Artificial Neural Networks (Part I)

CS576 MACHINE LEARNING

Dr. Jin S. Yoo, Professor
Department of Computer Science
Purdue University Fort Wayne

Reference

- Kelleher et al., Fundamentals of Machine Learning (2nd edition), Ch 8

Outline

- Introduction
- Fundamentals: Artificial Neural Networks
- Training Neural Networks with a Single Layer
- Standard Approach: Backpropagation Algorithm
- A Worked Example

Introduction

- ***Artificial Neural Networks*** are machine learning models inspired by the networks of biological neurons found in our brains.
- The primary concept behind neural networks is to create computational models mirroring the structure and functions of the human brain.
  - Although ANNs have gradually become quite different from their biological cousins, e.g., by saying “units” rather than “neurons”

Biological Learning Mechanism

- Biological learning mechanisms consist of complicated *networks of interconnected* ***neurons***, where each neuron is essentially a basic signal processor.
  - It's estimated that the human brain comprises a densely interconnected network of around 100 billion neurons, with each neuron on average connected to 10,000 others. (Herculano-Houzel, 2009)
- **Biological Neuron**
  - A **neuron** is composed of a ***cell body*** with many branching extensions called ***dendrites***, plus one very long extension called the ***axon***.
  - The axon splits off into many branches called ***telodendria***, and at the tip of these branches are minuscule structures called ***synaptic terminals*** (or simply ***synapses***), which are connected to the dendrites or cell bodies of other neurons.

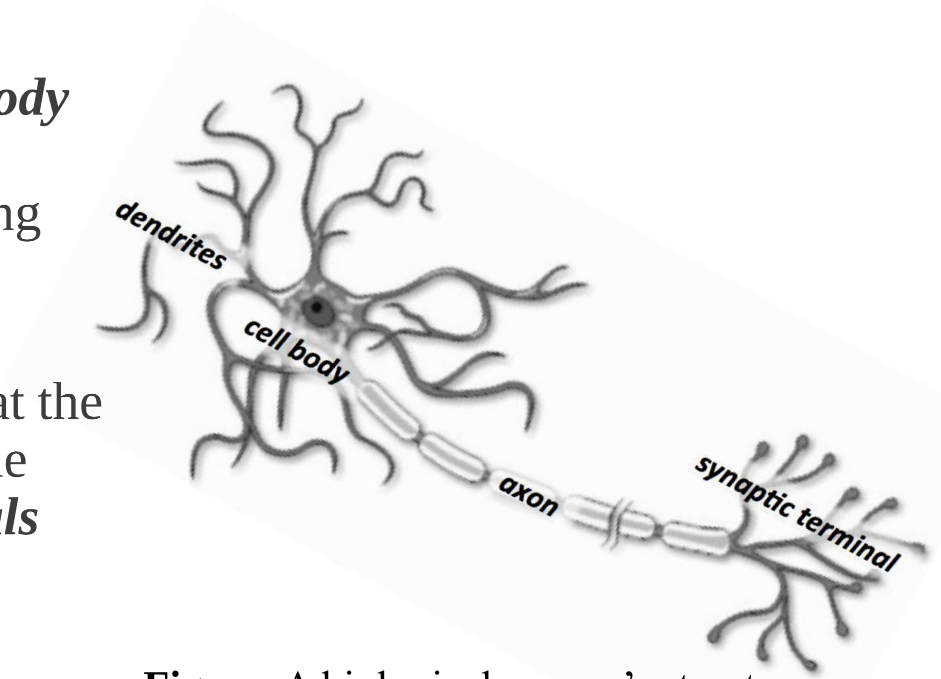


Figure. A biological neuron's structure

Biological Learning Mechanism

- **Donald O. Hebb's Theory** (1949)
  - Complex behaviors arise from interactions among vast numbers of tightly interconnected neurons, not from intricate processes within individual neurons.
  - Learning happens as connections between neurons evolve, and behavior arises from the transmission of information through these connections

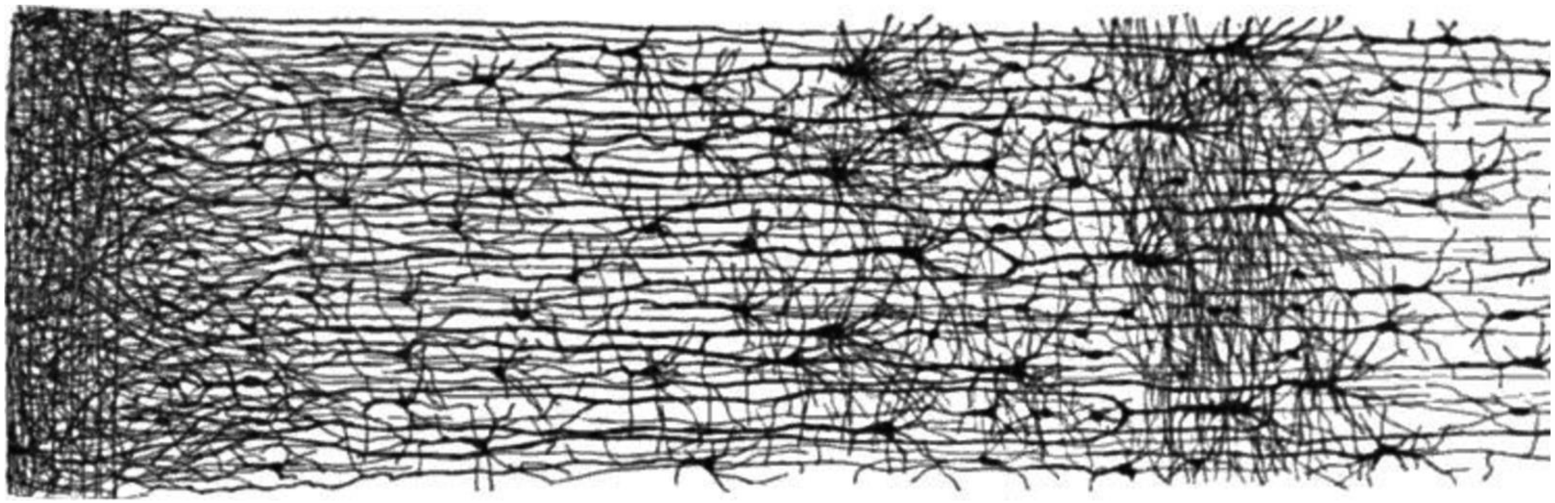


Figure. Multiple layers in a biological neural network *(human cortex)*

Artificial Neural Network

- ***Artificial Neural Networks*** (***ANNs***) consist of numerous simple processing **units**, termed as '**neurons**' in a biological context.
- These units are typically **organized into layers and are densely interconnected.**
- ANNs form the foundation of **Deep Learning**.
  - Deep learning describes the modern evolution of artificial neural networks.
  - "Deep" in deep learning refers to the increased number of layers in these networks, allowing them to map more intricate input-output relationships.

Applications

- ANNs excel in areas with numerous input features, such as image, speech, or language processing, especially when vast training datasets are available.
  - Image classification (e.g., Google Images),
  - Power speech recognition (e.g., Apple's Siri),
  - Recommending top videos (e.g., YouTube),
  - Mastering games (e.g., Go as demonstrated by DeepMind's AlphaGo).
- ANNs are some of the most powerful machine learning models, able to learning complex non-linear mappings from inputs to outputs.

Outline

- Introduction
- ☞ **Fundamentals: Standard ANN Architecture**
  - Artificial Neurons
  - Artificial Neural Networks
  - Limitation of a Single-layer Network
  - Network Depth and Representational Capability
- Training Neural Networks with a Single Layer
- Standard Approach: Backpropagation Algorithm
- A Worked Example

From Biological to Artificial Neurons

- Artificial Neural Networks were first introduced by the neurophysiologist **Warren MacCulloch** and the mathematician **Walter Pitts (1943).**
- They introduced **a basic model of a biological neuron**, (subsequently referred to as an ***artificial neuron***)
- The **McCulloch and Pitts neuron** possesses **one or more binary inputs** (on/off) and **a single binary output** (acting as an all-or-none switch).

Artificial Neuron by McCulloch and Pitts

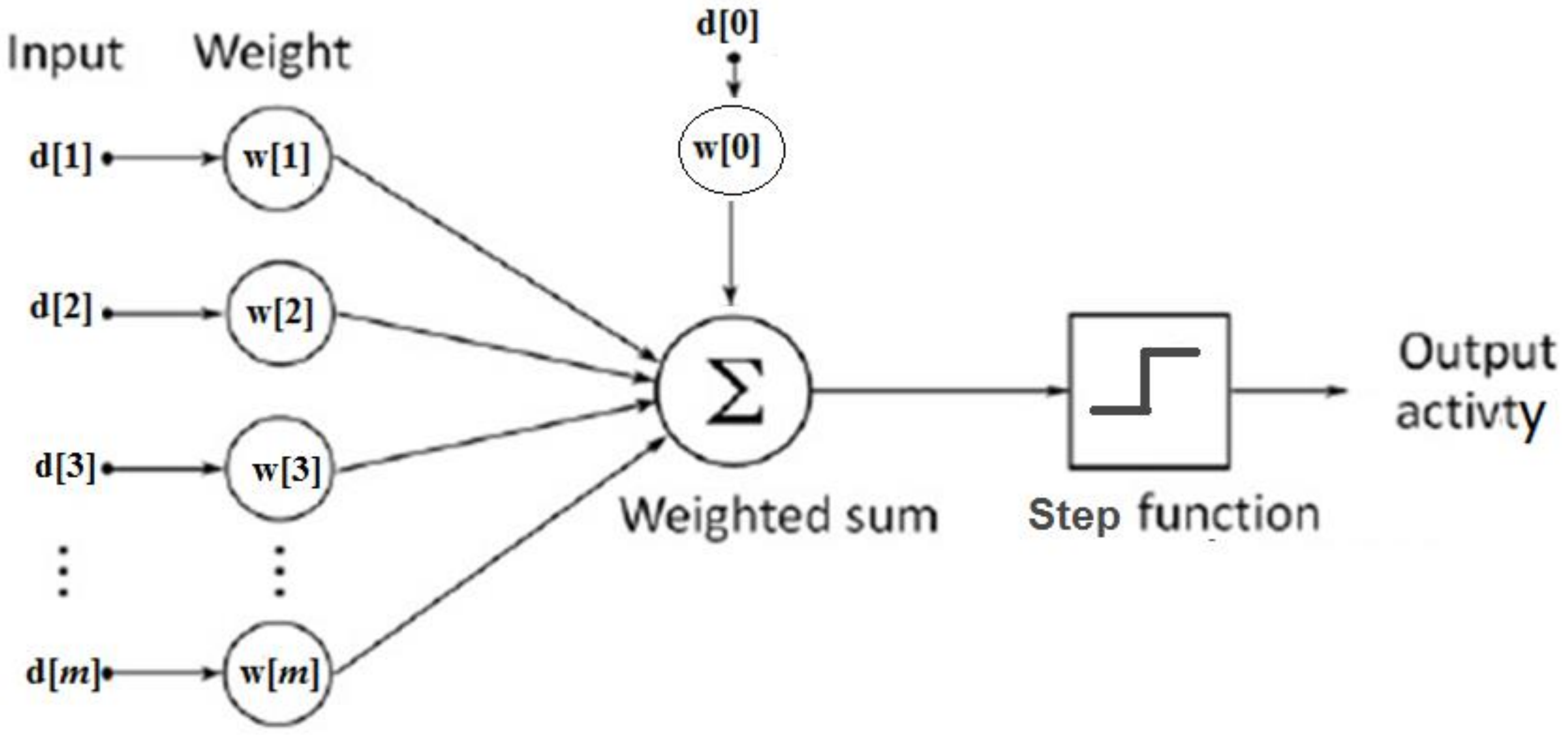


Figure: The template structure of computational neuron model by McCulloch and Pitts (1943)

McCulloch-Pitts Neuron

- The McCulloch-Pitts neuron model has a two-part structure.
- Stage 1
  - Each binary input $< \mathrm{d}[0], \ldots, \mathrm{d}[m] >$ is multiplied by a ***weight*** and the results of these multiplications are then added together. (known as a ***weighted sum***)

$$\boldsymbol{z} = \underbrace{\mathrm{w}[0] \times \mathrm{d}[0] + \mathrm{w}[1] \times \mathrm{d}[1] + \cdots + \mathrm{w}[m] \times \mathrm{d}[m]}_{\textit{weighted sum}} = \sum_{i=0}^{m} \mathrm{w}[i] \times \mathrm{d}[i]$$

$$= \underbrace{\mathbf{w} \cdot \mathbf{d}}_{\textit{dot product}}$$

where **d** is a vector of m+1 features, and **w** is a vector of m+1 weights.

$$= \underbrace{\mathbf{w}^T \mathbf{d}}_{\textit{matrix product}} = [w_0, w_1, \ldots, w_m] \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_m \end{bmatrix}$$

- **w**[0] is the equivalent of the **y-intercept** in the equation of the line and also referred to the **bias** parameter.
  - The bias inclusion changes the function from a linear function on the inputs to an *affine function* which does not necessarily pass through the origin of the input space
- **d**[0] is a dummy descriptive feature used for notational convenience and is always to 1.

McCulloch-Pitts Neuron(Cont.)

- **Stage 2**
  - The result of the weighted sum calculation ***z* is then converted into a high or a low activation** by comparing the value of z with a manually present threshold θ.
  - If z is greater than or equal to the threshold, the artificial neuron outputs a **1** (high activation), and otherwise it outputs a **0** (low activation).

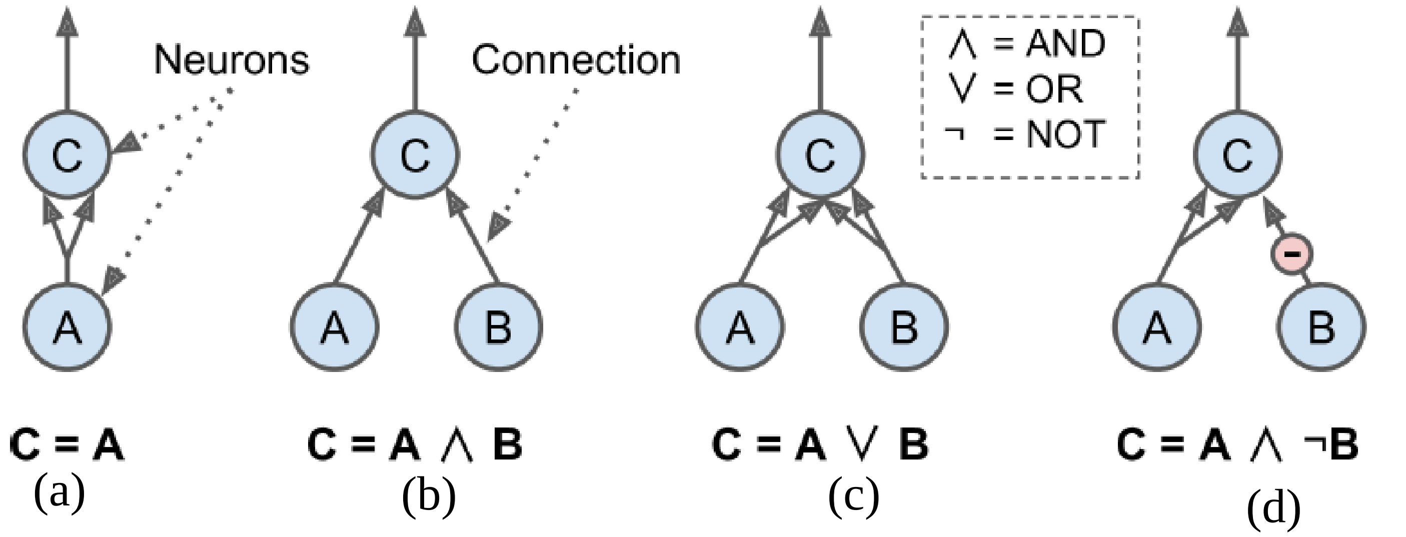
- **McCulloch and Pitts neuron model**

$$\mathbb{M}_{w}(\mathbf{d}) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

, where $\mathbb{M}_{w}(\mathbf{d})$ denotes the output of the neuron.
  - The McCulloch-Pitts neuron is also called a ***threshold unit, threshold logic unit (TLU),*** or ***linear threshold unit (LTU) .***
  - Weights and thresholds are fixed, not learned from data.

