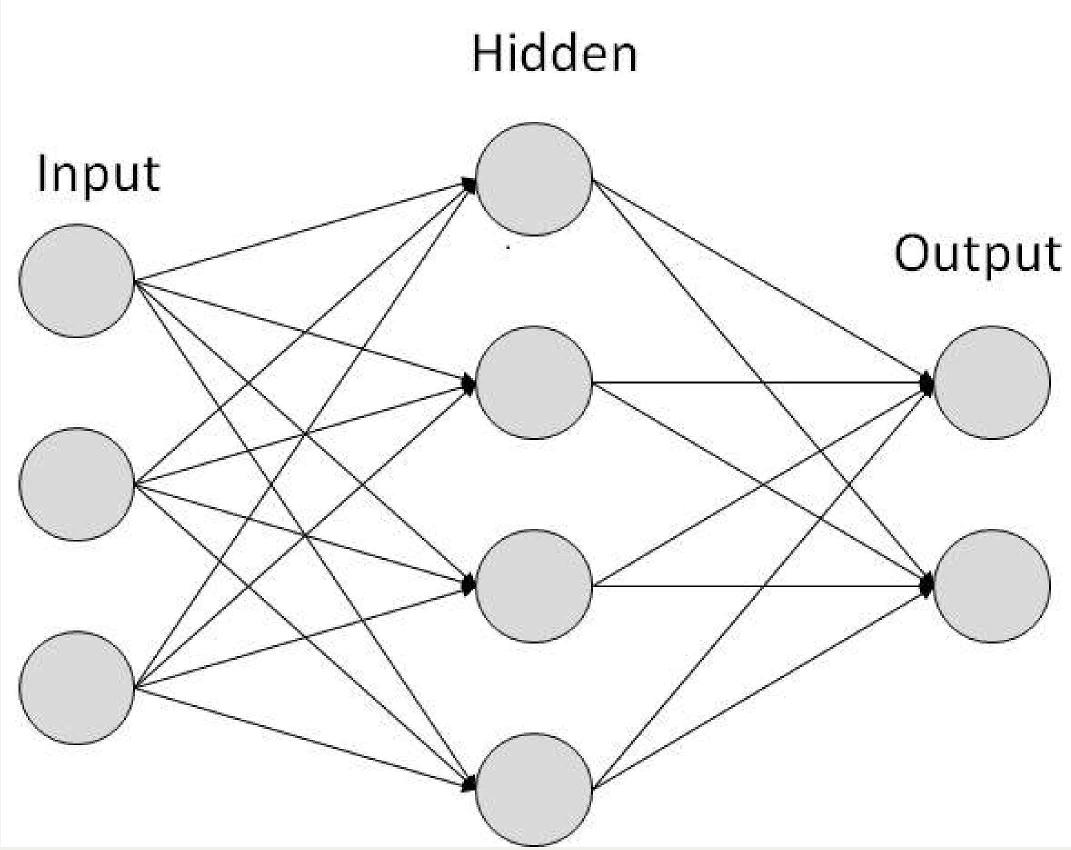
Lecture 11: Using Python for Artificial Intelligence

CS5001 / CS5003:
Intensive Foundations
of Computer Science



PDF of this presentation

Lecture 11: Using Python for Artificial Intelligence

Today's topics:

Introduction to Artificial Intelligence

Introduction to Artificial Neural Networks

Examples of some basic neural networks

Using Python for Artificial Intelligence

Example: PyTorch

Lecture 11: Introduction to Artificial Intelligence

Video Introduction

1950: Alan Turing: Turing Test

1951: First AI program

1965: Eliza (first chat bot)

1974: First autonomous vehicle

1997: Deep Blue beats Gary Kasimov at Chess

2004: First Autonomous Vehicle challenge

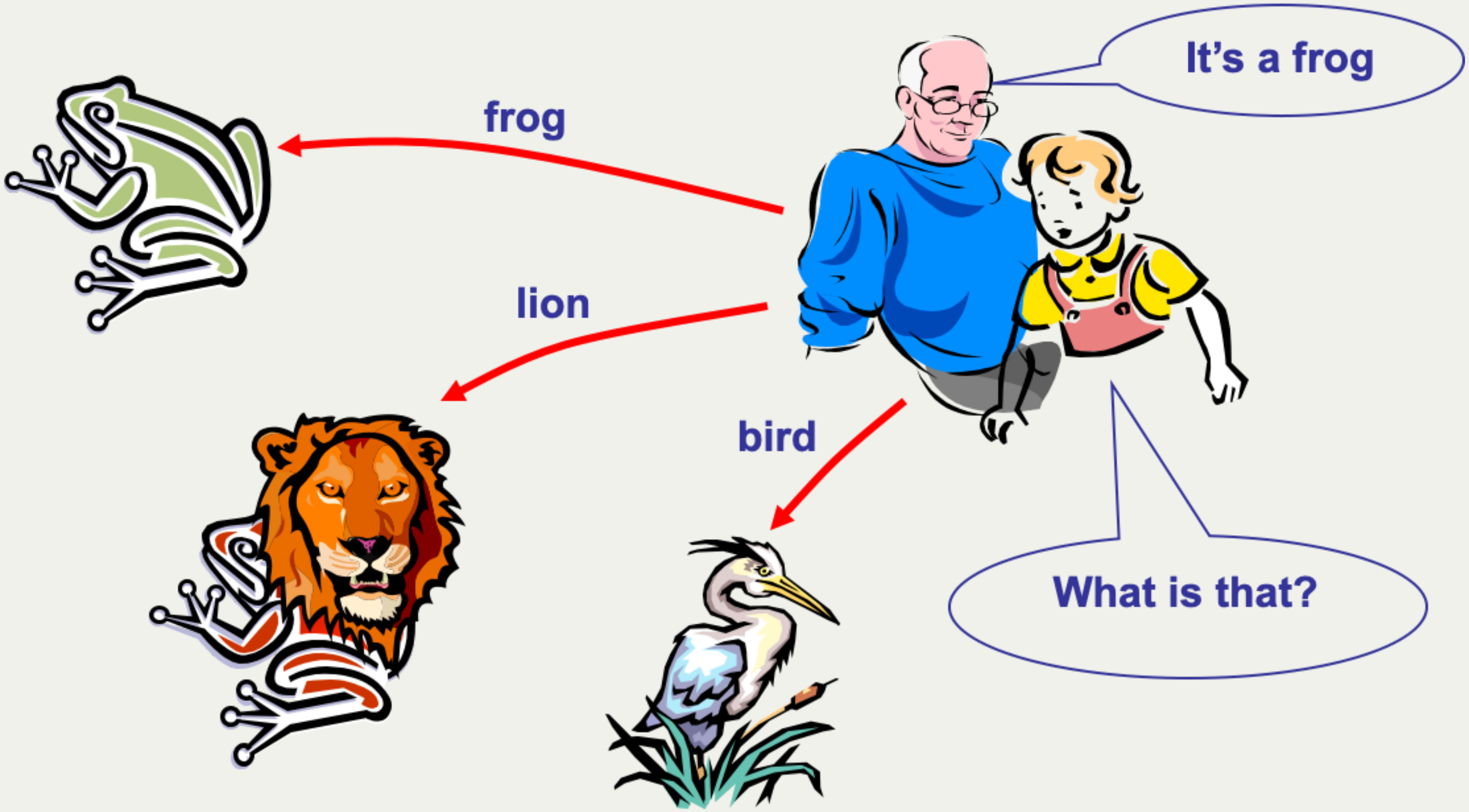
2011: IBM Watson beats Jeopardy winners

2016: Deep Mind beats Go champion

2017: AlphaGo Zero beats Deep Mind

Lecture 11: Introduction to Artificial Neural Networks (ANNs)

NNs learn relationship between cause and effect or organize large volumes of data into orderly and informative patterns.



