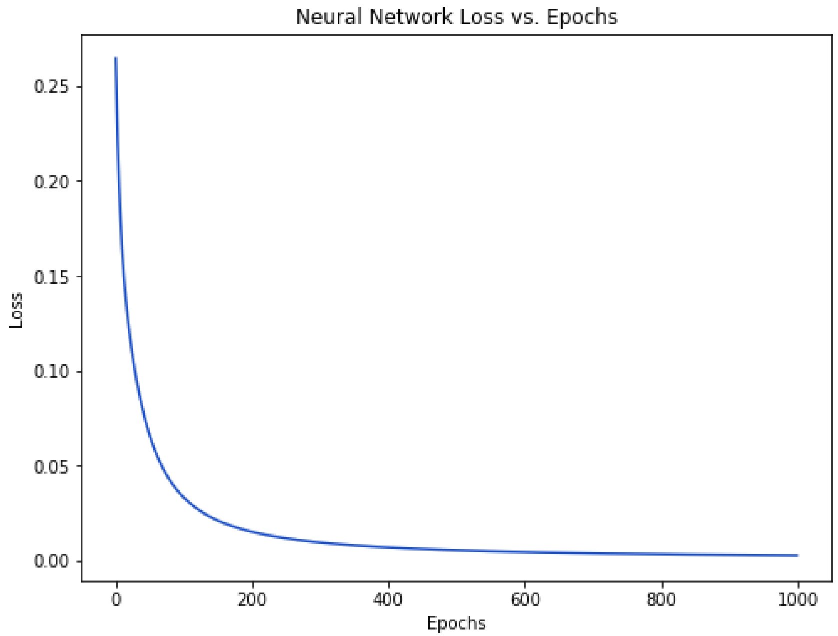
```
def train(self, data, all_y_trues):
  ''' - data is a (n x 2) numpy array, n = # of samples in the dataset.
  - all_y_trues is a numpy array with n elements.
  Elements in all_y_trues correspond to those in data. ‘‘’

  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))
```

Codes (4)

```
# 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)
```

- Training decreases the loss



Making Actual Prediction

```
# 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
```

More Resource

- Victor Zhou’s blog page on easy introduction to NN

- https://victorzhou.com/blog/intro-to-neural-networks/#code-a-complete-neural-network
- https://replit.com/@vzhou842/An-Introduction-to-Neural-Networks

Lab Session

- Follow the whole code and try to understand.

- Consider which of prediction you can make with this.
    - What is your major?

Dataset Treasure Island: Kaggle

- www.kaggle.com