Logical Computations with McCulloch-Pitts Neurons

- ANNs performing simple logical computations



(a) The identify functions: If neuron A is activated, then neuron C gets activated as well.
(b) A logical AND: neuron C is activated only when both neurons A and B are activated
(c) A logical OR: neuron C gets activated if either neuron A or neuron B are activated.
(d) neuron C is activated only if neuron A is active and if neuron B is off.

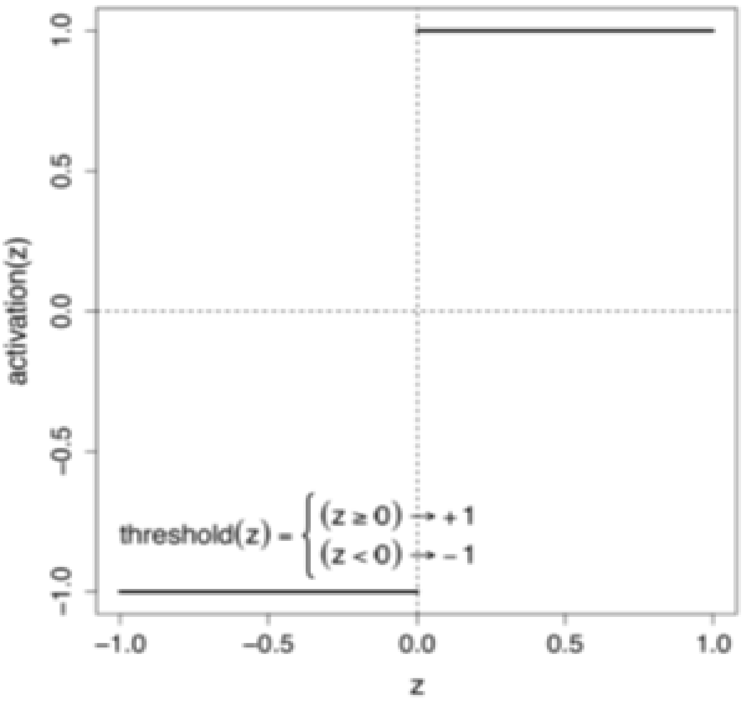
Activation Function

- **Activation function**, used as a general term, refers to the function employed in the second stage of an artificial neuron to map the weighted sum value, *z*, into the neuron's output or activation.
- Popular activation functions are :
- **Thresholding function:**

$$\boldsymbol{step(z)} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

$$unit\ step(\mathrm{z}) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$$

$$sign\ step(\mathrm{z}) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z = 0 \\ -1 & \text{if } z < 0 \end{cases}$$



Activation Function (cont.)

- **Logistic function**

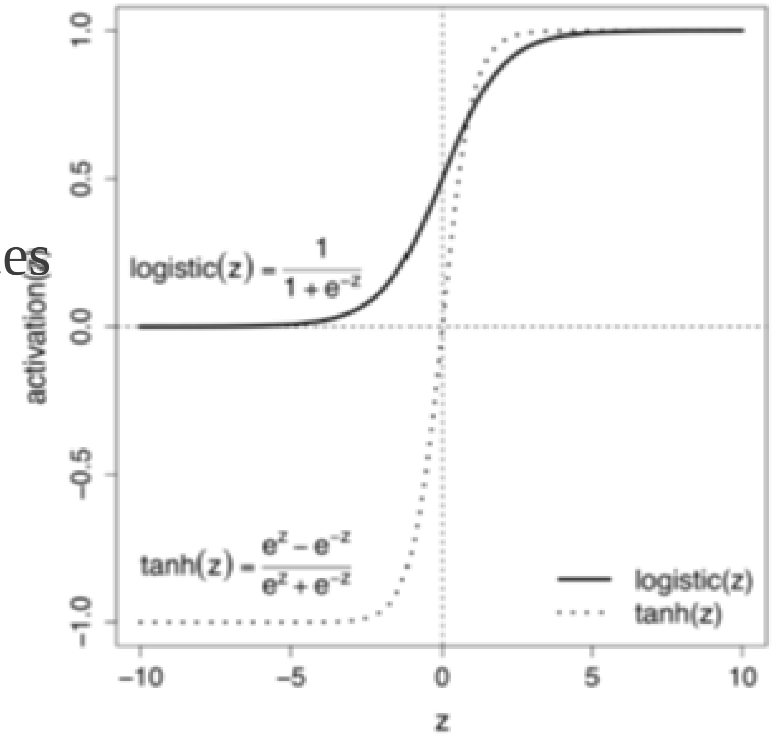
$$logistic(z) = \frac{1}{1 + e^{-z}}$$

  - It's one of the most widely used activation functions.
  - It has an S-shaped curve, is continuous, differentiae, and produces output values ranging from 0 to 1.

- **Hyperbolic tangent (sigmoid) function:**

$$tanh(z) = 2logistic(2z) - 1$$
$$= \frac{e^{2z} - 1}{e^{2z} + 1}$$

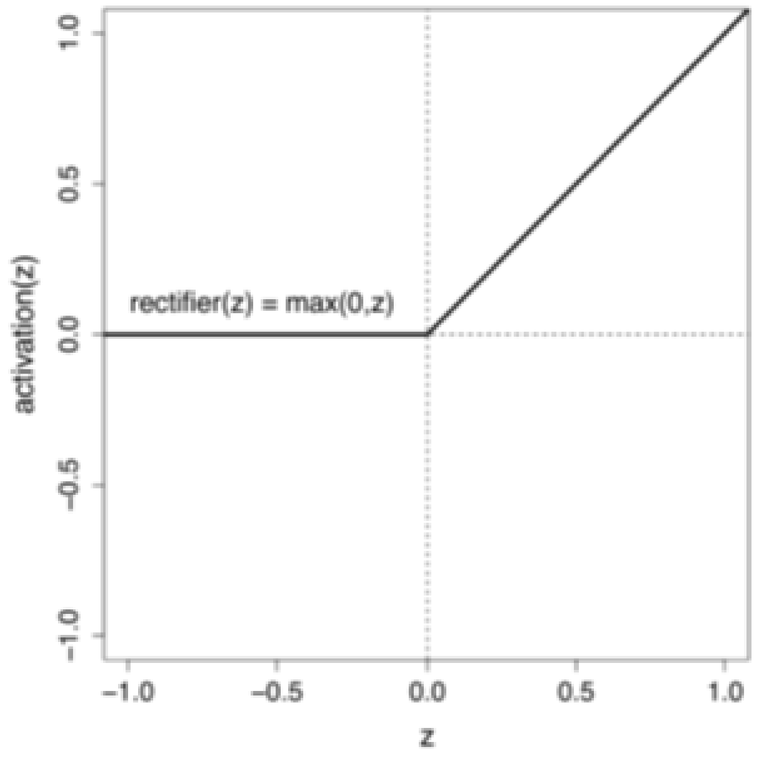
  - It has an S-shaped curve, is continuous, differentiae, and produces output values ranging from -1 to 1.



- **Rectifier function:**

  ***rectifier*(*z*)** = *max*(0, *z*)

  - It is quick to compute.
  - It is continuous, but not differentiable at z=0. Its derivative is 0 for z < 0
  - A neuron employing the rectifier is termed as a **Rectified Linear Unit (ReLU).**
  - The ReLU has recently become the neuron of choice for many tasks.



General Artificial Neuron Model

- **Artificial neuron model with activation function φ**

$$\mathbb{M}_{w}(\mathbf{d}) = \varphi(\mathbf{w}[0] \times \mathbf{d}[0] + \mathbf{w}[1] \times \mathbf{d}[1] + \cdots + \mathbf{w}[m] \times \mathbf{d}[m]) = \varphi\left(\sum_{i=0}^{m} w_i \times d_i\right)$$

$$= \varphi\left([w_0, w_1, \dots, w_m] \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_m \end{bmatrix}\right) = \varphi(\mathbf{w}^T\mathbf{d}) = \varphi(\mathbf{w} \cdot \mathbf{d})$$

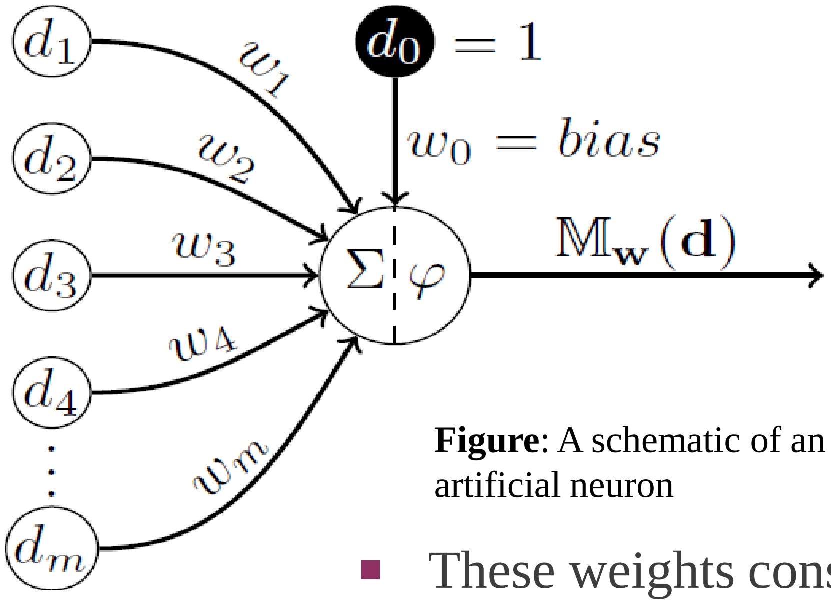


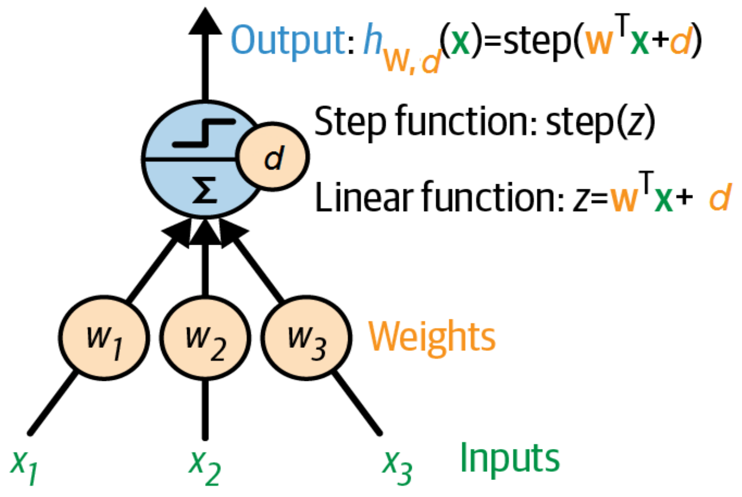
Figure: A schematic of an artificial neuron

- **Arrows** indicate the direction of activation flow.
- Each arrow's **weight label** represents the weight applied to the corresponding feature.
- The ' Σ ' **symbol** denotes the weighted sum of the inputs,
- The '**φ**' **symbol** symbolizes the activation function, which transforms the weighted sum into an activation.

- These weights constitute our parameter vector, **w**, and our ability to solve problems with neural networks depends on finding the optimal values to plug into **w**.

Rosenblatt's Perceptron (1957)

- The ***perceptron*** is one of the simplest ANN architecture, introduced by Frank Rosenblatt in 1957.
- It is inspired by the McCulloch-Pitts neuron.
- It uses a **threshold activation** function, similar to McCulloch-Pitts, but **also** introduced **other types of activation functions**.
- The model takes multiple inputs and produces a single output.
  - Unlike binary on/off values, the perceptron processes inputs and outputs as **numbers**. Each input connection is paired with a corresponding weight.
- A single threshold logic unit (TLU) can be used for simple linear binary classification as well as regression.
  - It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it outputs the negative class.



Output: $h_{\mathbf{w},d}(\mathbf{x})=\text{step}(\mathbf{w}^T\mathbf{x}+d)$
Step function: step(z)
Linear function: $z=\mathbf{w}^T\mathbf{x}+d$
d
Σ
w_1 w_2 w_3
Weights
x_1 x_2 x_3
Inputs

Express Linear Perceptron as Neurons

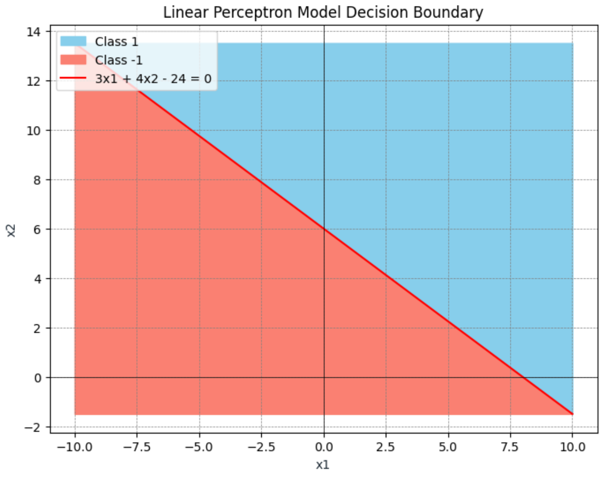
- **Example**: A machine learning model to capture the relationship between success on exams and time spent studying and sleeping.
  - A **linear perceptron classifier** that divided the Cartesian coordinate plane into two halves.

$$\mathbb{M}_{\boldsymbol{w}}(\mathbf{x}) = \begin{cases} -1 & \text{if } 3x_1 + 4x_2 - 24 < 0 \\ 1 & \text{if } 3x_1 + 4x_2 - 24 \geq 0 \end{cases}$$

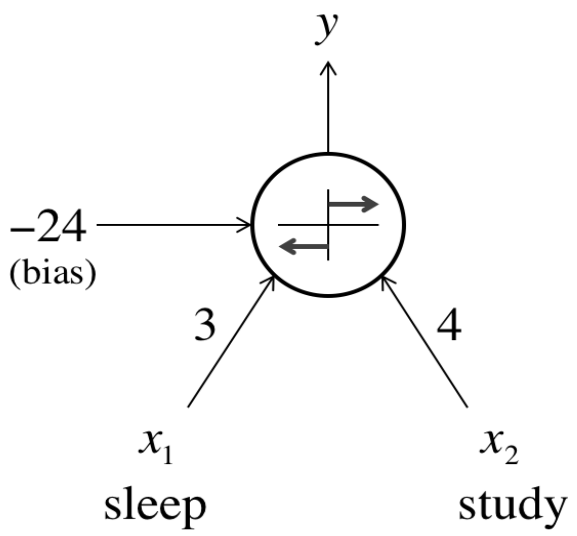
  - The equivalent **neuronal model** is:

$$\mathbb{M}_{\boldsymbol{w}}(\mathbf{x}) = \begin{cases} -1 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

- Every linear perceptron can be expressed as a single neuron, but single neuron can also express models that cannot be expressed by any linear perceptron.



(a) A linear perceptron classifier



(b) The equivalent neuron model

Perceptron with Multiple Neurons

- A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input.