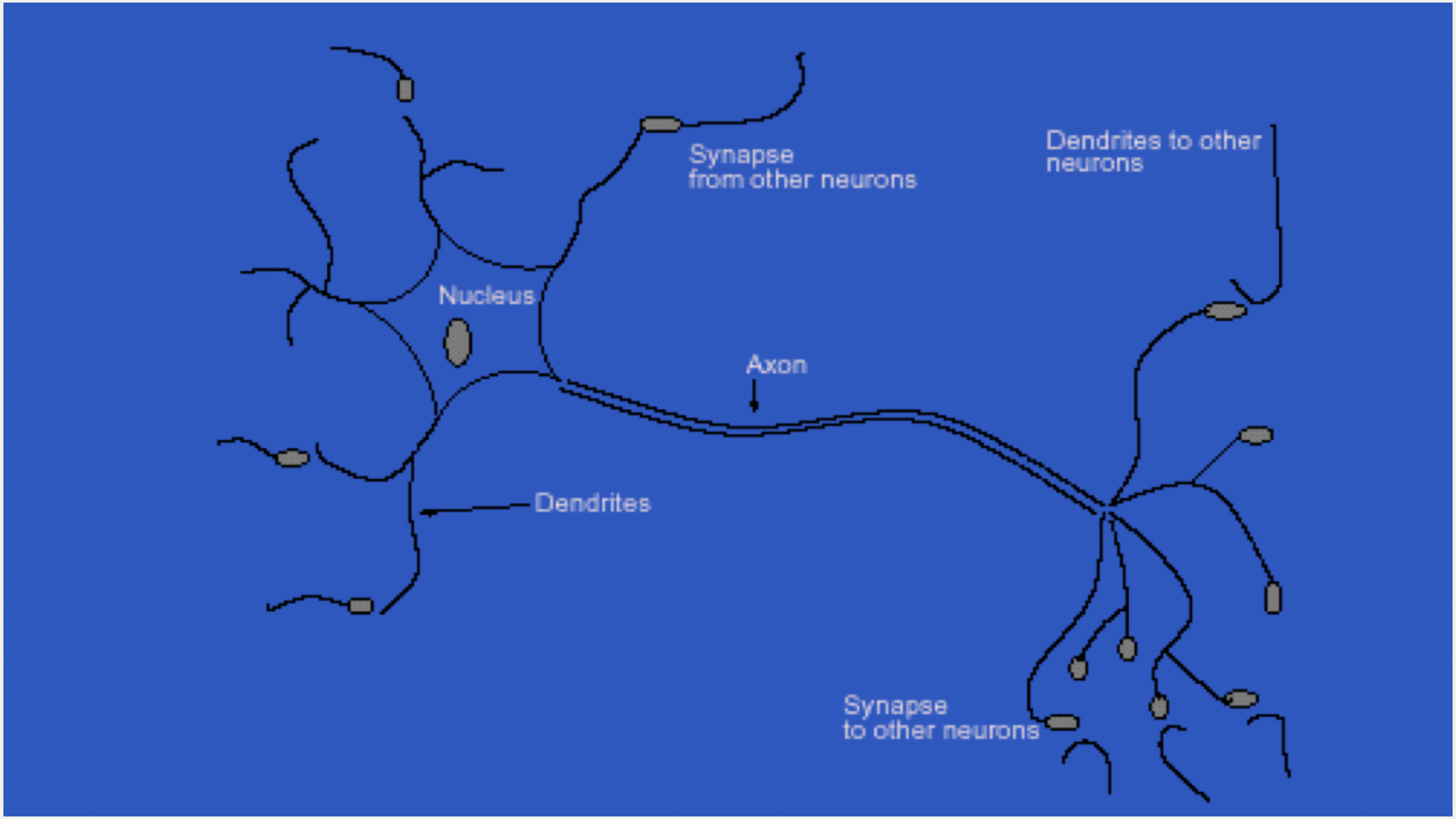
Slides modified from PPT by Mohammed Shbier

Lecture 11: Introduction to Artificial Neural Networks (ANNs)

- A Neural Network is a biologically inspired information processing idea, modeled after our brain.
- A neural network is a large number of highly interconnected processing elements (neurons) working together
- Like people, they learn from experience (by example)

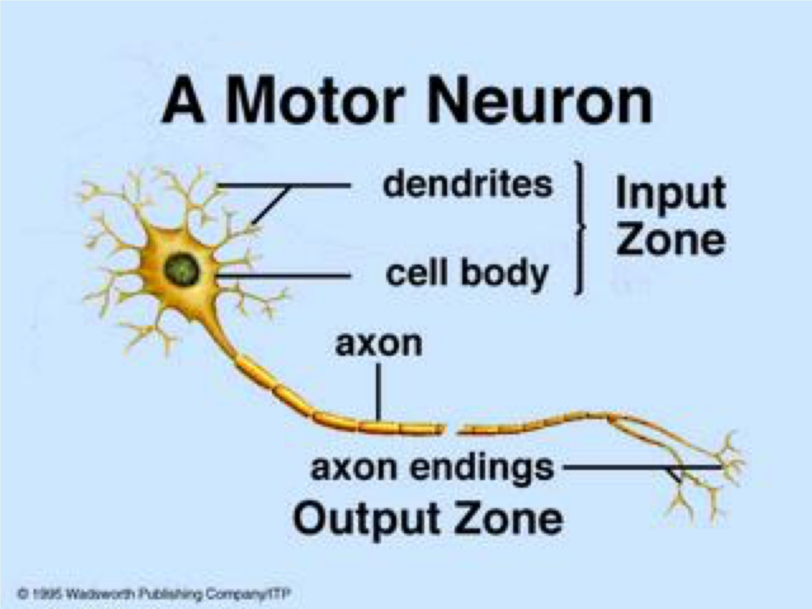
Lecture 11: Introduction to Artificial Neural Networks (ANNs)

- Neural networks take their inspiration from neurobiology
- This diagram is the human neuron:



Lecture 11: Introduction to Artificial Neural Networks (ANNs)

- A biological neuron has three types of main components; dendrites, soma (or cell body) and axon
- Dendrites receives signals from other neurons
- The soma, sums the incoming signals. When sufficient input is received, the cell fires; that is it transmit a signal over its axon to other cells.



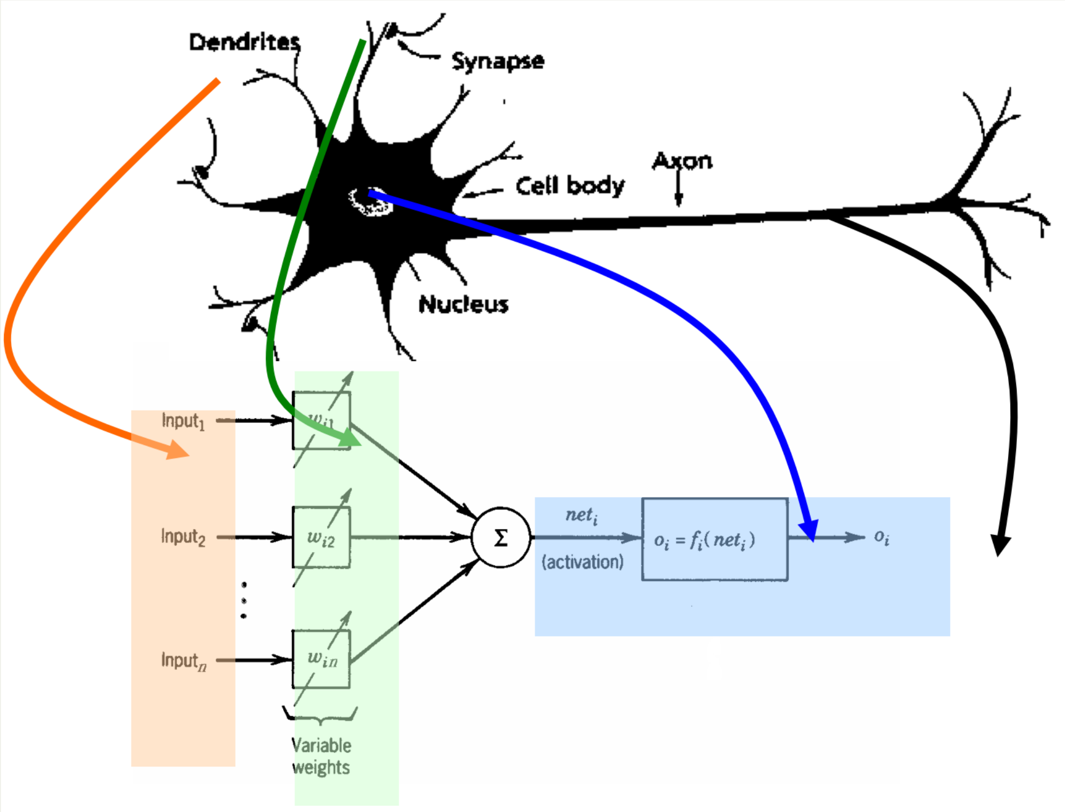
Lecture 11: Introduction to Artificial Neural Networks (ANNs)