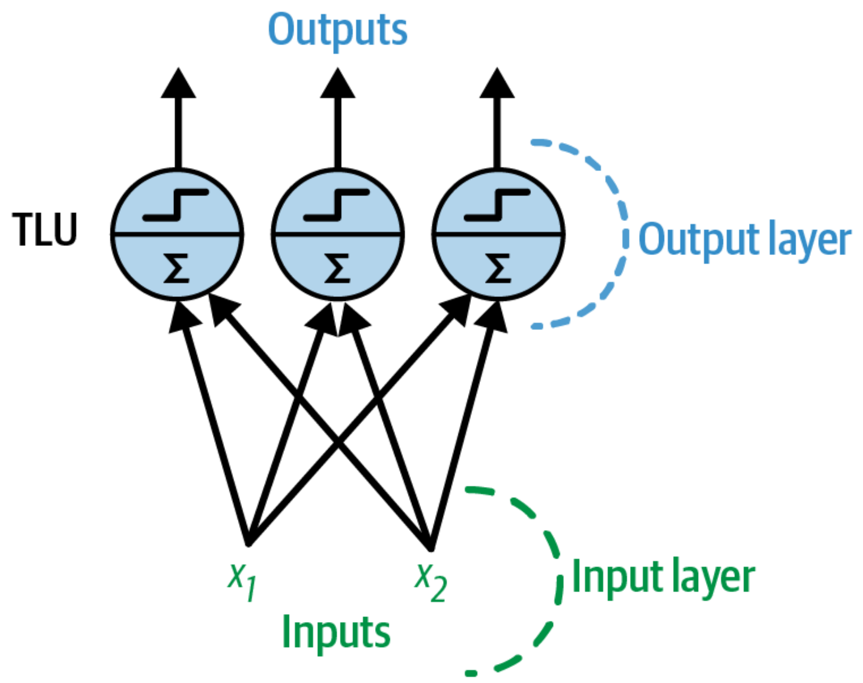


Figure: Architecture of a perceptron with two inputs and three output neurons

Outline

Artificial Neural Networks

- An **artificial neural network** is composed of interconnected artificial neurons organized into sequential layers:

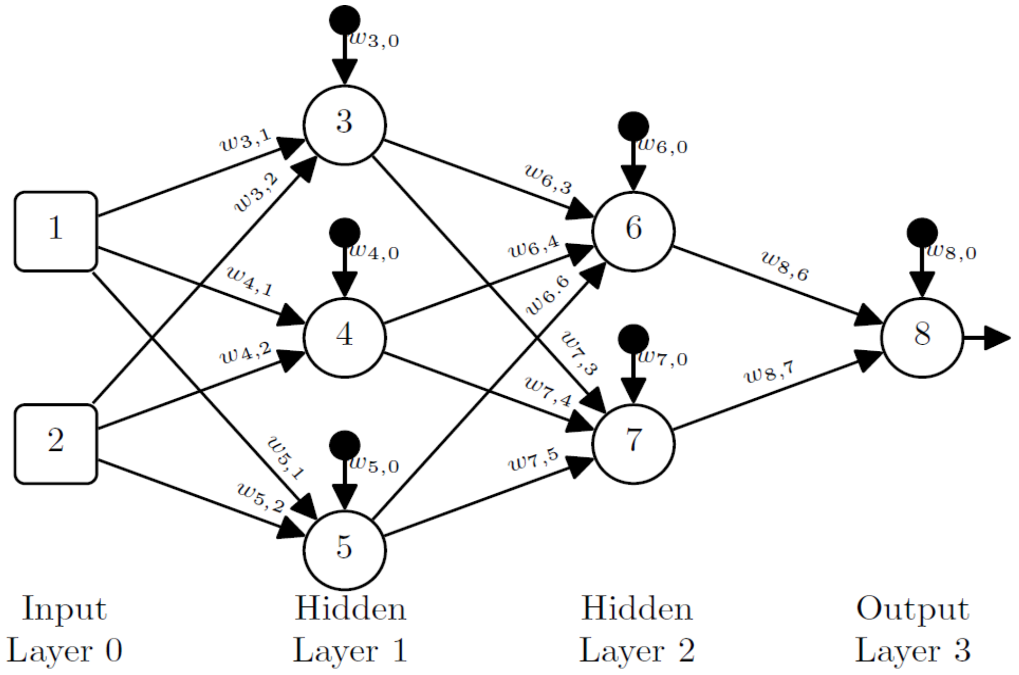
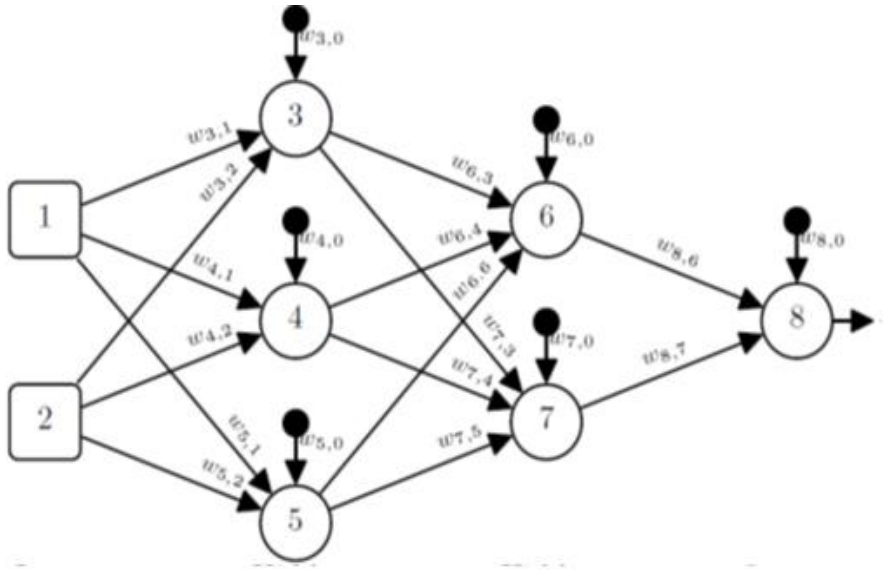


Figure: A schematic of a feedforward artificial neural network

  - The **input layer** features sensing neurons, depicted as squares, which introduce inputs to the network.
  - The **output layer** delivers the network's response to the given inputs.
  - **Hidden layers**, the **internal layers**, are neither part of the input nor the output layers.
  - The **network's depth** is determined by the sum of the hidden layers and the output layer

Artificial Neural Networks (Cont.)

- Neurons can have various **inputs**:
  - An external source,
  - The output activation from another neuron within the network,
  - A consistent dummy input set to 1 (represented by a black circle).

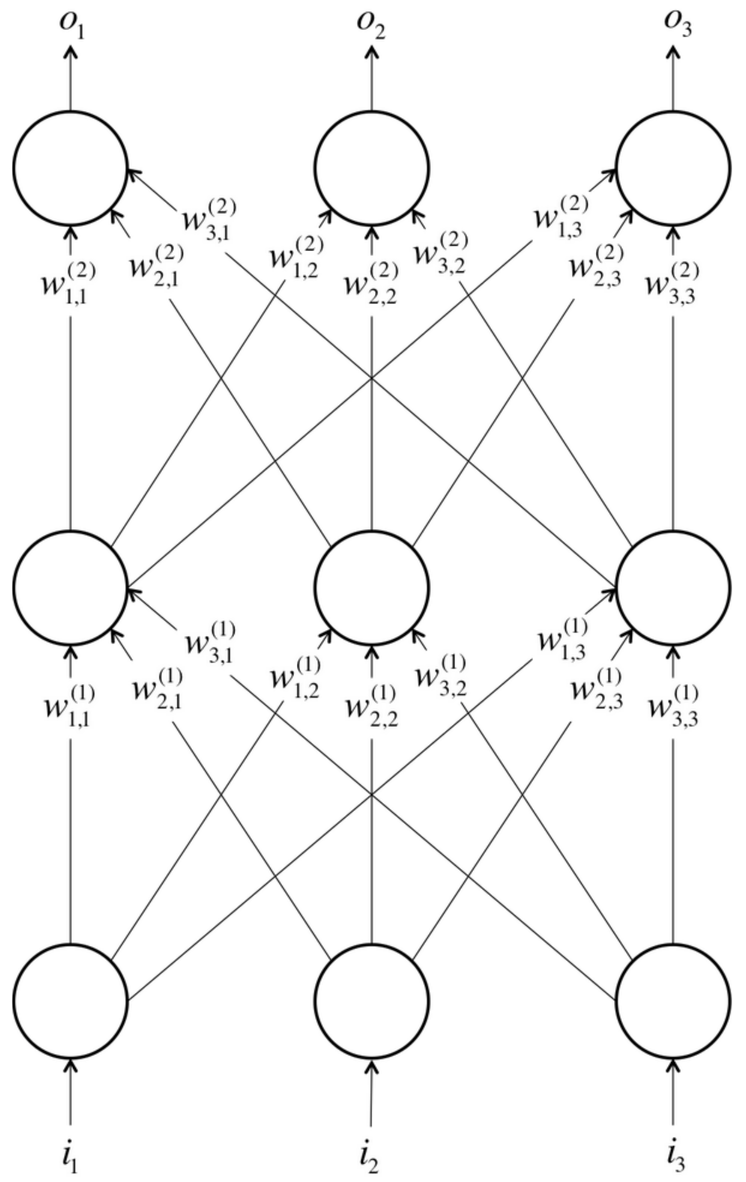


- In the figure:
  - A circle represents a processing neuron. This neuron first calculates a weighted sum of its inputs, then applies an activation function.
  - Arrows illustrate the direction of information flow in the network, with each arrow labeled by its weight.
  - For a weight $w_{i,j}$, the first subscript 'i' denotes the neuron receiving the activation, while the second subscript 'j' identifies the neuron producing the activation.
  - Bias terms, $w_{i,0}$, act as weights for the dummy inputs. The second index for these bias terms is consistently zero.

Network Type: Feed-Forward Network

- **Feed-Forward Network** has a network structure which has no loops or cycles. Activations consistently flow forward, moving through consecutive layers.
  - Connections only traverse from a lower layer to a higher layer.
  - There are no connections between neurons in the same layer
  - There are no connections that transmit data from a higher layer to a lower layer.

Example: A Feed-Forward Neural Network



Output layer

Hidden layer

Input layer

Figure: A simple example of a feed-forward neural network with three layers (input, one hidden, and output) and three neurons per layer

$w_{i,j}^{(k)}$ is the weight of the connection between the i^{th} neuron in the k^{th} layer with the j^{th} neuron in the $k+1^{th}$ layer

Network Type: Fully Connected Network

- **Fully Connected (Dense) Network** has a network structure where every neuron connects to all neurons from the previous layer and forwards its output activation to all neurons in the subsequent layer.

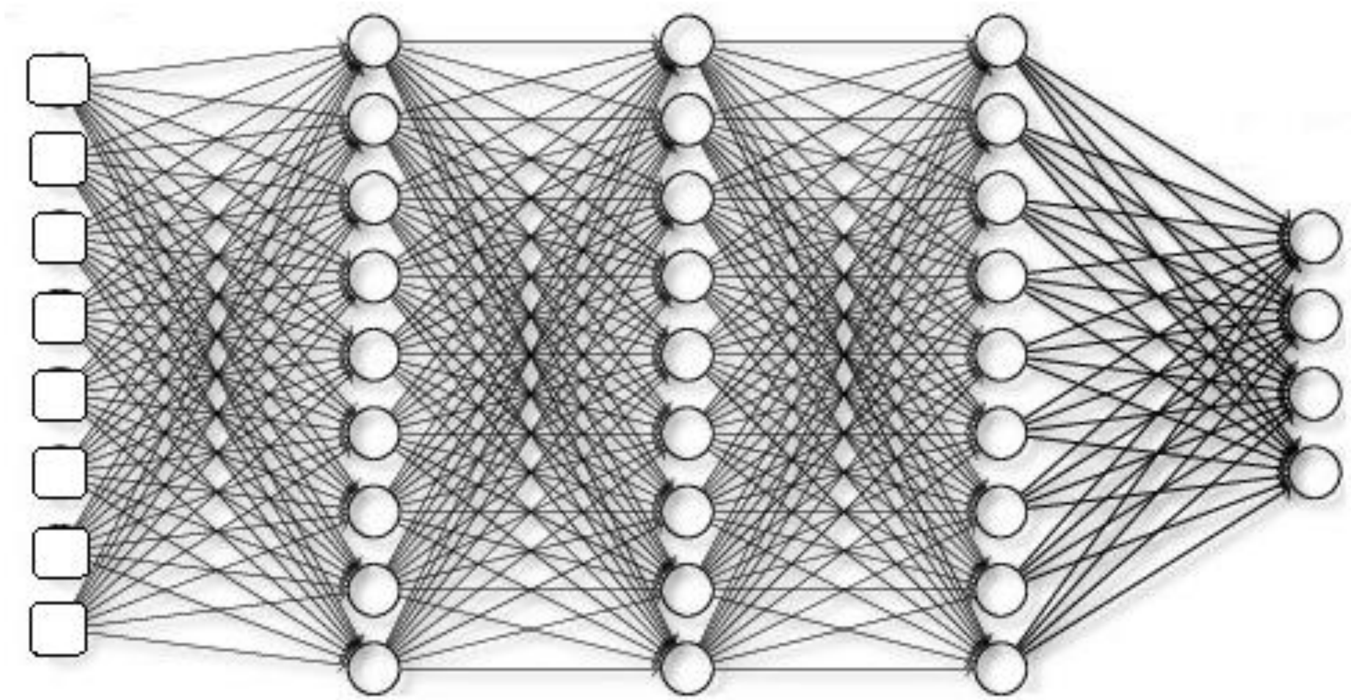


Figure: A fully connected feedforward network

Neural Network as Matrix Operations

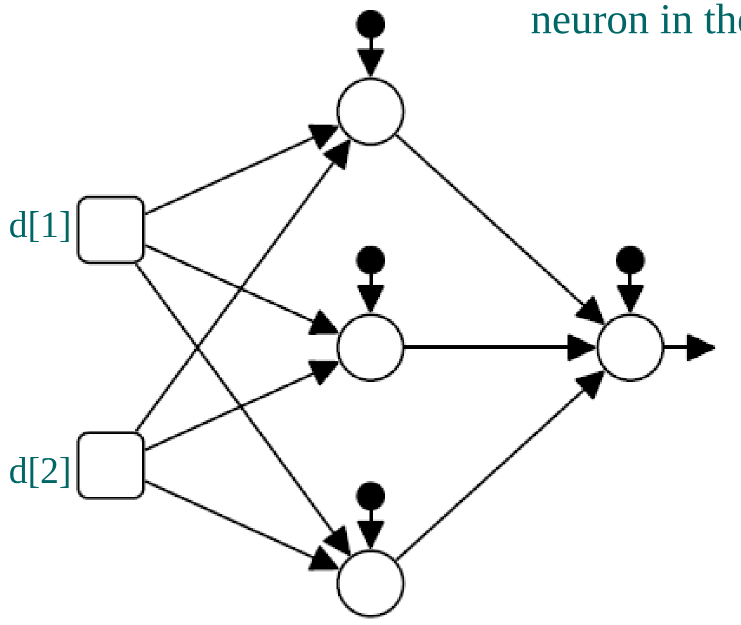
- A neural network's functioning can be visualized both graphically, as a series of interconnected nodes, and mathematically, as **matrix operations**.
  - Let’s consider the input to a layer of the network to be an example vector $\mathbf{d} = [\mathrm{d}[1], \mathrm{d}[2], \ldots \mathrm{d}[\mathrm{n}]]$ and a dummy feature d[0]=1 which is often introduced to handle biases.
  - We’d like to find the activation $\mathbf{a} = [\mathrm{a}_1, \mathrm{a}_2, \ldots, a_m]$ produced by propagating the input through ***m* neurons.**
  - We can express this as a simple **matrix multiply** if we construct a **weight matrix W** of size $m \times (n+1)$ including a bias vector.
    - In this matrix, each column corresponds to a neuron, where the j^{th} neuron element of the column corresponds to the weight of the connection pulling in the j^{th} neuron element of the input.
  - The output $\mathbf{a} = [\mathrm{a}_1, \mathrm{a}_2, \ldots, a_m]$ is

$$\mathbf{a} = \boldsymbol{\varphi}\left(\mathbf{W}^{\mathrm{T}}\mathbf{d}\right),$$

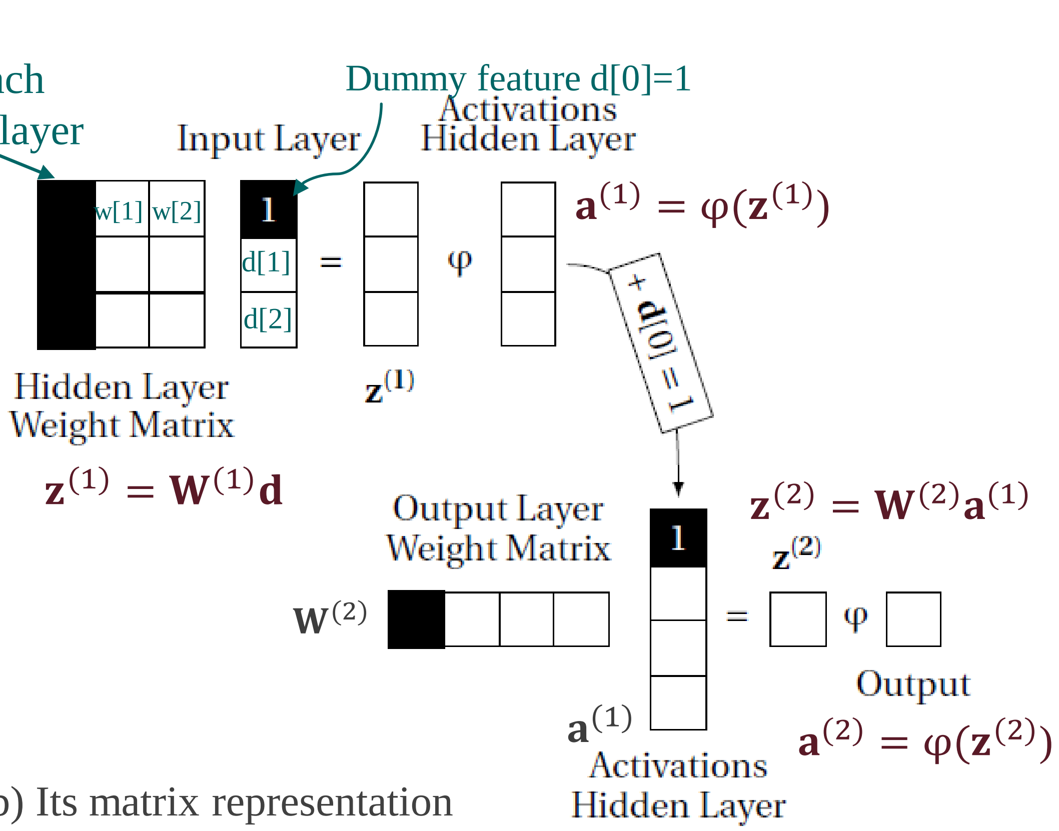
, where the activation function is applied to the vector elementwise.

Matrix Operations with a Single Example

- With a single example d=<d[1], d[2]>



(a) A graph-based representation of a neural network



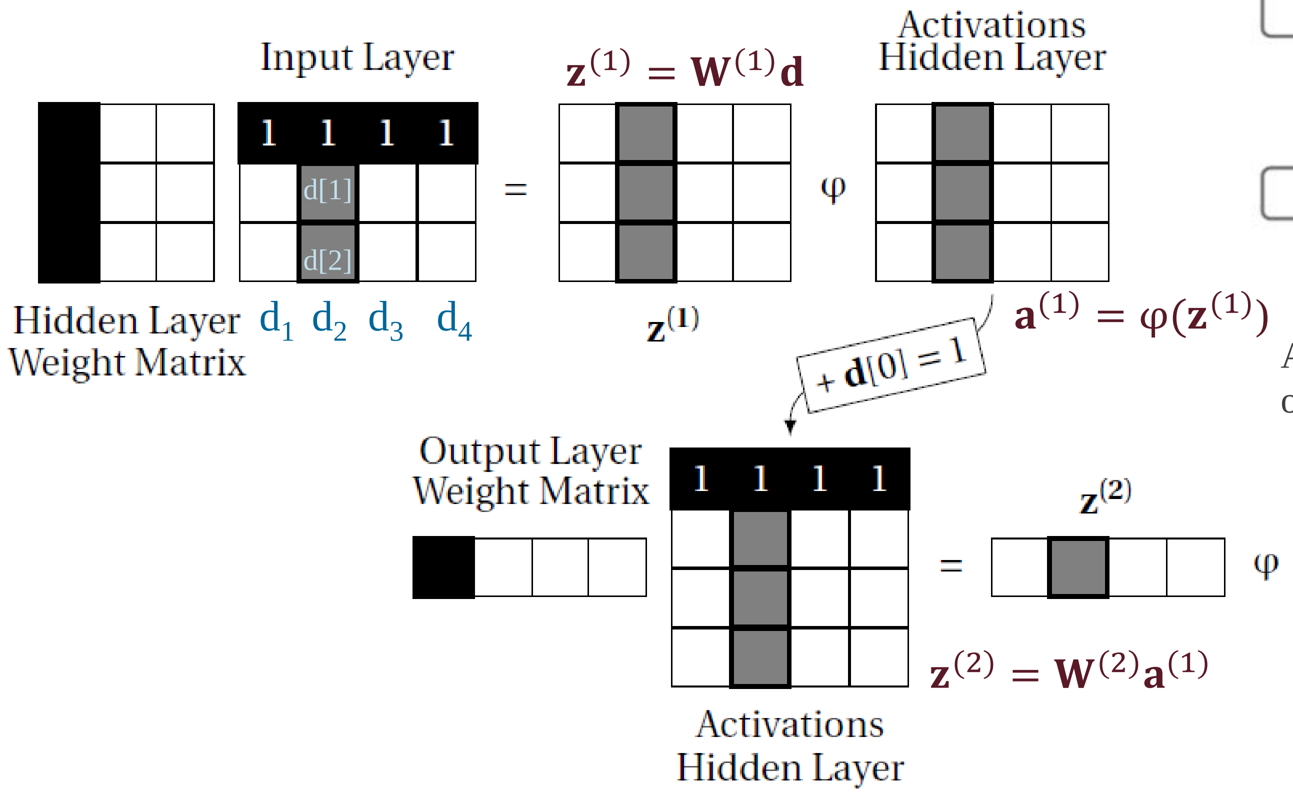
The bias term weights for each neuron in the layer
Dummy feature d[0]=1
Input Layer
Activations Hidden Layer
w[1]
w[2]
1
d[1]
d[2]
=
φ
$\mathbf{a}^{(1)} = \varphi(\mathbf{z}^{(1)})$
Hidden Layer Weight Matrix
$\mathbf{z}^{(1)}$
$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{d}$
+ d[0] = 1
Output Layer Weight Matrix
$\mathbf{W}^{(2)}$
1
=
φ
$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)}$
$\mathbf{z}^{(2)}$
Output
$\mathbf{a}^{(1)}$
Activations Hidden Layer
$\mathbf{a}^{(2)} = \varphi(\mathbf{z}^{(2)})$

(b) Its matrix representation

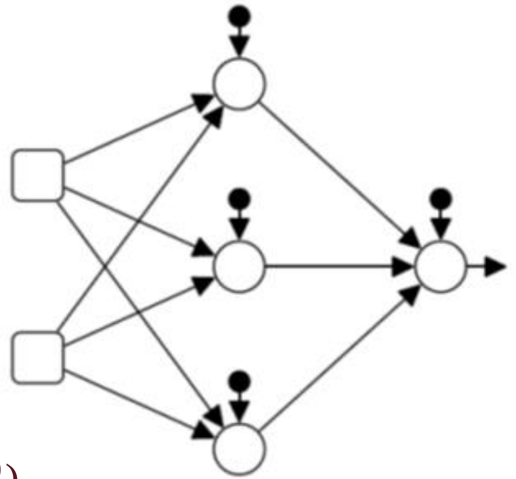
Figure: An illustration of the correspondence between graphical and matrix representations of a neural network

Matrix Operations with Batch of Multiple Examples

- With four examples, $\{d_1, d_2, d_3, d_4\}$
  , where d_i=< $d_i[1]$, $d_i[2]$>



Input Layer
$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{d}$
Activations Hidden Layer
1 1 1 1
d[1]
d[2]
=
φ
Hidden Layer Weight Matrix
d_1 d_2 d_3 d_4
$\mathbf{z}^{(1)}$
$\mathbf{a}^{(1)} = \varphi(\mathbf{z}^{(1)})$
+ d[0] = 1
Output Layer Weight Matrix
1 1 1 1
$\mathbf{z}^{(2)}$
=
φ
$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)}$
Output
$\mathbf{a}^{(2)} = \varphi(\mathbf{z}^{(2)})$
Activations Hidden Layer



A graph-based representation of a neural network

Figure: An illustration of how a batch of instances can be processed in parallel using matrix operations.

Matrix Representation of Neural Network

- In the previous figure,
  - **Input Layer**: The matrix on the left side, labeled as the "Input Layer", represents these **four examples in matrix form**, with each column being one of the input vectors (d_1, d_2, d_3, d_4). Every data instance is a 2-dimensional vector.
  - **Hidden Layer Weight Matrix** and **Output Layer Weight Matrix**: These are representations of the weight matrices associated with the connections between the input layer and the hidden layer, and between the hidden layer and the output layer, respectively. These matrices determine how the input data is transformed as it moves through the network.
  - **Activations Hidden Layer:** After the input data is multiplied by the Hidden Layer Weight Matrix and undergoes an activation function φ, it produces activations for the hidden layer..
  - **Output**: The final matrix demonstrates the output produced when the activations of the hidden layer are multiplied by the Output Layer Weight Matrix and undergo another activation function.

Matrix Representation of Neural Network

- By expressing the network operations in matrix form, we simplify the computations and make them amenable for efficient software implementations.
- This matrix representation not only provides a compact way to understand and implement neural networks but also aligns well with modern computing architectures that excel at matrix operations.

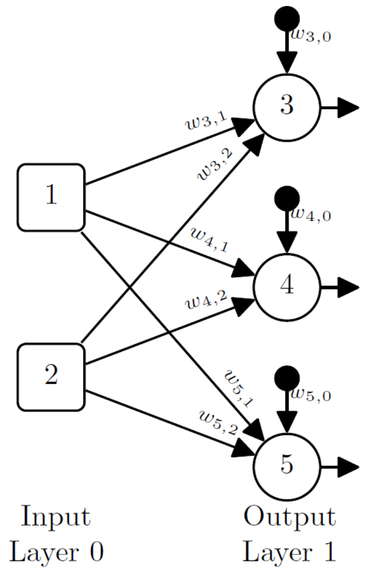
Neural Network Architecture: Key Insights

- **Hidden layer significance:**
  - The hidden layers are positioned between the input layer (first layer of neurons) and the output layer (final layer of neurons).
  - Within the hidden layers, the neural network conducts its intricate computations to identify patterns and solve problems.
  - Analyzing the behavior of these layers can offer insights into the distinctive features the network detects from the input data.
- **Number of neurons:** Not all layers within a neural network must have the same neuron count, and typically, it's not even advised.
- **Hidden layer characteristics:** To encourage the network to capture essential patterns and discard redundant information, hidden layers frequently have fewer neurons than the input layer.
- **Neuron connectivity:** It's not required that every neuron must connect to all neurons in the subsequent layer.
  - Deciding the inter-neuronal connections is more of an art, refined through experience and the specific problem being addressed.

Outline

Weakness of A Single-layer Network

- In single-layer networks, the output neurons cannot represent non-linear decision boundaries.
- Even though the XOR function appears straightforward, a single-layer network, like a perceptron, multivariate linear regression or logistic regression classifier, is unable to define its decision boundary. (Minsky and Papert, 1969)



- Each input value is *True* (1) or *False* (0)
- Each output is *True* (white circle) or *False* (black circle)

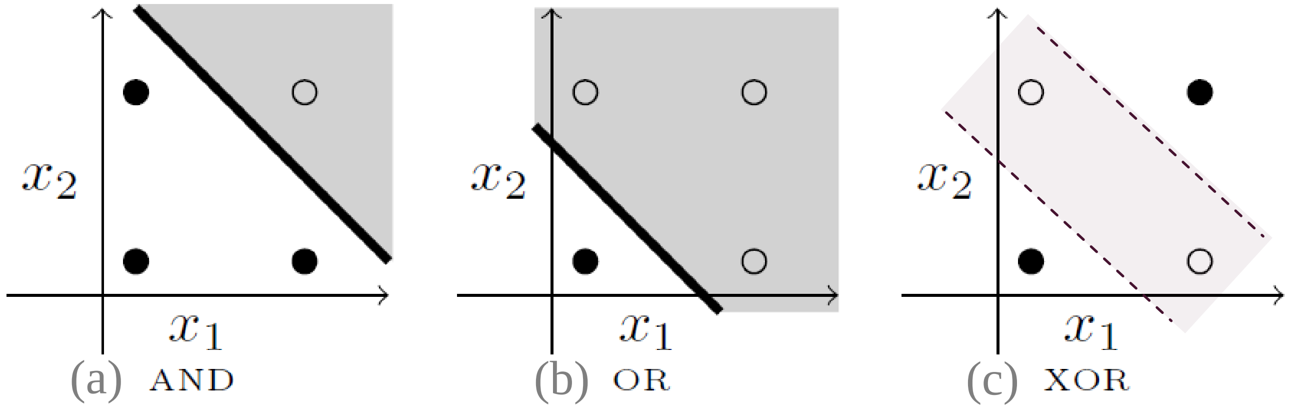


Figure. Activation functions (a) logical AND and (b) OR functions are linearly separable, but (c) the XOR is **not linearly** separable.

A Two-layer Network for XOR

- **By adding a single hidden layer**, the capabilities of single-layer networks can be significantly enhanced, allowing them to act as universal approximators.
- Example: A **two-layer network** with just one hidden layer and a threshold activation function, $\mathbb{M}_w(\mathrm{d}) = \begin{cases} 1 & \text{if } z \geq 1 \\ 0 & \text{otherwise} \end{cases}$, can effectively represent the XOR function.

e.g., [bias, FALSE, FALSE]→[1, 0, 0]

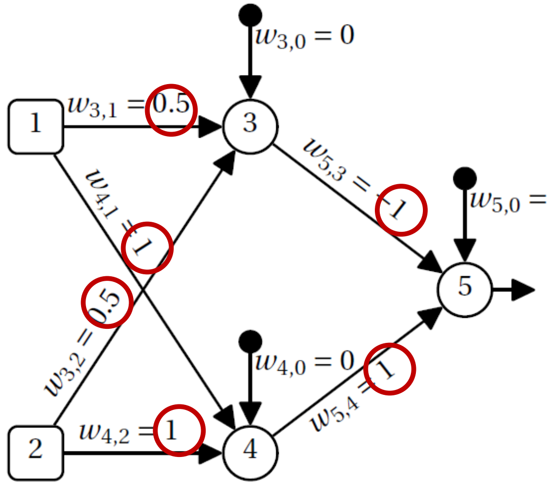
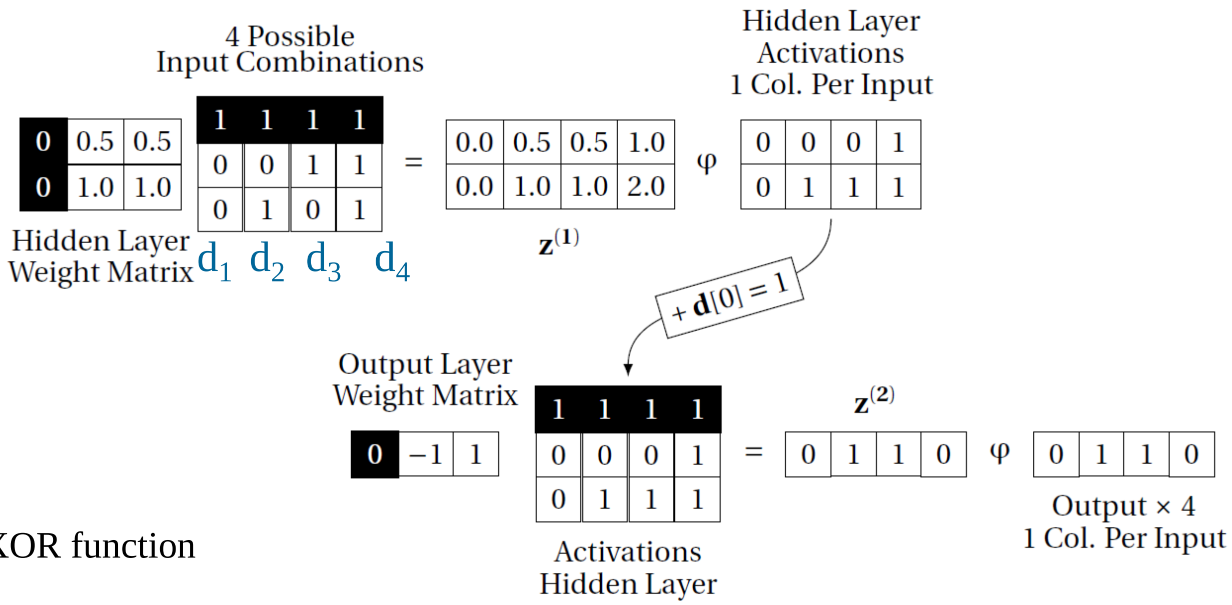


Figure: A two-layer network for the XOR function



4 Possible Input Combinations
Hidden Layer Activations 1 Col. Per Input
Hidden Layer Weight Matrix
d_1 d_2 d_3 d_4
$\mathbf{z}^{(1)}$
φ
$+\,\mathbf{d}[0] = 1$
Output Layer Weight Matrix
$\mathbf{z}^{(2)}$
Activations Hidden Layer
Output × 4 1 Col. Per Input

$$\begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 1.0 & 1.0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.0 & 0.5 & 0.5 & 1.0 \\ 0.0 & 1.0 & 1.0 & 2.0 \end{bmatrix} \ \varphi \ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix} \ \varphi \ \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}$$

Neural Networks with a Hidden Layer

- The ability of a neural network to represent different functions expands as additional layers are integrated.
  - The network's representational capacity denotes the variety of function mappings it can create from input to output as its weights adjust.
- **Networks with at least one hidden layer** possess the capability to approximate a wide range of functions.
- Modern neural networks, categorized under deep learning from the mid-2000s, typically have a more extensive layer structure than those developed from the 1940s to the early 2000s

Network Depth's Importance

- ***Why is it beneficial to add extra layers to a network**, even when incorporating non-linearities*?
  - **Neural networks with a single hidden layer** can universally approximate (bounded) continuous functions if they **either (1)** incorporate complex activation functions like logistic function into their neuronal structure, **or (2)** have a sufficiently broad hidden layer, meaning they possess an adequate number of neurons in that layer.
  - **Neural networks with two hidden layers and smooth activation functions** can represent any function, often fewer neurons than networks with just one hidden layer.
- However, as these networks deepen or incorporate more complex activation functions or additional neurons in the hidden layer, their trainability can become more challenging.