- An artificial neural network (ANN) is an information processing system that has certain performance characteristics in common with biological nets.
- Several key features of the processing elements of ANN are suggested by the properties of biological neurons:

1. The processing element receives many signals.
2. Signals may be modified by a weight at the receiving synapse.
3. The processing element sums the weighted inputs.
4. Under appropriate circumstances (sufficient input), the neuron transmits a single output.
5. The output from a particular neuron may go to many other neurons.

Lecture 11: Introduction to Artificial Neural Networks (ANNs)

- From experience: examples / training data
- Strength of connection between the neurons is stored as a weight-value for the specific connection.
- Learning the solution to a problem = changing the connection weights

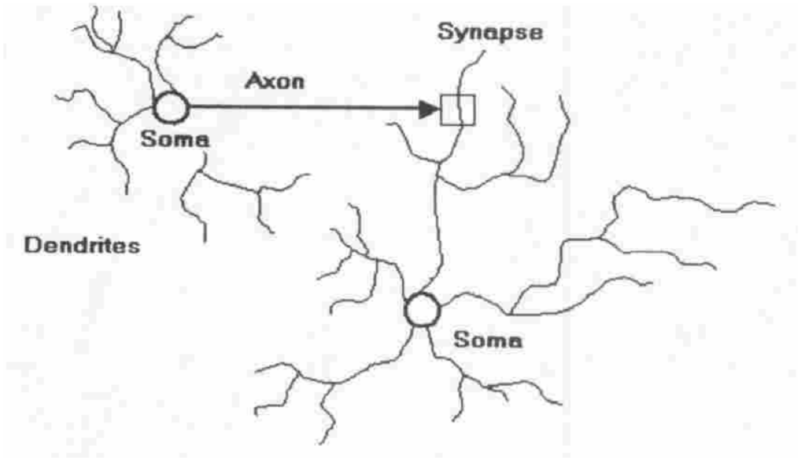


Lecture 11: Introduction to Artificial Neural Networks (ANNs)

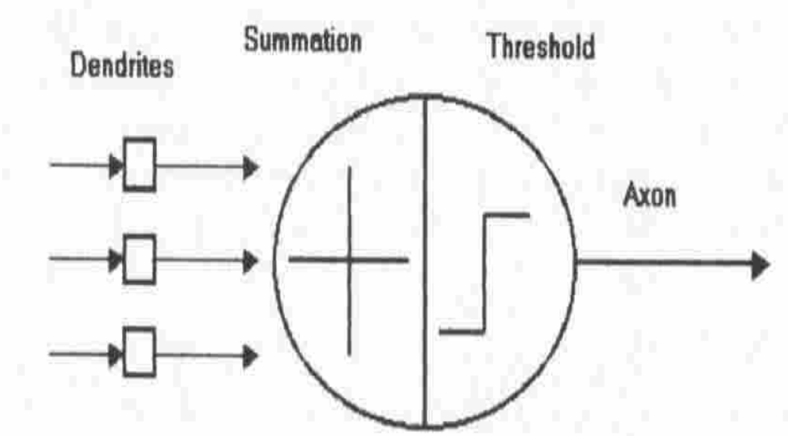
- ANNs have been developed as generalizations of mathematical models of neural biology, based on the assumptions that:

1. Information processing occurs at many simple elements called neurons.
2. Signals are passed between neurons over connection links.
3. Each connection link has an associated weight, which, in typical neural net, multiplies the signal transmitted.
4. Each neuron applies an activation function to its net input to determine its output signal.

Lecture 11: Introduction to Artificial Neural Networks (ANNs)



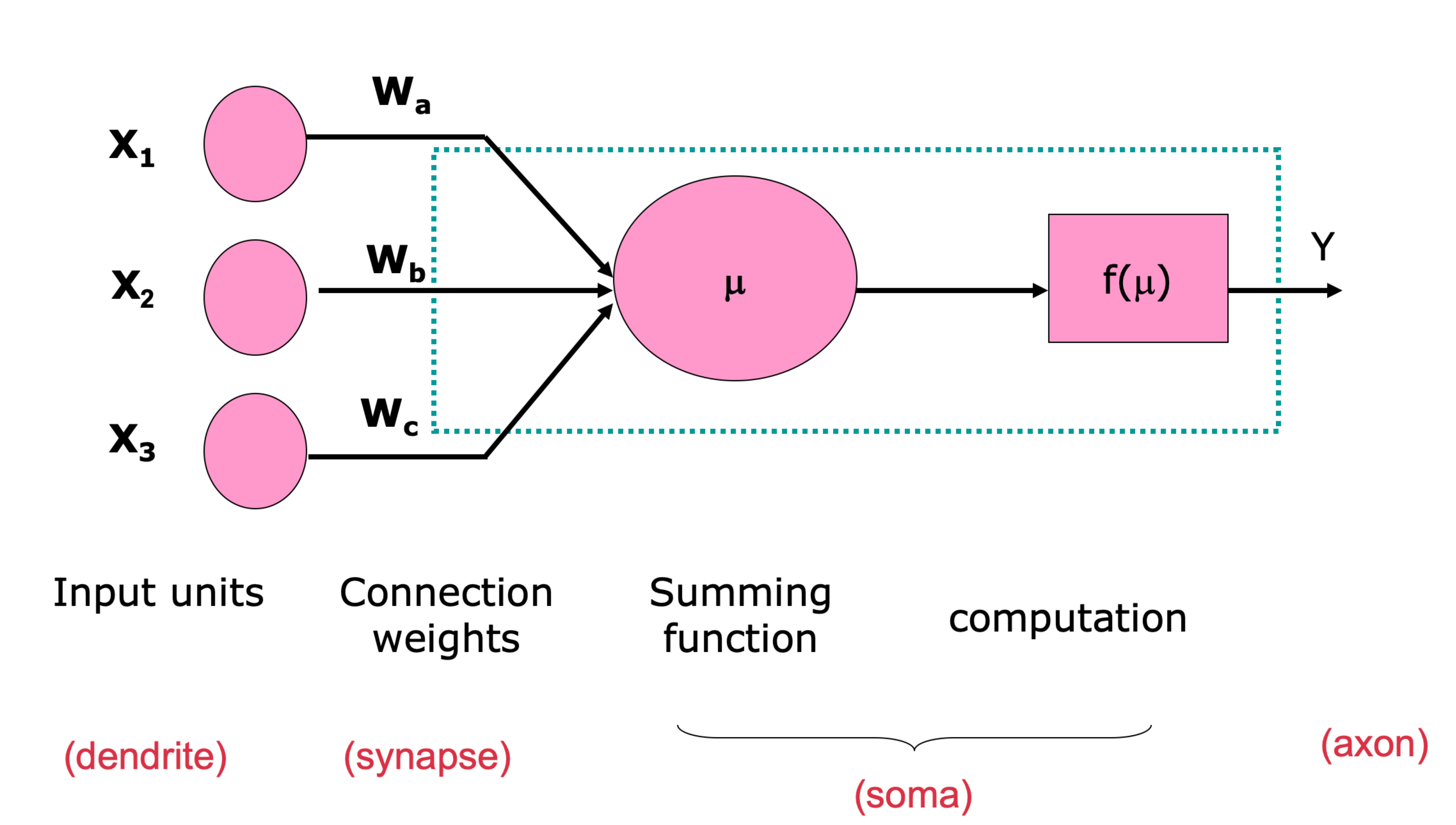
Four basic components of a human biological neuron