Example: Representational Capacity of a NN

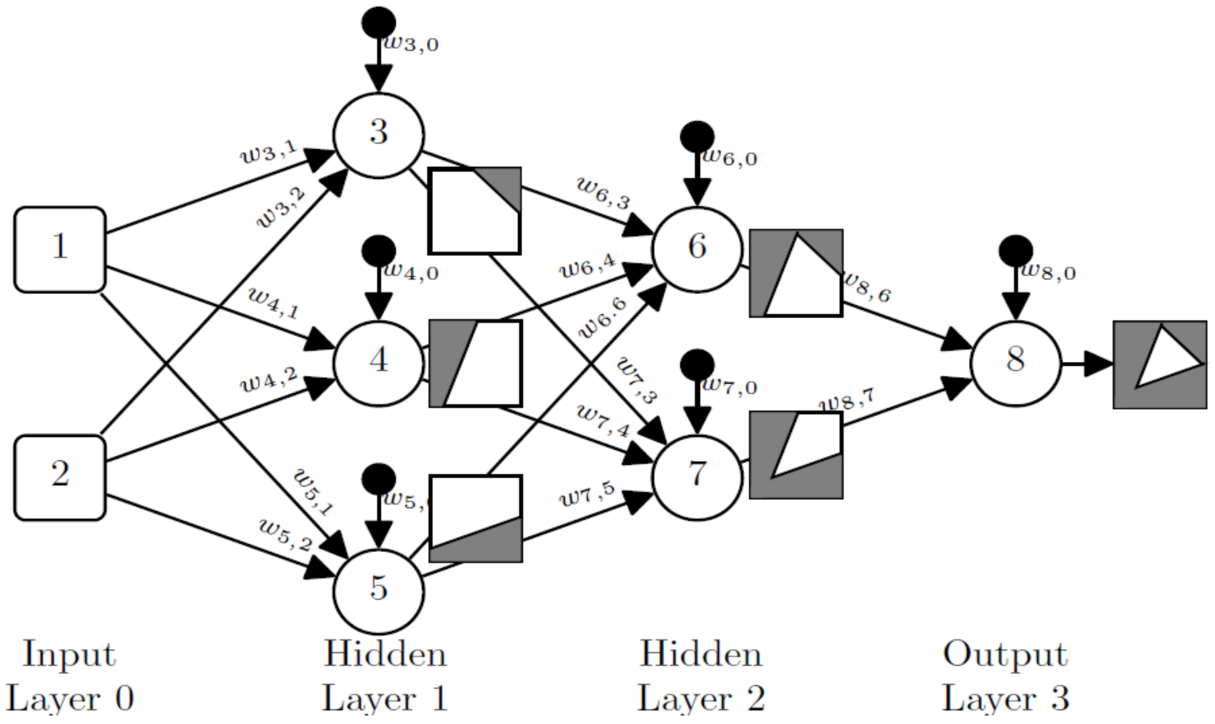


Figure. A network which is able to represent a function that maps to a convex region

- Successive layers in the network leverage the functions learned by preceding layers to form a more complex function (model).
- Utilizing outputs from one or multiple functions as inputs to another function to create a new function is known as **function composition**.

Importance of Non-Linear Activation Functions

- Merely adding layers isn't sufficient to enable networks to represent complex non-linear functions or models.
  - A multi-layer feedforward neural network with only linear neurons is functionally equivalent to a single-layer network of linear neurons.
    - Proof : Kelleher et al.’s book, Ch 8.2.4 (2nd ed.)
- It's unavoidable to introduce non-linearities between these layers.
- By incorporating basic non-linear activation functions, like the logistic or rectifier functions, between layers, neural networks can potentially represent highly complex functions, provided they are sufficiently deep.
- However, the choice of the specific non-linear function can significantly influence the training speed of a deep network.

Network Representation and Training

- A function can be represented by a network if there exists a set of weights that allow the network to implement that function.
- Determining whether a specific network architecture can represent a given function is crucial. This is because the primary objective in predictive modeling is to derive functions (i.e., models) from data.
- However, just because a neural network has the potential to represent a function doesn't ensure its successful training to learn that function. This could be because:
    - The training algorithm might not find the appropriate weight set, or
    - The chosen weights might lead to overfitting of the data.

Outline

- Introduction
- Fundamental: Standard ANN Architecture
- ☞ **Training Neural Networks with a Single Layer**
- Standard Approach: Backpropagation Algorithm
- A Worked Example

The Fast-Food Problem

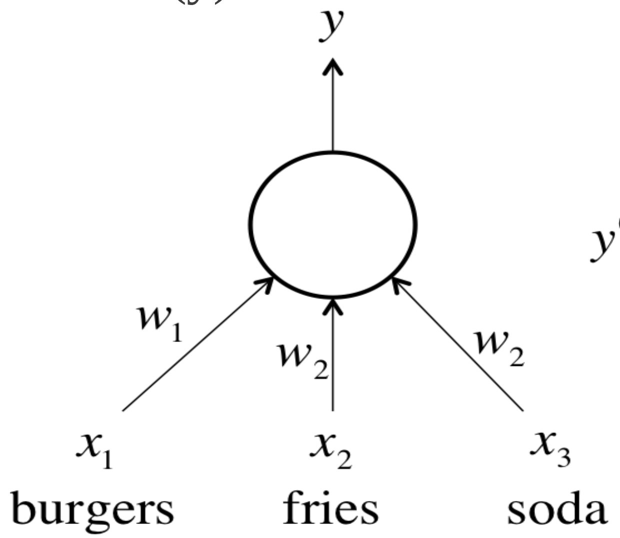
- The setting is a restaurant where we buy meals consisting of burgers, fries, and sodas.
- **Our goal is to determine the cost of each individual item (burger, fries, soda).**
- However, there are a few challenges:
  1. No Price Tags: The individual items (burgers, fries, sodas) don't have separate price tags. This means we can't just look at an item and know how much it costs.
  2. Total Price Only: When we purchase a combination of these items, the cashier only tells us the total price of all items combined.

The Fast-Food Problem (cont.)

- To determine the cost of each individual item, we might buy each item individually on different days:
  - Day 1: Buy only a burger and see how much it costs.
  - Day 2: Buy only fries and check its price.
  - Day 3: Buy only a soda and find out its price.
- However, the approach doesn't work well because, in real-life scenarios, items often have combo prices or discounts when bought together, and prices can fluctuate.

The Fast-Food Problem (cont.)

- If we assume a linear relationship between inputs and outputs, frame the problem using perceptron with a single linear neuron:
  - **Inputs**: The quantities of each item (x_1= number of burgers, x_2 = number of fries, x_3 = number of sodas).
  - **Weights**: These would represent the price of each individual item. (w_1 = price of a burger, w_2 = price of fries, w_3 = price of a soda).
  - **Output**: The total cost (y) of the meal.



$$y^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + w_3 x_3^{(i)}$$

Figure. The neuron we want to train for the fast-food problem

The Fast-Food Problem (cont.)

- Given this setting, we are attempting to train the single linear neuron to predict the price of individual items based on the total price of the meal.
- If we know the total price (y) and we have different combinations of items (x_1, x_2, x_3), we can adjust the weights (w_1, w_2, w_3) during training to approximate the cost of each item.

Training Perceptron

- ***How to train the perceptron?***
- The **perceptron training algorithm** proposed by Rosenblatt was largely inspired by Hebb's rule.
  - *Hebb's Postulate* (also known as ***Hebb's rule*** or *Hebbian learning*) (1949) : Donald Hebb suggested that when a neuron triggers another neuron often, the synaptic connection between them strengthens. Learning in the brain happens through alterations in the connections between neurons.

Perceptron Training Algorithm

- The perceptron is fed one training example at a time, and for each example it makes its predictions.
- For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

Perceptron learning rule (weight update)

$$w_{i,j}^{(next\ step)} = w_{i,j} + \epsilon(y_j - \hat{y}_j)x_i$$

- $\boldsymbol{w_{i,j}}$ is the connection weight between the j^{th} input and the i^{th} neuron
- x_j is the j^{th} input value of the current training example.
- $\hat{y}$ is the output of the i^{th}output neuron for the current training example.
- y_i is the target output of the i^{th} output neuron for the current training example.
- $\boldsymbol{\epsilon}$ is the learning rate

- The perceptron learning algorithm strongly **resembles stochastic gradient descent.**

Optimizing Neural Network Weights

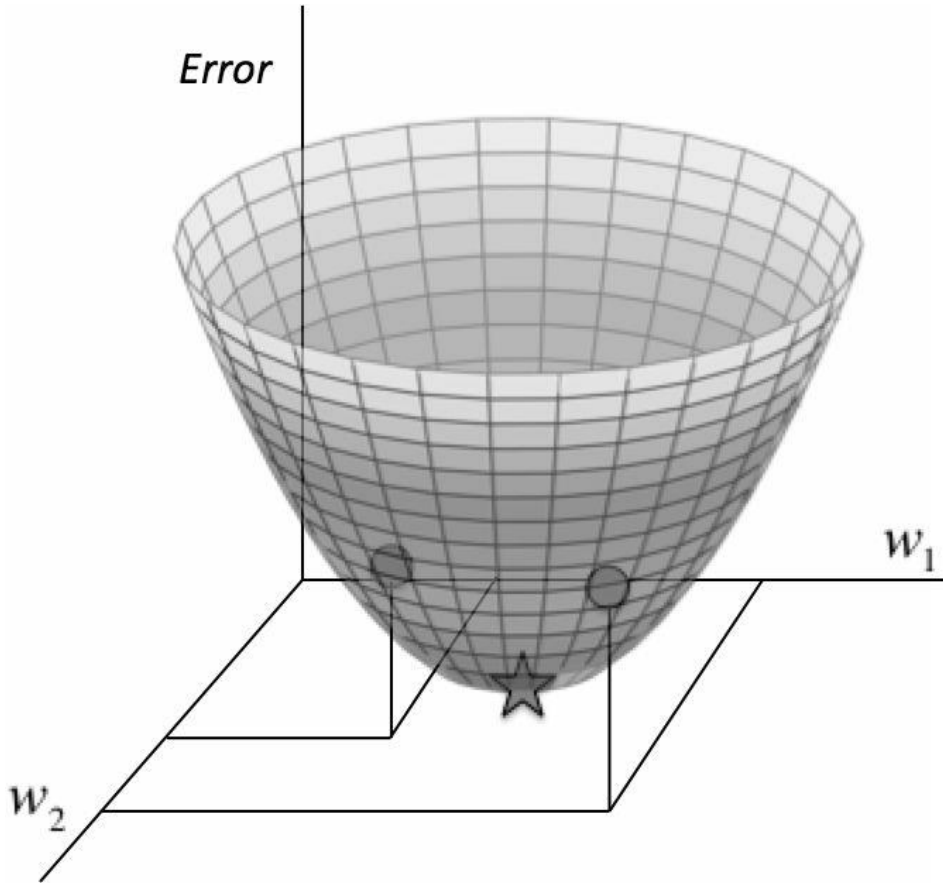
- Assume we possess a vast collection of training data.
- We can then determine the neural network's output for the i^{th} training data using the formula:, $y^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + w_3 x_3^{(i)}$.
- Our objective is to optimize the neuron's weights to minimize errors, such as the square error E, across all training data.

$$E = \frac{1}{2} \sum_i (t^{(i)} - y^{(i)})^2$$

- The smaller the value of E, the better the model's performance.
- Our ultimate **goal** is to choose a parameter vector θ (the values for all the weight in the model) that makes the square error E as close to 0 as feasible.

Error Surface

- Let's visualize how we might minimize the squared error over all of the training examples by simplifying the problem.



- For a linear neuron with only two inputs (and thus only two weights, w_1 and w_2), the error space is visualized with a three-dimensional space where the horizontal dimensions correspond to the weights w_1 and w_2, and the vertical dimension corresponds to the value of the error function E.

Figure. The quadratic error surface (a quadratic bowl) for a linear neuron

Error Surface

- The **quadratic bowl** in the three-dimensional space also visualizes the surface as a set of **elliptical contours**, where the minimum error is at the center of the ellipses.
  - Contours correspond to settings of *w*1 and *w*2 that evaluate to the same value of *E*.
  - The closer the contours are to each other, the steeper the slope.
    - In fact, it turns out that the direction of the steepest descent is always perpendicular to the contours.

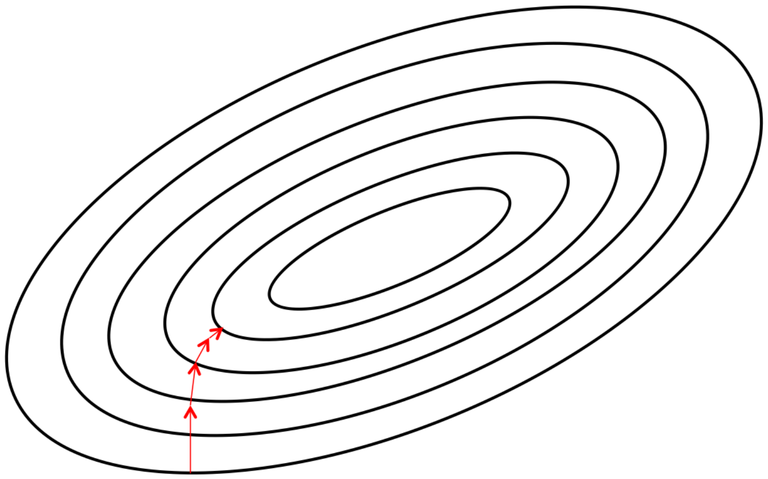


Figure. Visualizing the error surface as a set of contours

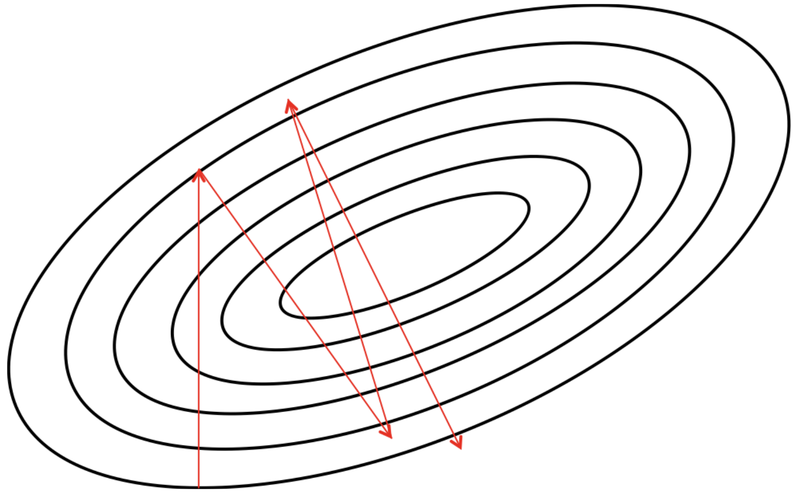
- The direction of the steepest descent is expressed as a vector known as the ***gradient.***

Gradient Descent Algorithm

- **Gradient Descent Algorithm**
  - Randomly initialize the weights of our network so we find ourselves somewhere on the horizontal plane.
  - By evaluating the gradient at our current position, we can find the direction of steepest descent, and we can take a step in that direction.
    - Then we'll find ourselves at a new position that's closer to the minimum than we were before.
  - We can reevaluate the direction of steepest descent by taking the gradient at this new position and taking a step in this new direction.
  - The closer we are to the minimum, the shorter we want to step forward.
    - We know we are close to the minimum, because the surface is a lot flatter,
    - so **we can use the steepness as an indicator of how close we are to the minimum.**
  - Following this strategy will eventually get us to the point of minimum error.

Gradient Descent – Hyperparameter

- In addition to the weight parameters defined in our neural network, the Gradient Descent algorithm requires the ***learning rate*** ϵ hyperparameter.
- We often multiply the gradient by the learning rate to reduce the training time.
- Picking the learning rate is a hard problem
  - If we pick a learning rate that's too small, we risk taking too long during the training process.
  - If we pick a learning rate that's too big, we'll mostly likely start diverging away from the minimum.