The components of a basic artificial neuron

Lecture 11: Introduction to Artificial Neural Networks (ANNs)

- Model of a neuron



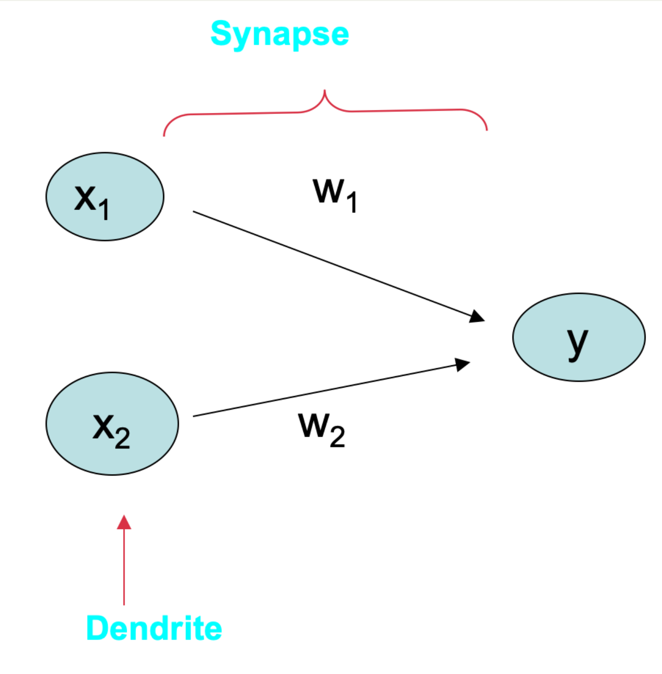
Lecture 11: Introduction to Artificial Neural Networks (ANNs)

- A neural net consists of a large number of simple processing elements called neurons, units, cells or nodes.
- Each neuron is connected to other neurons by means of directed communication links, each with associated weight.
- The weight represent information being used by the net to solve a problem.
- Each neuron has an internal state, called its activation or activity level, which is a function of the inputs it has received. Typically, a neuron sends its activation as a signal to several other neurons.
- It is important to note that a neuron can send only one signal at a time, although that signal is broadcast to several other neurons.

Lecture 11: Introduction to Artificial Neural Networks (ANNs)

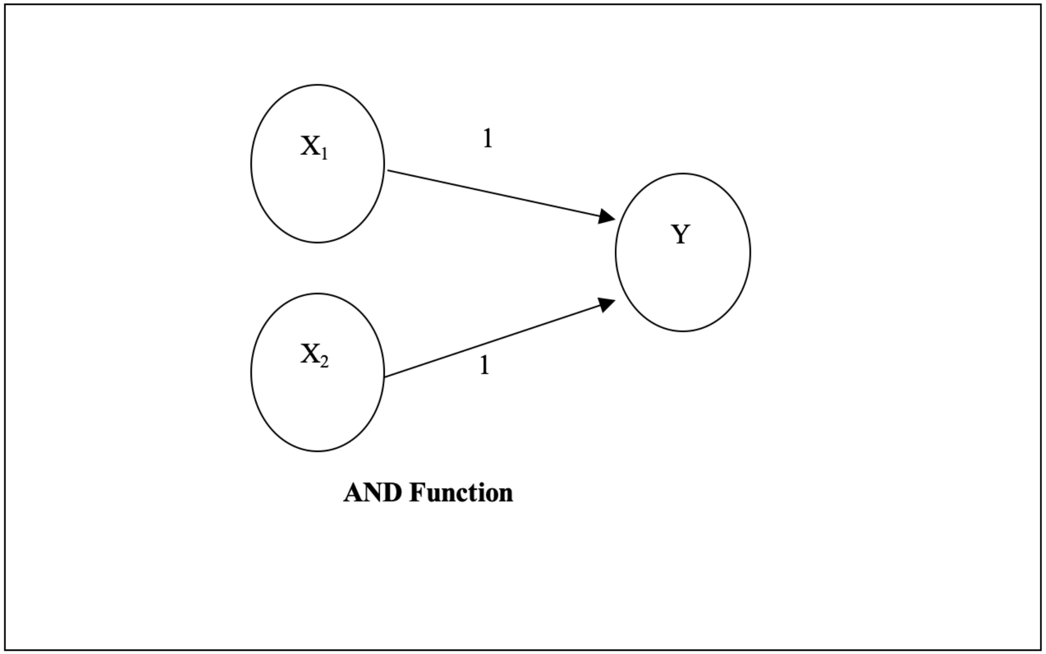
- Neural networks are configured for a specific application, such as pattern recognition or data classification, through a learning process
- In a biological system, learning involves adjustments to the synaptic connections between neurons
- This is the same for artificial neural networks (ANNs)!

Lecture 11: Introduction to Artificial Neural Networks (ANNs)



- A neuron receives input, determines the strength or the weight of the input, calculates the total weighted input, and compares the total weighted with a value (threshold)
- The value is in the range of 0 and 1
- If the total weighted input greater than or equal the threshold value, the neuron will produce the output, and if the total weighted input less than the threshold value, no output will be produced

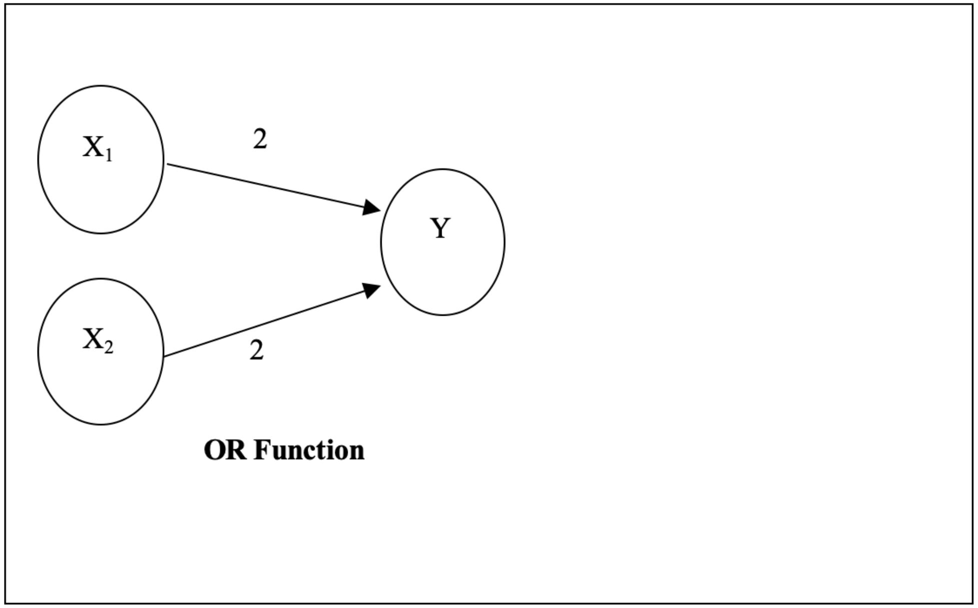
Lecture 11: Introduction to Artificial Neural Networks (ANNs)



AND		
X1	X2	Y
1	1	1
1	0	0
0	1	0
0	0	0

$$\text{Threshold}(Y) = 2$$

Lecture 11: Introduction to Artificial Neural Networks (ANNs)

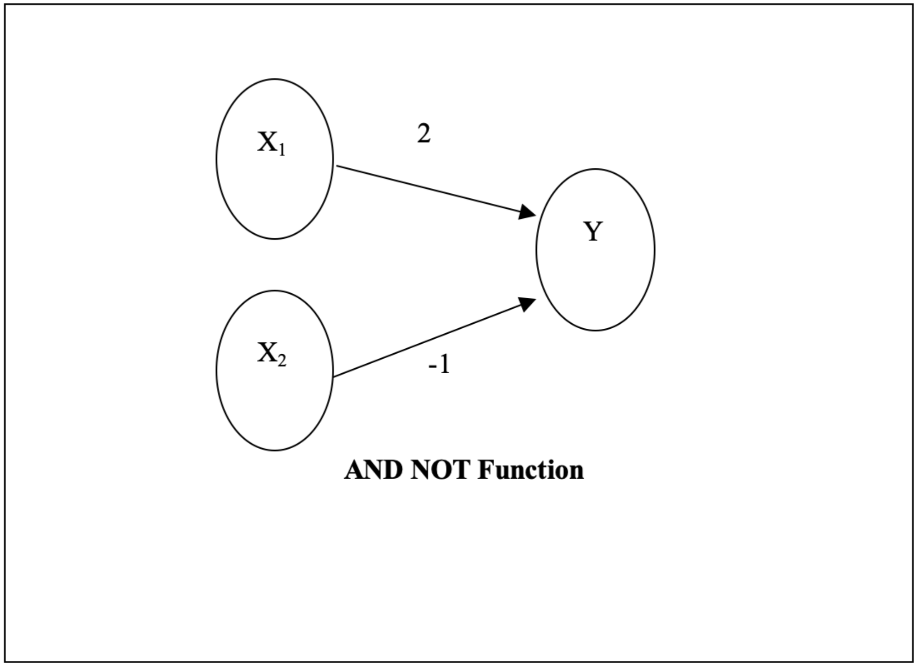


$$\text{Threshold}(Y) = 2$$

OR		
X1	**X2**	**Y**
1	1	1
1	0	1
0	1	1
0	0	0

Slides modified from Graham Kendall's Introduction to Artificial Intelligence

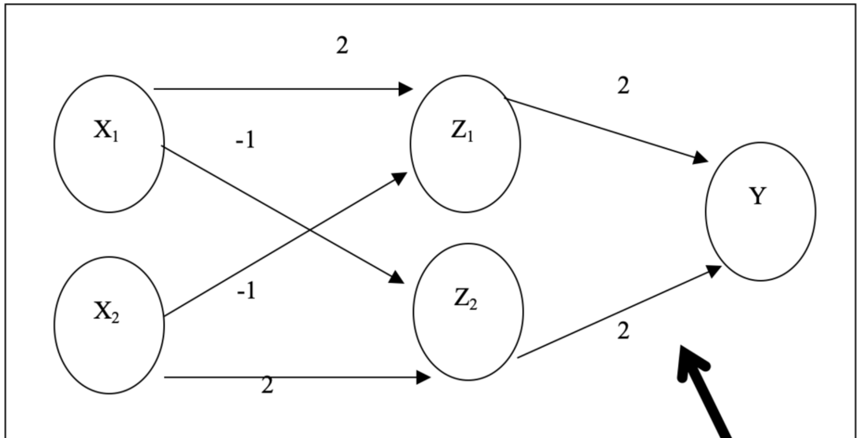
Lecture 11: Introduction to Artificial Neural Networks (ANNs)



$$\text{Threshold}(Y) = 2$$

AND NOT		
X1	**X2**	**Y**
1	1	0
1	0	1
0	1	0
0	0	0

Slides modified from Graham Kendall's Introduction to Artificial Intelligence



XOR Function

XOR		
X1	**X2**	**Y**
1	1	0
1	0	1
0	1	1
0	0	0

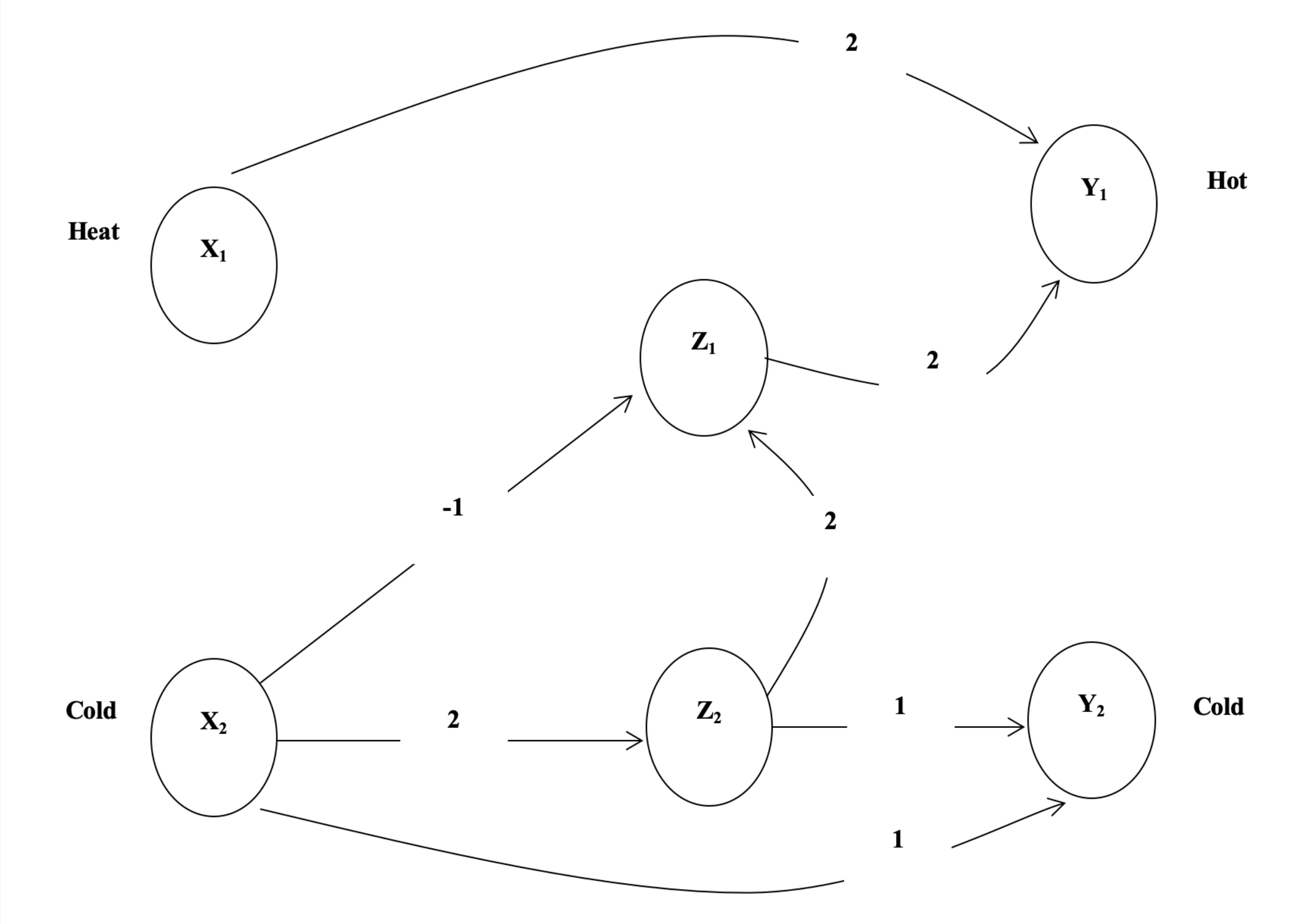
$$X_1 \text{ XOR } X_2 = (X_1 \text{ AND NOT } X_2) \text{ OR } (X_2 \text{ AND NOT } X_1)$$

Let's model a slightly more complicated neural network:

1. If we touch something **cold** we perceive heat
2. If we keep touching something **cold** we will perceive cold
3. If we touch something **hot** we will perceive heat