Gradient Descent – Delta Rule

- **In order to calculate how to change each weight,**
  - We **evaluate the gradient** (**errorDelta**), which is essentially **the partial derivative of the error with respect to each of the weights.**

$$\frac{\partial E}{\partial w[k]} = \frac{\partial}{\partial w[k]}\left(\frac{1}{2}\sum_i (t_i - y_i)^2\right) = \sum_i (t_i - y_i)\frac{\partial y_i}{\partial w[k]}$$

$$= \sum_i (t_i - y_i)\frac{\partial y_i}{\partial w[k]} = \sum_i (t_i - y_i)x_k$$

  - ***Delta rule***: $\mathbf{w}[k] \leftarrow \mathbf{w}[k] + \epsilon \times$ **errorDelta**$(\mathcal{D}, \mathbf{w}[k])$

, where **errorDelta**$(\mathcal{D}, \mathbf{w}[k]) = \sum_{i=1}^{n}((t_i - y_i) \times \boldsymbol{x}_k)$

- **Applying the *delta rule*, we change the weights at every iteration for training the linear neuron.**

Outline

- Introduction
- Fundamental: Standard ANN Architecture
- Training Feed-Forward Neural Networks
- ☞ **Standard Approach: Backpropagation Algorithm**
- A Worked Example

Basic Concept: Error Gradient in Neural Networks

- The **error gradient** quantifies how much each weight in the network should be adjusted to minimize the error between the predicted and actual outputs.
- It indicates the **direction** (positive or negative) **and magnitude** of change needed for each weight.
- Importance of the Error Gradient
  - Neural networks learn by adjusting their weights to minimize the error in predictions.
  - The error gradient provides the necessary information on how to adjust these weights effectively.

Two Types of Error Gradients

- Error gradient for each neuron (δ_k)
  - Represents how the network error changes with respect to adjustments in **a neuron's weighted sum calculation**.
- Error gradient for each weight
  - Represents how the network error changes with respect to adjustments in **the network's weights.**
  - Every weight in the network has its corresponding error gradient.
  - These gradients guide the weight updates within the network.

Basic Concept: Gradient Descent Algorithm

- **Gradient descent algorithm** is an optimization technique used to adjust the weights of the network in the direction that minimizes the error.
- The gradient descent uses the **error gradient** to determine how much and in which direction each weight should be adjusted.

Training Neural Networks with Multiple Layers

- ***How do we train networks with multiple layers of neurons?***
- To update weights during each iteration, we compute the error gradient for neurons in the output layer by directly comparing their activations with the anticipated outputs.
- However direct computation of an error gradient for hidden neurons isn't feasible. Instead, we must determine how much a hidden neuron contributed to the network's total error.
- This assessment for each neuron in a network is termed the **blame assignment** problem.
- The **backpropagation** algorithm addresses the blame assignment problem.

Backpropagation in Neural Networks

- **Backpropagation** is a supervised learning algorithm used **for training multi-layered neural networks.**
- Instead of determining the error at just the output layer, backpropagation calculates how much each neuron in previous layers contributed to the error at the output, with propagating the error backward through the network.
- Backpropagation is a systematic way of adjusting the weights in a neural network by determining how much each neuron contributed to the overall error and making corrections accordingly.
  - After employing the backpropagation algorithm to address the blame assignment problem for all neurons in the network, we can apply the weight update rule from the Gradient Descent algorithm to adjust the weights of each neuron.

General Structure of Backpropagation Algorithm

1. Backpropagation **initiates with the weight initialization** of the network.
   - Typically, weights are set to random values near zero, e.g., by drawing from a normal distribution with μ=0.0 and σ=0.1.
2. **The weight update then proceeds iteratively**, **utilizing a two-step procedure** that allocates blame (or error gradient) to each neuron:
   1) For every training sample, the algorithm initially predicts (**forward pass**) and quantifies the resultant error.
   2) It then traverses each layer in reverse order to compute the error contribution from every connection (**backward pass**).
   3) Ultimately, the connection weights are adjusted to minimize the error (via the **Gradient Descent** step).

Backpropagation: Forward Pass

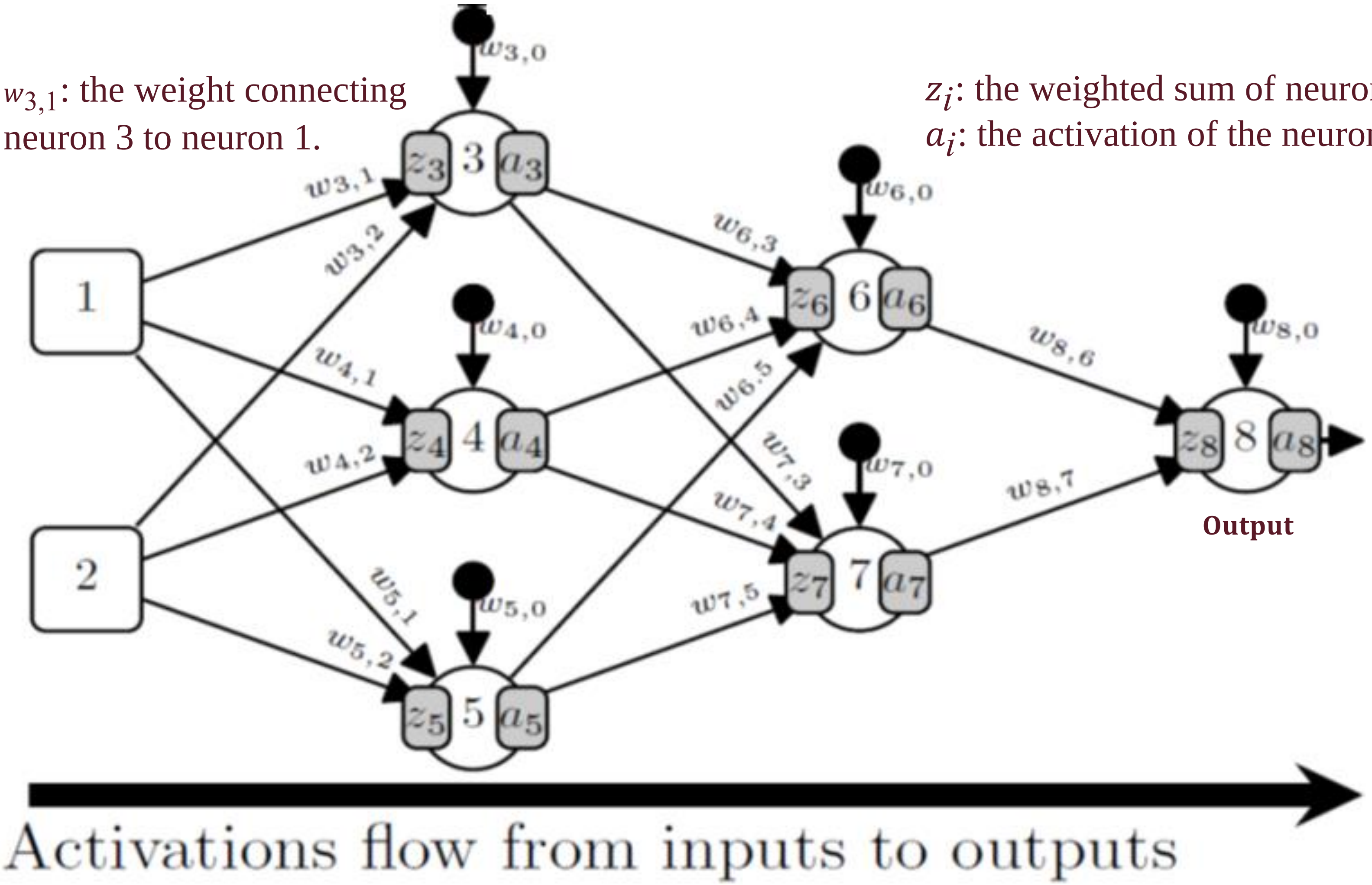
- **Forward Pass:** The network receives an input, and activations propagate through it until an output emerges.
  1. <u>Each example (or every mini-batch,</u> typically comprising 32 examples) <u>feeds into the network's input layer</u>, which channels it to the first hidden layer.
  2. <u>The algorithm calculates the outcomes of every neuron</u> within the first hidden layer for each example in the mini-batch.
  3. <u>These results get relayed to the subsequent layer.</u> Their outputs are determined and forwarded to the next layer.
  4. This pattern continues until the final layer, <u>the output layer, generates its outcome.</u>
- This procedure is exactly like prediction-making, except that all intermediary results are retained for the backward phase.

Forward Pass

$w_{3,1}$: the weight connecting neuron 3 to neuron 1.

z_i: the weighted sum of neuron i,
a_i: the activation of the neuron i

Inpt



Output

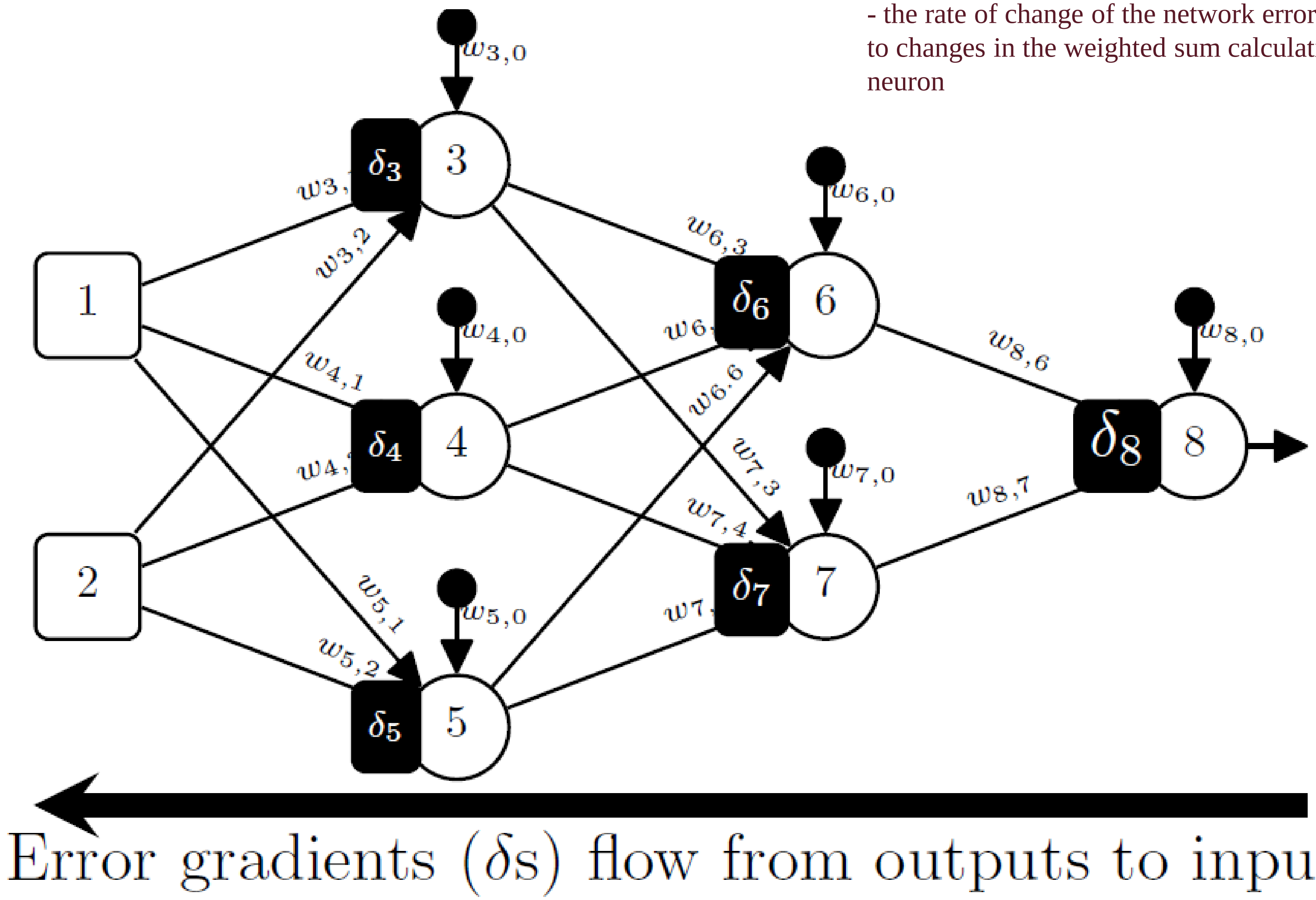
Backpropagation: Backward Pass

- **Backward Pass:**

1. The error in the network's output is determined by comparing the predicted output from the forward pass with the target output from the dataset.
   - To measure this discrepancy, the algorithm employs a **loss function** that assesses the difference between the desired and the actual outputs, yielding a measure of the error.
2. Subsequently, the algorithm evaluates the contribution of each output connection to this error.
3. Moving backwards through the layers, the algorithm determines how much of the error contributions are associated with each connection in the preceding layer, continuing this assessment until it reaches the input layer.
   - This backward propagation of errors allows for an efficient computation of the **error gradient**, denoted as δ (*delta*), for all connection weights in the network.
4. At the end of this process, the algorithm employs a **Gradient Descent** step to modify all the network connection weights based on the freshly calculated error gradients.

Backward Pass

δ_i: the error gradient for a neuron i
- the rate of change of the network error with respect to changes in the weighted sum calculation of the neuron



Calculation of Error Gradient δs in Backward Pass

- **δ for a neuron in the output layer**
  - In the simplest scenario, the error gradient $\boldsymbol{\delta}$ is determined by subtracting the neuron's activation from the desired output provided in the dataset.
- **δ for a neuron in hidden layers**
  - Allocate a fraction of the δ from each output neuron to every hidden neuron connected to it.
    - The δ for a neuron in the final hidden layer is computed by summing the portions of δs backpropagated to it from all output layer neurons it links to.
  - After determining the δs for all neurons in the last hidden layer, the procedure is repeated to compute the δs for the neurons in the preceding hidden layer.
  - Upon completion of this process, a δ has been determined for every neuron in the network.

Calculation of δ for a Neuron

- **The error gradient for neuron** k, δ_k, is the rate of change in the network error concerning alterations in the neuron's weighted sum calculation.

  - ε : the network' error at its output layer
  - z_k: the weighted sum calculation for neuron k
  - a_k: the neuron's activation

$$\delta_k = \frac{\partial \mathcal{E}}{\partial z_k}$$

- It can be expressed into a production two terms:

$$\delta_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k}$$

- The first term $\frac{\partial a_k}{\partial z_k}$ (**Activation Function's Derivative**) is the rate of change of the neuron's activation with respect to changes in the weighted sum determined at the neuron. This computation is consistent, regardless of the neuron being in the output or hidden layer.
- The second term $\frac{\partial \mathcal{E}}{\partial a_k}$ (**Sensitivity of the Error to Activation)** is the rate of change of the network's error with respect to the activation changes of the neuron. The method of computing this term varies based on whether the neuron belongs to the output layer or one of hidden layers.

Calculation of $\frac{\partial \mathcal{E}}{\partial a_k}$ for an Output Neuron k

In determining $\delta_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k}$,

- The second term $\frac{\partial \mathcal{E}}{\partial \boldsymbol{a_k}}$ for output neurons is influenced by the specific error function (loss function) used during the training.
- **Example**: If the error function is **Sum of Squared Errors (**L_2 norm**)**,

$$L_2(\mathbb{M}_{\mathbf{w}}, \mathcal{D}) = \frac{1}{2}\sum_{i=1}^{n}\left(t_i - \mathbb{M}_{\mathrm{w}}(\mathrm{d}_i)\right)^2 = \frac{1}{2}\sum_{i=1}^{n}(t_i - a_i)^2 = \varepsilon$$

, where $d_i \in \mathcal{D}$ (an element of the training dataset), t_i is the corresponding true target value, $\mathbb{M}_w(d_i)$ is the predicted output for d_i, which is the activation of the output layer, a_i

$\frac{\partial \mathcal{E}}{\partial \boldsymbol{a_k}}$ for a output neuron k is the difference between true target value and the predicted activation of the neuron, with a negative sign to move in the direction of reducing the error.

By the chain rule.

$$\frac{\partial \mathcal{E}}{\partial a_k} = \frac{\partial L_2(\mathbb{M}_{\mathbf{w}}, \mathbf{d})}{\partial \mathbb{M}_{\mathbf{w}}(\mathbf{d})} \times \frac{\partial \mathbb{M}_{\mathbf{w}}(\mathbf{d})}{\partial a_k} = t - \mathbb{M}_{\mathbf{w}}(\mathbf{d}) = t_k - a_k$$

However, often in neural networks, we consider the negative of this derivative value to compute the gradient descent update, since we want to move in the opposite direction of the gradient to minimize the error.

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{a_k}} = (\boldsymbol{t_k} - \boldsymbol{a_k}) \times -\mathbf{1} = \boldsymbol{a_k} - \boldsymbol{t_k}$$

Calculation of δ_k for an Output Neuron k

For the calculation of $\delta_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k}$,

- If we use the logistic activation function, the first term $\frac{\partial a_k}{\partial z_k}$ is

$$\boxed{\frac{\partial a}{\partial z} = \frac{d}{dz} logistic(z) = logistic(z) \times (1 - logistic(z))}$$

$$\frac{d}{dz} logistic(z = 0) = logistic(0) \times (1 - logistic(0))$$
$$= 0.5 \times (1 - 0.5)$$
$$= 0.25$$

- Thus, $\boldsymbol{\delta_k}$ for an output neuron k is

$$\delta_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k}$$
$$= \frac{\partial a_k}{\partial z_k} \times -(t_k - a_k)$$
$$= \underbrace{\frac{d}{dz} logistic(z)}_{\text{Assuming a logistic activation function}} \times -(t_k - a_k)$$
$$= \boxed{\underbrace{(logistic(z) \times (1 - logistic(z)))}_{\text{Assuming a logistic activation function}} \times -(t_k - a_k)}$$

Calculation of $\frac{\partial \mathcal{E}}{\partial a_k}$ for a Hidden Neuron

In calculation of $\delta_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k}$,

- Again, the term $\frac{\partial \mathcal{E}}{\partial \boldsymbol{a_k}}$ indicates how the total error $\mathcal{E}$ changes with adjustments to the activation a_k of the neuron k.
- In the case of $\frac{\partial \mathcal{E}}{\partial \boldsymbol{a_k}}$ of **hidden neurons**, this influence is not direct, but it comes from the subsequent layers or neurons that it feeds into.
- The connection between the activation $a_k$of a hidden neuron k and the overall network error $\mathcal{E}$ is established by constructing a "chain" of such sensitivities through the activations of the following (downstream) neurons.

Calculation of $\frac{\partial \mathcal{E}}{\partial a_k}$ for a Hidden Neuron (cont.)

In calculation of $\delta_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k}$ for hidden neurons,

- For every neuron *i* that the hidden neuron *k* connects forward to, $\begin{tikzcd} k \arrow[r] & i \end{tikzcd}$, we need to consider:
  - The error gradient, δ_i, for the neuron *i*.
  - The contribution of neuron *k* to the weighted sum z_i of neuron *i* , that is $\boldsymbol{w_{i,k}}$
- To compute $\frac{\partial \mathcal{E}}{\partial a_k}$ for a hidden neuron *k*, we sum the product of :
  - the weight $\boldsymbol{w_{i,k}}$ from neuron *k* to all neurons *i* it directly influences, and
  - the error gradients $\boldsymbol{\delta_i}$ for those neurons *i*

$$\frac{\partial \mathcal{E}}{\partial a_k} = \sum_{i=1}^{n} w_{i,k} \times \delta_i$$

Calculation of δ_k for a Hidden Neuron k

In calculation of $\delta_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k}$,

$$\frac{\partial \mathcal{E}}{\partial a_k} = \sum_{i=1}^{n} w_{i,k} \times \delta_i$$

$$\begin{aligned}
\delta_k &= \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k} \\
&= \frac{\partial a_k}{\partial z_k} \times \left(\sum_{i=1}^{n} w_{i,k} \times \delta_i \right) \\
&= \underbrace{\frac{d}{dz} logistic(z)}_{\text{Assuming a logistic activation function}} \times \left(\sum_{i=1}^{n} w_{i,k} \times \delta_i \right)
\end{aligned}$$

$$\boxed{= \underbrace{(logistic(z) \times (1 - logistic(z)))}_{\text{Assuming a logistic activation function}} \times \left(\sum_{i=1}^{n} w_{i,k} \times \delta_i \right)}$$

After δ Calculation

- After calculating the error gradient δ for every neuron in the network, the backward pass is complete.
- Subsequently, we must update the weights of the network using the δs in accordance with **the gradient descent weight update rule.**
- A core principle regarding weight adjustments in a neural network is that a weight should be modified in relation to the sensitivity of the network error to changes in that weight.
  - When the network error displays minimal sensitivity to changes in a weight, it suggests that the error isn't influenced by that weight, indicating the weight had little to no impact on the error.
  - On the other hand, if the network error demonstrates substantial sensitivity to adjustments in the weight, it signifies that the weight plays a pivotal role in the error, meaning that weight contributes notably to the error.

Error Gradient of a Weight

- To determine **the sensitivity of the network error concerning a weight on an incoming connection to a neuron,** we multiply the neuron's δ by the activation forwarded on that connection.
- That is, $\frac{\partial \mathcal{E}}{\partial w_{i,k}}$, representing **the rate of change of the network error with respect to the weight connecting neuron $\boldsymbol{k}$ to neuron $\boldsymbol{i}$**, $$\begin{tikzcd} k \arrow[r] & i \end{tikzcd}$$, is:

Utilizing the chain rule, $\frac{\partial \mathcal{E}}{\partial w_{i,k}} = \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i} \times \frac{\partial z_i}{\partial w_{i,k}}$

$$= \delta_i \times \frac{\partial z_i}{\partial w_{i,k}} = \boldsymbol{\delta_i \times a_k}$$

- ***How do we use this value to update a weight?***

Updating the Weights in a Network

- In neuronal networks, the weight update rule is commonly presented as:

$$w_{i,k} \leftarrow w_{i,k} - \alpha \times gradient$$

, where α: learning rate

  - If the gradient is positive (meaning increasing the weight would increase the error), the weight will decrease, and vice versa.

- **Weight update after processing a single training example**:

$$w_{i,k} \leftarrow w_{i,k} - \alpha \times \delta_i \times a_k$$

  - If a_k is positive, the weight $w_{i,k}$decreases. If a_k is negative, the weight $w_{i,k}$ increases.

- **Weight update for the entire input example set**:
  - The update is the sum of the error gradients for the weight $w_{i,k}$ over one complete pass.

$$\Delta w_{i,k} = \sum_{j=1}^{m} \delta_{i,j} \times a_{k,j}$$

, where $\delta_{i,j}$ is the δ value for neuron i for example j

$$w_{i,k} \leftarrow w_{i,k} - \alpha \times \Delta w_{i,k}$$, where α a running rate

Different Types for Weight Updating

- When weights are updated after processing each individual training example, it's termed as **stochastic gradient descent**, also known as **on-line** or **sequential gradient descent**.
- The process of updating weights after going through all the examples in the dataset is termed as **batch gradient descent**.
- The **mini-batch gradient descent** method divides the dataset into multiple subsets and applies batch gradient descent on each subset.
  - Typically, these mini-batches are formed by random sampling.
  - Most of the time, mini-batches are of equal size.
  - This method processes the entire training dataset multiple times.
  - Every single complete traversal through the training set is referred to as an ***epoch***.
  - Determining the optimal values for the batch size and the learning rate hyper-parameter α usually requires some trial-and-error experimentation.

Backpropagation: The Algorithm

- The algorithm requires
  - A set of training examples ($\mathcal{D}$)
  - A learning rate α which controls the speed at which the algorithm reaches convergence.
  - A batch size B specifying the number of examples contained in each batch
  - A criterion for determining convergence

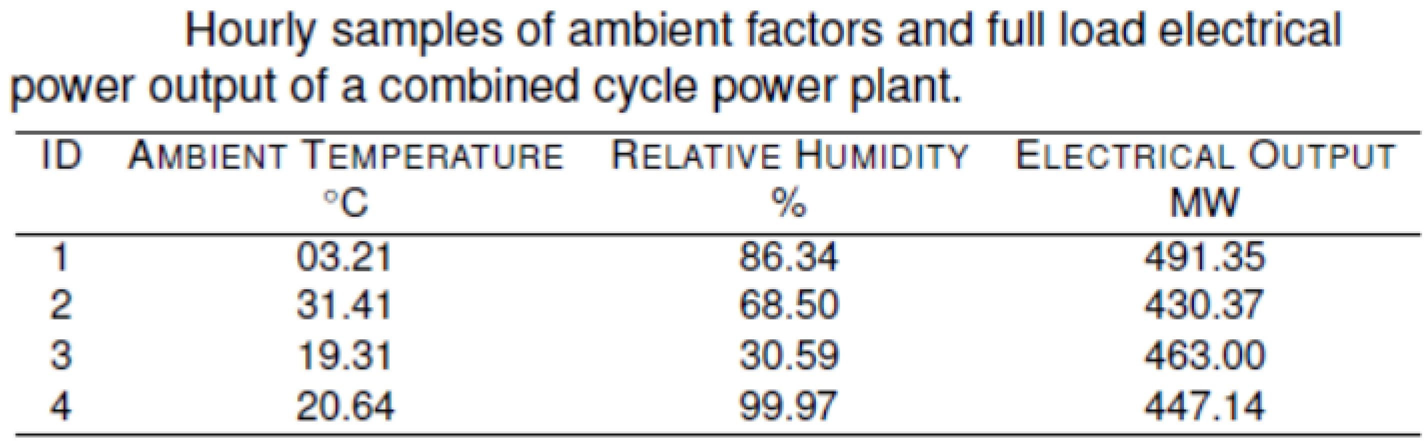
```
1: Shuffle $\mathcal{D}$ and create the mini-batches: $[(\mathbf{X}^{(1)}, \mathbf{Y}^{(1)}), \ldots, (\mathbf{X}^{k}, \mathbf{Y}^{k})]$
2: Initialize the weight matrices for each layer: $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}$
3: repeat                                                        ▷ Each repeat loop is one epoch
4:     for t=1 to number of mini-batches do                      ▷ Each for loop is one iteration
5:         $\mathbf{A}^{(0)} \leftarrow \mathbf{X}^{(t)}$        $\mathbf{A}^{(0)}$: the matrix containing the activations of Layer 0 (input layer)
6:         for $l$=1 to L do
7:             $\mathbf{v} \leftarrow [1_0, \ldots 1_m]$         ▷ Create $\mathbf{v}$ the vector of bias terms
8:             $\mathbf{A}^{(l-1)} \leftarrow [\mathbf{v}; \mathbf{A}^{(l-1)}]$    $l$: the number of layers    ▷ Insert $\mathbf{v}$ into the activation matrix
9:             $\mathbf{Z}^{(l)} \leftarrow \mathbf{W}^{l}\mathbf{A}^{(l-1)}$    $\mathbf{Z}^{(l)}$: the matrix containing the activations of Layer $l$
10:            $\mathbf{A}^{(l)} \leftarrow \varphi(\mathbf{Z}^{(l)})$    $\varphi$: activation function    ▷ Elementwise application of $\varphi$ to $\mathbf{Z}^{(l)}$
11:        end for
12:        for each weight $w_{i,k}$ in the network do
13:            $\Delta w_{i,k} = 0$
14:        end for
15:        for each example in the mini-batch do                 ▷ Backpropagate the $\delta$s
16:            for each neuron $i$ in the output layer do
17:                $\delta_i = \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i}$
18:            end for
19:            for $l$ = L-1 to 1 do
20:                for each neuron $i$ in the layer $l$ do
21:                    $\delta_i = \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i}$
22:                end for
23:            end for
24:            for each weight $w_{i,k}$ in the network do
25:                $\Delta w_{i,k} = \Delta w_{i,k} + (\delta_i \times a_k)$    $\Delta w_{i,k}$ is the sum of the error gradients for the weight $w_{i,k}$
26:            end for
27:        end for
28:        for each weight $w_{i,k}$ in the network do
29:            $w_{i,k} \leftarrow w_{i,k} - \alpha \times \Delta w_{i,k}$
30:        end for
31:    end for
32:    shuffle($[(\mathbf{X}^{(1)}, \mathbf{Y}^{(1)}), \ldots, (\mathbf{X}^{k}, \mathbf{Y}^{k})]$)
33: until convergence occurs
```

Outline

- Introduction
- Fundamental: Standard ANN Architecture
- Training Neural Networks with a Single Layer
- Standard Approach: Backpropagation Algorithm
- ☞ **A Worked Example**: Using backpropagation to train a feedforward network for a regression task.

Dataset for a Regression Task

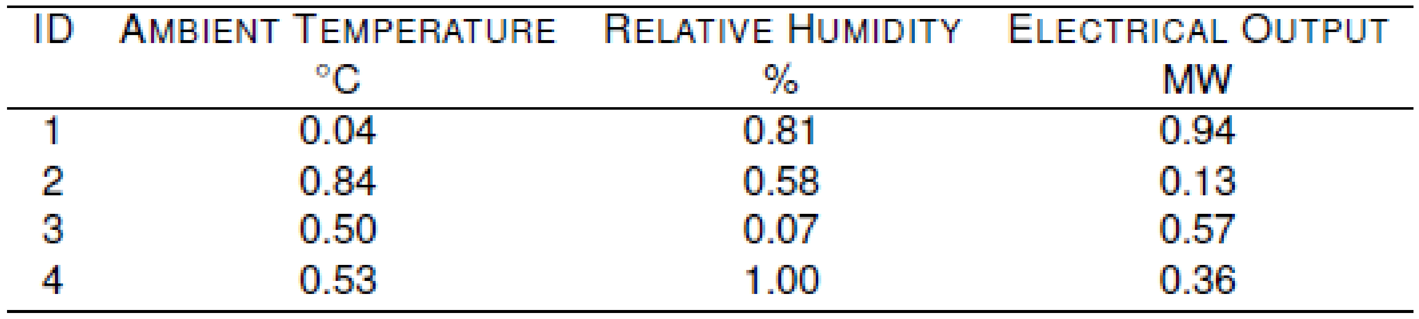
- A training dataset



Hourly samples of ambient factors and full load electrical power output of a combined cycle power plant.

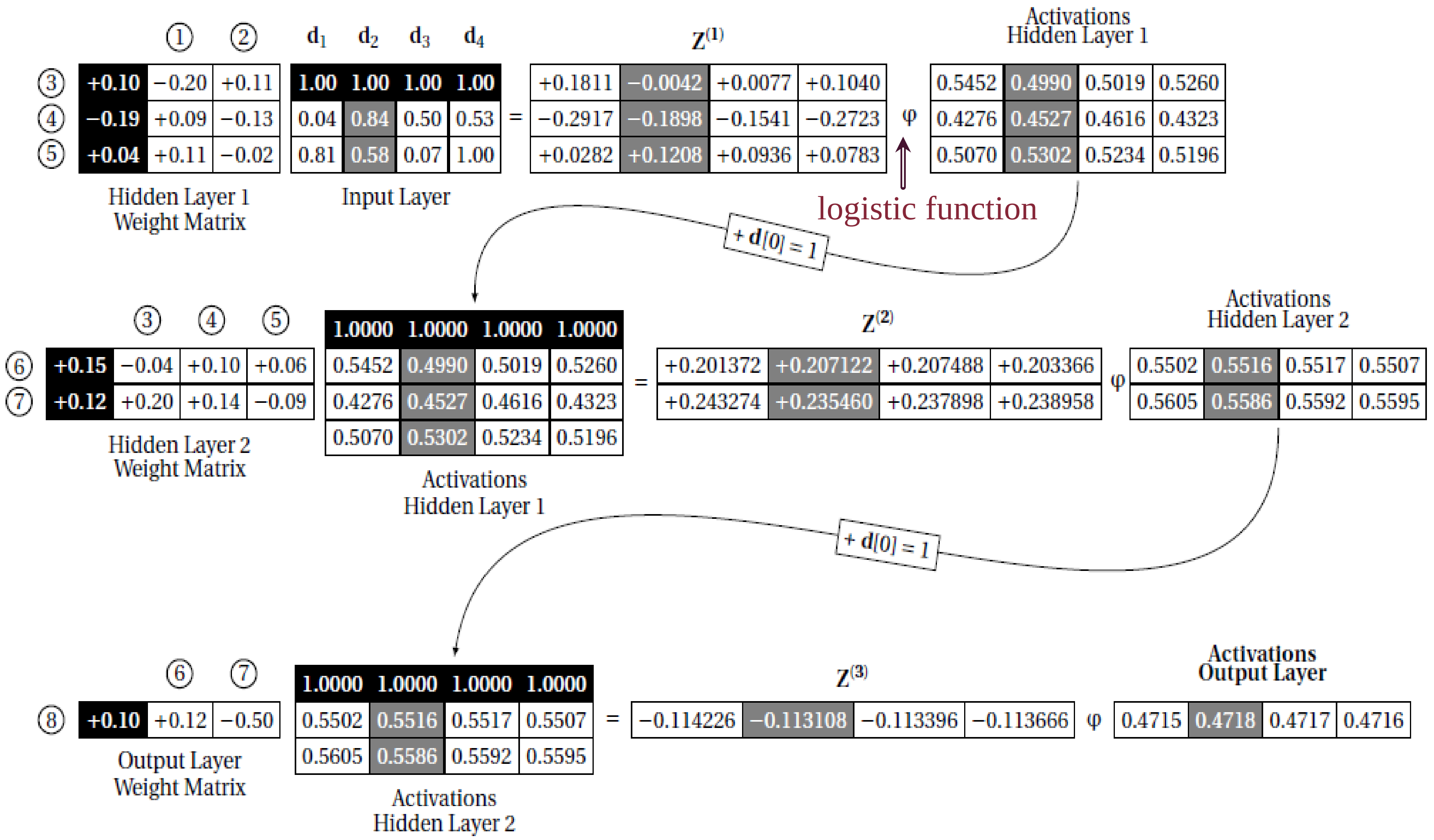
ID	Ambient Temperature °C	Relative Humidity %	Electrical Output MW
1	03.21	86.34	491.35
2	31.41	68.50	430.37
3	19.31	30.59	463.00
4	20.64	99.97	447.14