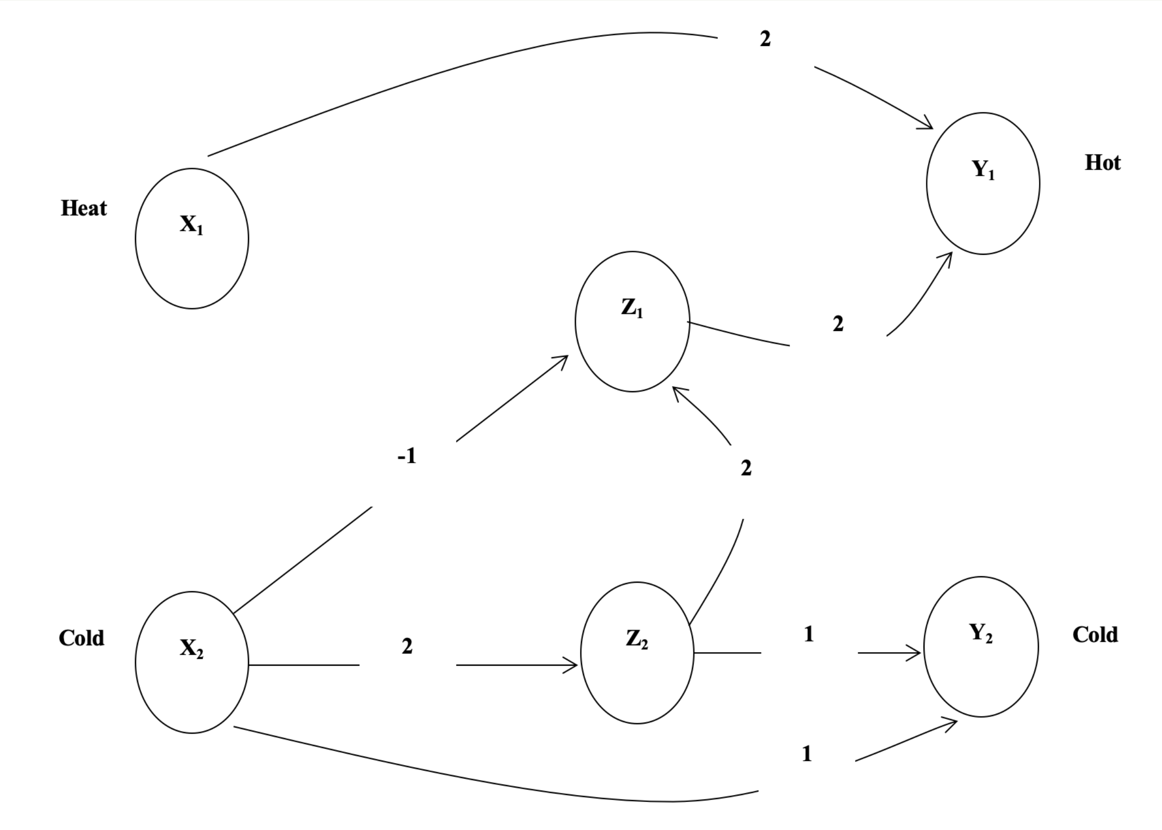
- We will assume that we can only change things on discrete time steps
- If cold is applied for **one time step** then heat will be perceived
- If a cold stimulus is applied for **two time steps** then cold will be perceived
- If heat is applied at a time step, then we should perceive heat

Lecture 11: Introduction to Artificial Neural Networks (ANNs)



Slides modified from Graham Kendall's Introduction to Artificial Intelligence

Lecture 11: Introduction to Artificial Neural Networks (ANNs)

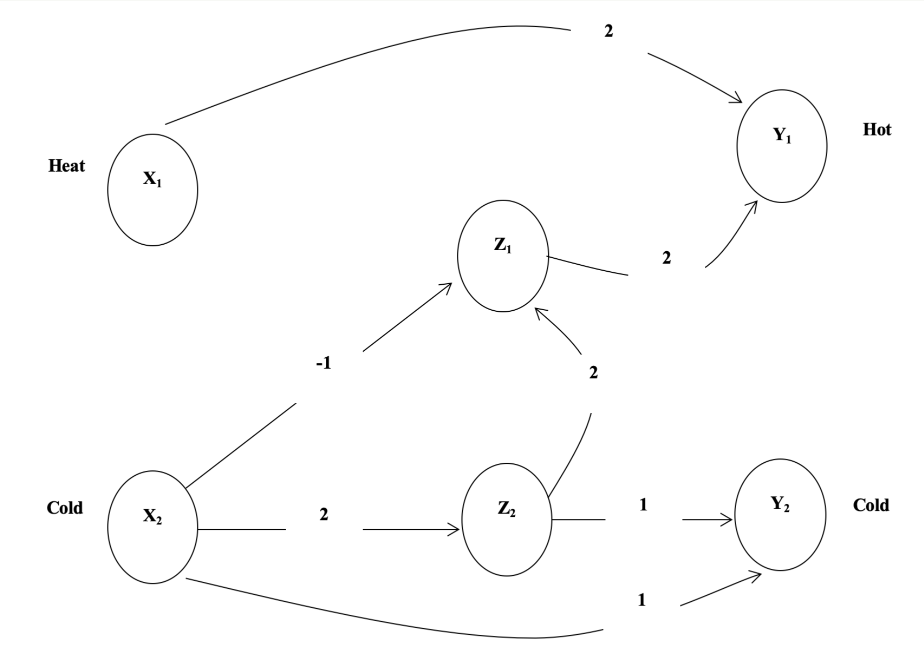


- It takes time for the stimulus (applied at X1 and X2) to make its way to Y1 and Y2 where we perceive either heat or cold

- At t(0), we apply a stimulus to X1 and X2
- At t(1) we can update Z1, Z2 and Y1
- At t(2) we can perceive a stimulus at Y2
- At t(2+n) the network is fully functional

Slides modified from Graham Kendall's Introduction to Artificial Intelligence

Lecture 11: Introduction to Artificial Neural Networks (ANNs)



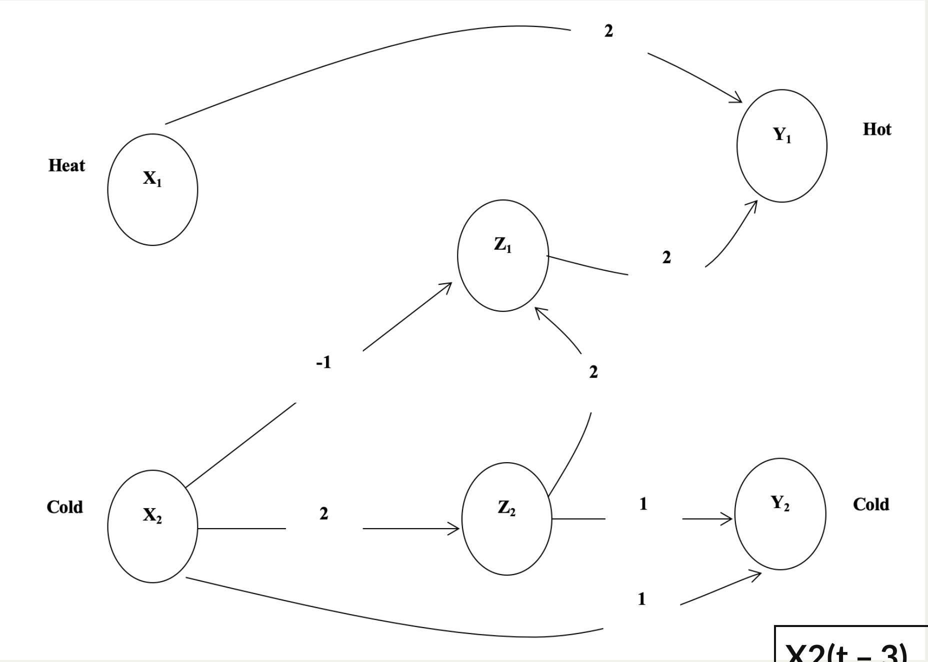
- We want the system to perceive cold if a cold stimulus is applied for two time steps

Y2(t) = X2(t – 2) AND X2(t – 1)

X2(t – 2)	X2(t – 1)	Y2(t)
1	1	1
1	0	0
0	1	0
0	0	0

Slides modified from Graham Kendall's Introduction to Artificial Intelligence

Lecture 11: Introduction to Artificial Neural Networks (ANNs)



- We want the system to perceive heat if either a hot stimulus is applied or a cold stimulus is applied (for one time step) and then removed

Y1(t) = [X1(t – 1)] OR [X2(t – 3) AND NOT X2(t – 2)]

X2(t – 3)	X2(t – 2)	AND NOT	X1(t – 1)	OR
1	1	0	1	1
1	0	1	1	1
0	1	0	1	1
0	0	0	1	1
1	1	0	0	0
1	0	1	0	1
0	1	0	0	0
0	0	0	0	0

Lecture 11: Introduction to Artificial Neural Networks (ANNs)

- The network shows

  Y1(t) = X1(t – 1) OR Z1(t – 1)

  Z1(t – 1) = Z2(t – 2) AND NOT X2(t – 2)

  Z2(t – 2) = X2(t – 3)

  Substituting, we get

  Y1(t) = [X1(t – 1)] OR [X2(t – 3) AND NOT X2(t – 2)]

  which is the same as our original requirements

Lecture 11: Using Python for Artificial Intelligence

- This is great...but how do you build a network that learns?
- We have to *use input* to *predict output*
- We can do this using a mathematical algorithm called *backpropogation*, which measures statistics from input values and output values.
- Backpropogation uses a *training set*
- We are going to use the following training set:

	Input			Output
Example 1	0	0	1	0
Example 2	1	1	1	1
Example 3	1	0	1	1
Example 4	0	1	1	0

New situation	1	0	0	?

- Can you figure out what the question mark should be?

Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

Lecture 11: Using Python for Artificial Intelligence

- This is great...but how do you build a network that learns?
- We have to *use input* to *predict output*
- We can do this using a mathematical algorithm called *backpropogation*, which measures statistics from input values and output values.
- Backpropogation uses a *training set*
- We are going to use the following training set:

	Input			Output
Example 1	0	0	1	0
Example 2	1	1	1	1
Example 3	1	0	1	1
Example 4	0	1	1	0

New situation	1	0	0	**?**

- Can you figure out what the question mark should be?
- The output is always equal to the value of the leftmost input column. Therefore the answer is the '**?**' should be 1.

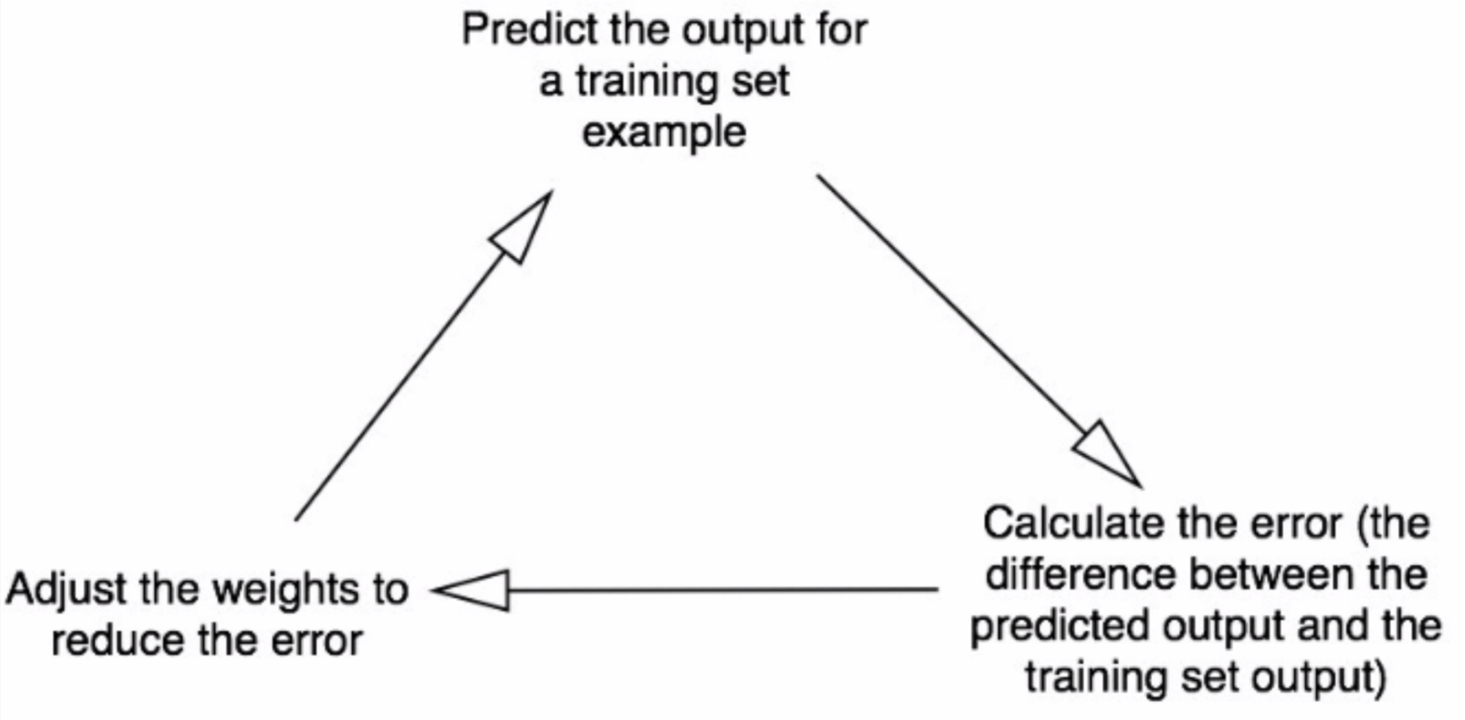
Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

Lecture 11: Using Python for Artificial Intelligence

- We start by giving each input a *weight*, which will be a positive or negative number.
- Large numbers (positive or negative) will have a large effect on the neuron's output.
- We start by setting each weight to a random number, and then we train:

1. Take the inputs from a training set example, adjust them by the weights, and pass them through a special formula to calculate the neuron's output.
2. Calculate the error, which is the difference between the neuron's output and the desired output in the training set example.
3. Depending on the direction of the error, adjust the weights slightly.
4. Repeat this process 10,000 times.

Example borrowed from: *How to build a simple neural network in 9 lines of Python code*



Eventually the weights of the neuron will reach an optimum for the training set. If we allow the neuron to think about a new situation, that follows the same pattern, it should make a good prediction.

Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

Lecture 11: Using Python for Artificial Intelligence

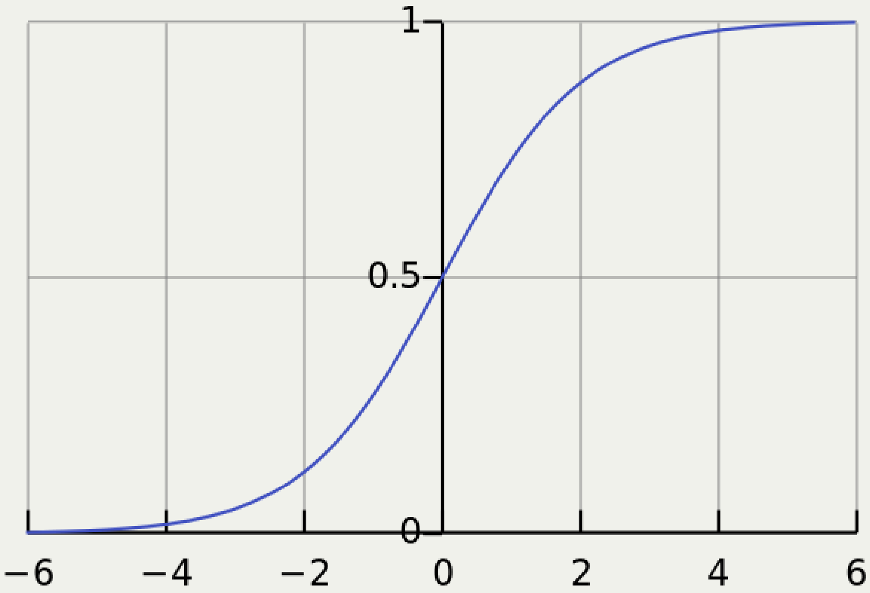
- What is this special formula that we're going to use to calculate the neuron's output?
- First, we take the weighted sum of the neuron's inputs:

$$\sum weight_i \times input_i = weight_1 \times input_1 + weight_2 \times input_2 + weight_3 \times input_3$$

- Next we *normalize* this, so the result is between 0 and 1. For this, we use a mathematically convenient function, called the *Sigmoid function*:

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

- The Sigmoid function looks like this when plotted:
- Notice the characteristic "S" shape, and that it is *bounded* by 1 and 0.



Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

Lecture 11: Using Python for Artificial Intelligence

- We can substitute the first function into the Sigmoid:

$$\frac{1}{1+e^{-\left(\sum weight_i \times input_i\right)}}$$

- During the training, we have to adjust the weights. To calculate this, we use the *Error Weighted Derivative* formula:

$$error \times input \times SigmoidCurvedGradient(output)$$

- What's going on with this formula?

1. We want to make an adjustment proportional to the size of the error
2. We multiply by the input, which is either 1 or 0
3. We multiply by the *gradient* (steepness) of the Sigmoid curve.

Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

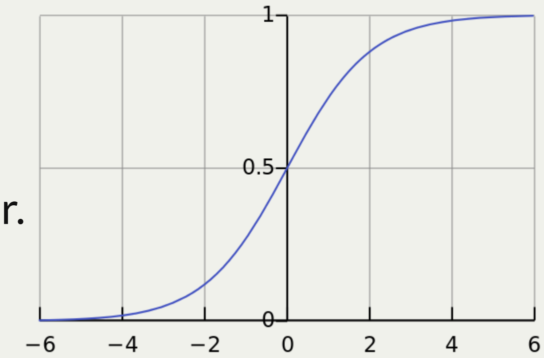
Lecture 11: Using Python for Artificial Intelligence

- What's going on with this formula?

1. We want to make an adjustment proportional to the size of the error
2. We multiply by the input, which is either 1 or 0
3. We multiply by the *gradient* (steepness) of the Sigmoid curve.

- Why the gradient of the Sigmoid?

1. We used the Sigmoid curve to calculate the output of the neuron.
2. If the output is a large positive or negative number, it signifies the neuron was quite confident one way or another.
3. From the diagram, we can see that at large numbers, the Sigmoid curve has a shallow gradient.
4. If the neuron is confident that the existing weight is correct, it doesn't want to adjust it very much. Multiplying by the Sigmoid curve gradient achieves this.



Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

Lecture 11: Using Python for Artificial Intelligence

- The gradient of the Sigmoid curve, can be found by taking the derivative (remember calculus?)

$$SigmoidCurvedGradient(output) = output \times (1 - output)$$

- So by substituting the second equation into the first equation (from two slides ago), the final formula for adjusting the weights is:

$$error \times input \times output \times (1 - output)$$

- There are other, more advanced formulas, but this one is pretty simple.

Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

Lecture 11: Using Python for Artificial Intelligence

- Finally, Python!
- We will use the *numpy* module, which is a mathematics library for Python.
- We want to use four methods:

1. exp — the natural exponential
2. array — creates a matrix
3. dot — multiplies matrices
4. random — gives us random numbers

array() creates list-like arrays that are faster than regular lists. E.g., for the training set we saw earlier:

```
training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])
training_set_outputs = array([[0, 1, 1, 0]]).T
```

- The '.T' function, transposes the matrix from horizontal to vertical. So the computer is storing the numbers like this:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

Lecture 11: Using Python for Artificial Intelligence

In 10 lines of Python code:

```
from numpy import exp, array, random, dot
training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])
training_set_outputs = array([[0, 1, 1, 0]]).T
random.seed(1)
synaptic_weights = 2 * random.random((3, 1)) - 1
for iteration in range(10000):
    output = 1 / (1 + exp(-(dot(training_set_inputs, synaptic_weights))))
    synaptic_weights += dot(training_set_inputs.T, (training_set_outputs - output)
                                                    * output * (1 - output))
print 1 / (1 + exp(-(dot(array([1, 0, 0]), synaptic_weights))))
```

Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

Lecture 11: Using Python for Artificial Intelligence

With comments, and in a Class:

Too small! Let's do this in PyCharm

https://github.com/miloharper/simple-neural-network

```
from numpy import exp, array, random, dot


class NeuralNetwork():
    def __init__(self):
        # Seed the random number generator, so it generates the same numbers
        # every time the program runs.
        random.seed(1)

        # We model a single neuron, with 3 input connections and 1 output connection.
        # We assign random weights to a 3 x 1 matrix, with values in the range -1 to 1
        # and mean 0.
        self.synaptic_weights = 2 * random.random((3, 1)) - 1

    # The Sigmoid function, which describes an S shaped curve.
    # We pass the weighted sum of the inputs through this function to
    # normalise them between 0 and 1.
    def __sigmoid(self, x):
        return 1 / (1 + exp(-x))

    # The derivative of the Sigmoid function.
    # This is the gradient of the Sigmoid curve.
    # It indicates how confident we are about the existing weight.
    def __sigmoid_derivative(self, x):
        return x * (1 - x)

    # We train the neural network through a process of trial and error.
    # Adjusting the synaptic weights each time.
    def train(self, training_set_inputs, training_set_outputs, number_of_training_iterations):
        for iteration in range(number_of_training_iterations):
            # Pass the training set through our neural network (a single neuron).
            output = self.think(training_set_inputs)

            # Calculate the error (The difference between the desired output
            # and the predicted output).
            error = training_set_outputs - output

            # Multiply the error by the input and again by the gradient of the Sigmoid curve.
            # This means less confident weights are adjusted more.
            # This means inputs, which are zero, do not cause changes to the weights.
            adjustment = dot(training_set_inputs.T, error * self.__sigmoid_derivative(output))

            # Adjust the weights.
            self.synaptic_weights += adjustment

    # The neural network thinks.
    def think(self, inputs):
        # Pass inputs through our neural network (our single neuron).
        return self.__sigmoid(dot(inputs, self.synaptic_weights))


if __name__ == "__main__":

    #Intialise a single neuron neural network.
    neural_network = NeuralNetwork()

    print("Random starting synaptic weights: ")
    print(neural_network.synaptic_weights)

    # The training set. We have 4 examples, each consisting of 3 input values
    # and 1 output value.
    training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])
    training_set_outputs = array([[0, 1, 1, 0]]).T

    # Train the neural network using a training set.
    # Do it 10,000 times and make small adjustments each time.
    neural_network.train(training_set_inputs, training_set_outputs, 10000)

    print("New synaptic weights after training: ")
    print(neural_network.synaptic_weights)

    # Test the neural network with a new situation.
    print("Considering new situation [1, 0, 0] -> ?: ")
    print(neural_network.think(array([1, 0, 0])))
```

Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

Lecture 11: Using Python for Artificial Intelligence

When we run the code, we get something like this:

```
Random starting synaptic weights:
[[-0.16595599]
[ 0.44064899]
[-0.99977125]]

New synaptic weights after training:
[[ 9.67299303]
[-0.2078435 ]
[-4.62963669]]

Considering new situation [1, 0, 0] -> ?:
[ 0.99993704]
```

- First the neural network assigned itself random weights, then trained itself using the training set. Then it considered a new situation [1, 0, 0] and predicted 0.99993704. The correct answer was 1. So very close!
- This was one neuron doing one task, but if we had millions of these working together, we could create a much more robust network!

Example borrowed from: *How to build a simple neural network in 9 lines of Python code*

Lecture 11: Example: PyTorch

- The example we just finished is pretty tiny, and involves only one neuron.
- If we want to do more powerful neural networks, we should use a library. One of the most widely used machine learning library is called PyTorch, and it is open source and available for many platforms.
- PyTorch allows you to use *Graphics Processing Units (GPUs)* for doing the substantial processing necessary for large machine learning problems
-