- The range-normalized data in [0, 1]



ID	Ambient Temperature °C	Relative Humidity %	Electrical Output MW
1	0.04	0.81	0.94
2	0.84	0.58	0.13
3	0.50	0.07	0.57
4	0.53	1.00	0.36

Forward Pass with Four Examples

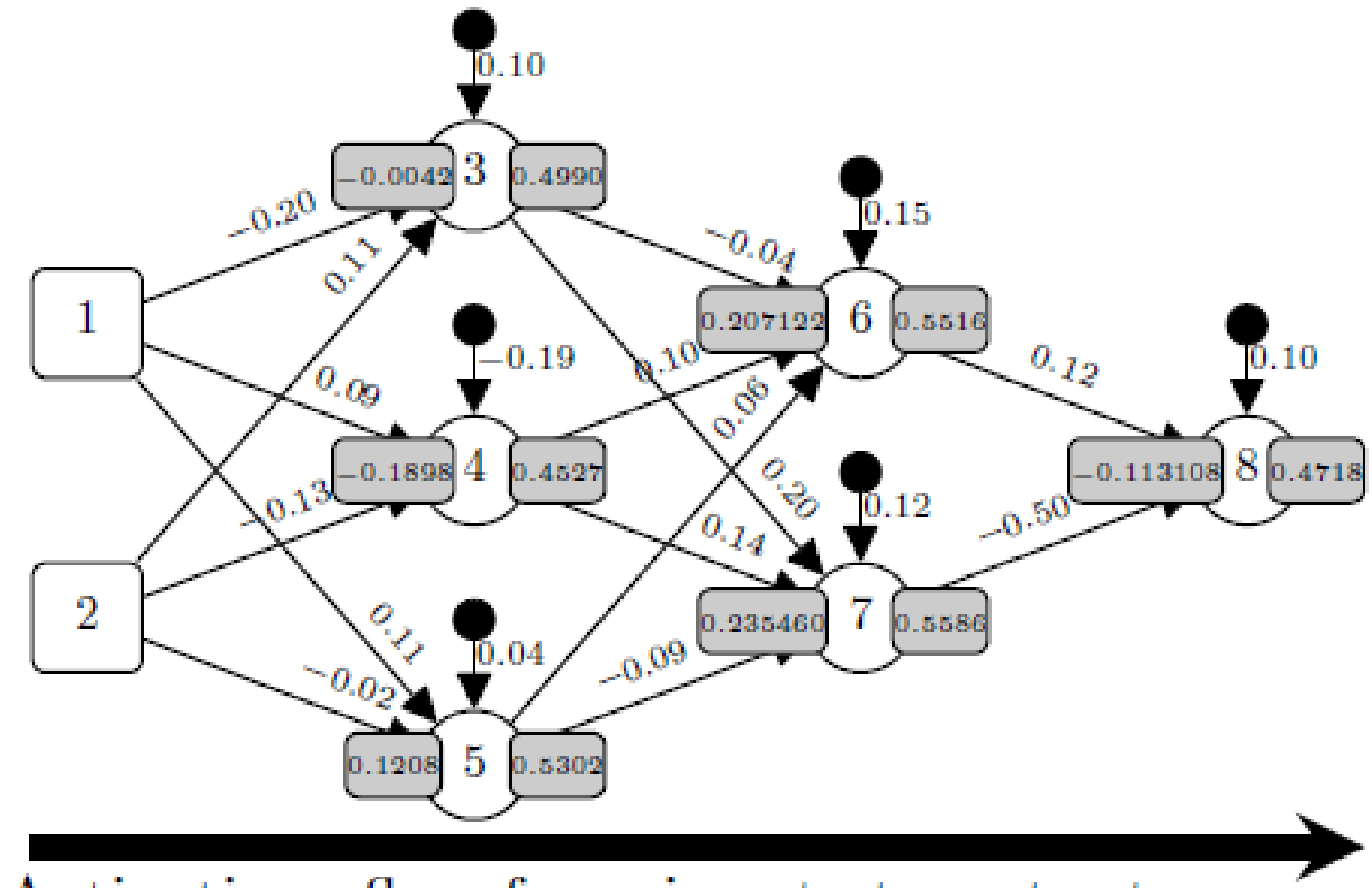


$$\underset{\text{Hidden Layer 1 Weight Matrix}}{\begin{array}{c|ccc} & & ① & ② \\ \hline ③ & +0.10 & -0.20 & +0.11 \\ ④ & -0.19 & +0.09 & -0.13 \\ ⑤ & +0.04 & +0.11 & -0.02 \end{array}} \underset{\text{Input Layer}}{\begin{array}{cccc} d_1 & d_2 & d_3 & d_4 \\ \hline 1.00 & 1.00 & 1.00 & 1.00 \\ 0.04 & 0.84 & 0.50 & 0.53 \\ 0.81 & 0.58 & 0.07 & 1.00 \end{array}} = \overset{Z^{(1)}}{\begin{bmatrix} +0.1811 & -0.0042 & +0.0077 & +0.1040 \\ -0.2917 & -0.1898 & -0.1541 & -0.2723 \\ +0.0282 & +0.1208 & +0.0936 & +0.0783 \end{bmatrix}} \xrightarrow{\varphi} \overset{\text{Activations Hidden Layer 1}}{\begin{bmatrix} 0.5452 & 0.4990 & 0.5019 & 0.5260 \\ 0.4276 & 0.4527 & 0.4616 & 0.4323 \\ 0.5070 & 0.5302 & 0.5234 & 0.5196 \end{bmatrix}}$$

$$\underset{\text{Hidden Layer 2 Weight Matrix}}{\begin{array}{c|cccc} & & ③ & ④ & ⑤ \\ \hline ⑥ & +0.15 & -0.04 & +0.10 & +0.06 \\ ⑦ & +0.12 & +0.20 & +0.14 & -0.09 \end{array}} \underset{\text{Activations Hidden Layer 1}}{\begin{bmatrix} 1.0000 & 1.0000 & 1.0000 & 1.0000 \\ 0.5452 & 0.4990 & 0.5019 & 0.5260 \\ 0.4276 & 0.4527 & 0.4616 & 0.4323 \\ 0.5070 & 0.5302 & 0.5234 & 0.5196 \end{bmatrix}} = \overset{Z^{(2)}}{\begin{bmatrix} +0.201372 & +0.207122 & +0.207488 & +0.203366 \\ +0.243274 & +0.235460 & +0.237898 & +0.238958 \end{bmatrix}} \xrightarrow{\varphi} \overset{\text{Activations Hidden Layer 2}}{\begin{bmatrix} 0.5502 & 0.5516 & 0.5517 & 0.5507 \\ 0.5605 & 0.5586 & 0.5592 & 0.5595 \end{bmatrix}}$$

$$\underset{\text{Output Layer Weight Matrix}}{\begin{array}{c|ccc} & & ⑥ & ⑦ \\ \hline ⑧ & +0.10 & +0.12 & -0.50 \end{array}} \underset{\text{Activations Hidden Layer 2}}{\begin{bmatrix} 1.0000 & 1.0000 & 1.0000 & 1.0000 \\ 0.5502 & 0.5516 & 0.5517 & 0.5507 \\ 0.5605 & 0.5586 & 0.5592 & 0.5595 \end{bmatrix}} = \overset{Z^{(3)}}{\begin{bmatrix} -0.114226 & -0.113108 & -0.113396 & -0.113666 \end{bmatrix}} \xrightarrow{\varphi} \overset{\text{Activations Output Layer}}{\begin{bmatrix} 0.4715 & 0.4718 & 0.4717 & 0.4716 \end{bmatrix}}$$

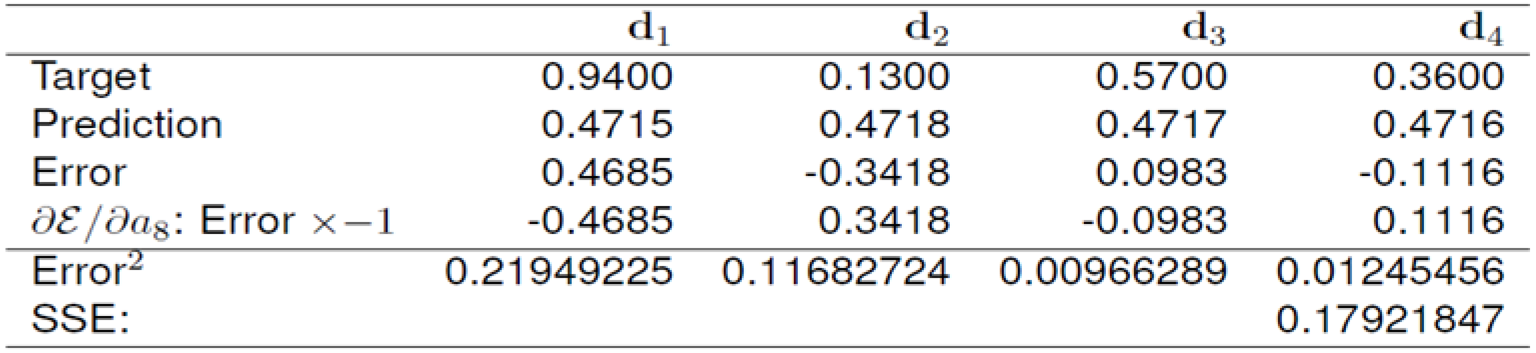
The Forward Pass of $\mathbf{d}_2$



Activations flow from inputs to outputs

Example Errors and SSE after Forward Pass

- This table shows the errors per example for a logistic network post-forward pass, including individual derivatives with respect to activations and the Sum of Squared Errors (SSE).



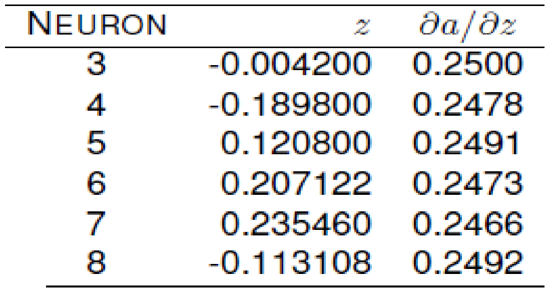
	$\mathbf{d}_1$	$\mathbf{d}_2$	$\mathbf{d}_3$	$\mathbf{d}_4$
Target	0.9400	0.1300	0.5700	0.3600
Prediction	0.4715	0.4718	0.4717	0.4716
Error	0.4685	-0.3418	0.0983	-0.1116
$\partial\mathcal{E}/\partial a_8$: Error $\times -1$	-0.4685	0.3418	-0.0983	0.1116
Error^2	0.21949225	0.11682724	0.00966289	0.01245456
SSE:				0.17921847

- $\frac{\partial \mathcal{E}}{\partial a_k} = Error \times -1 = -(t_k - a_k),$
- SSE: Sum of squared errors.

Calculation of Error Gradients through Backward Pass

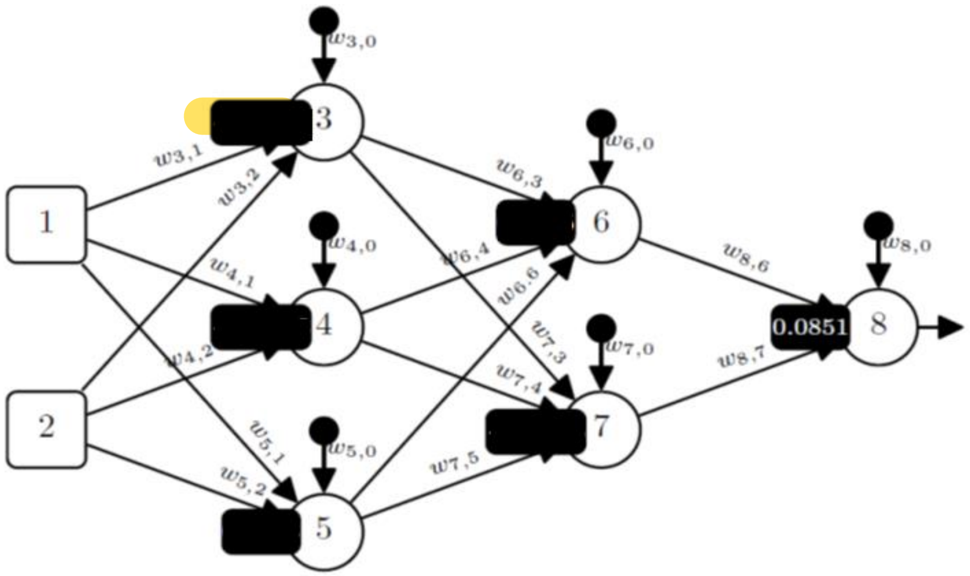
- Error Gradient of Output Nodes For $\mathbf{d}_2$
  - $\frac{\partial \mathcal{E}}{\partial a_8} = 0.3418$ from the previous table.
  - $$\delta_8 = \frac{\partial \mathcal{E}}{\partial a_8} \times \frac{\partial a_8}{\partial z_8}$$
    $$= 0.3418 \times 0.2492$$
    $$= 0.0852$$

$\frac{\partial a_k}{\partial z_k}$ are computed during the forward pass



Neuron	z	$\partial a/\partial z$
3	-0.004200	0.2500
4	-0.189800	0.2478
5	0.120800	0.2491
6	0.207122	0.2473
7	0.235460	0.2466
8	-0.113108	0.2492

$$\frac{\partial a_k}{\partial z_k} = \frac{d}{dz_k} logistic(z_k)$$
$$= \left(logistic(z_k) \times \left(1 - logistic(z_k)\right)\right)$$



Error Gradients of Hidden Nodes for $\mathbf{d}_2$

- For $\mathbf{d}_2$

NEURON	z	$\partial a/\partial z$
3	-0.004200	0.2500
4	-0.189800	0.2478
5	0.120800	0.2491
6	0.207122	0.2473
7	0.235460	0.2466
8	-0.113108	0.2492

$$\boldsymbol{\delta_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k}}$$

, where $$\frac{\partial \mathcal{E}}{\partial a_k} = \sum_{i=1}^{n} w_{i,k} \times \delta_i$$

$$\frac{\partial a_k}{\partial z_k} = \frac{d}{dz_k} logistic(z_k) = (logistic(z_k) \times (1 - logistic(z_k)))$$

$$\begin{aligned} \delta_6 &= \frac{\partial \mathcal{E}}{\partial a_6} \times \frac{\partial a_6}{\partial z_6} \\ &= \left(\sum \delta_i \times w_{i,6}\right) \times \frac{\partial a_6}{\partial z_6} \\ &= (\delta_8 \times w_{8,6}) \times \frac{\partial a_6}{\partial z_6} \\ &= (0.0852 \times 0.12) \times 0.2473 \\ &= 0.0025 \end{aligned}$$

$$\begin{aligned} \delta_7 &= \frac{\partial \mathcal{E}}{\partial a_7} \times \frac{\partial a_7}{\partial z_7} \\ &= \left(\sum \delta_i \times w_{i,7}\right) \times \frac{\partial a_7}{\partial z_7} \\ &= (\delta_8 \times w_{8,7}) \times \frac{\partial a_6}{\partial z_6} \\ &= (0.0852 \times -0.50) \times 0.2466 \\ &= -0.0105 \end{aligned}$$

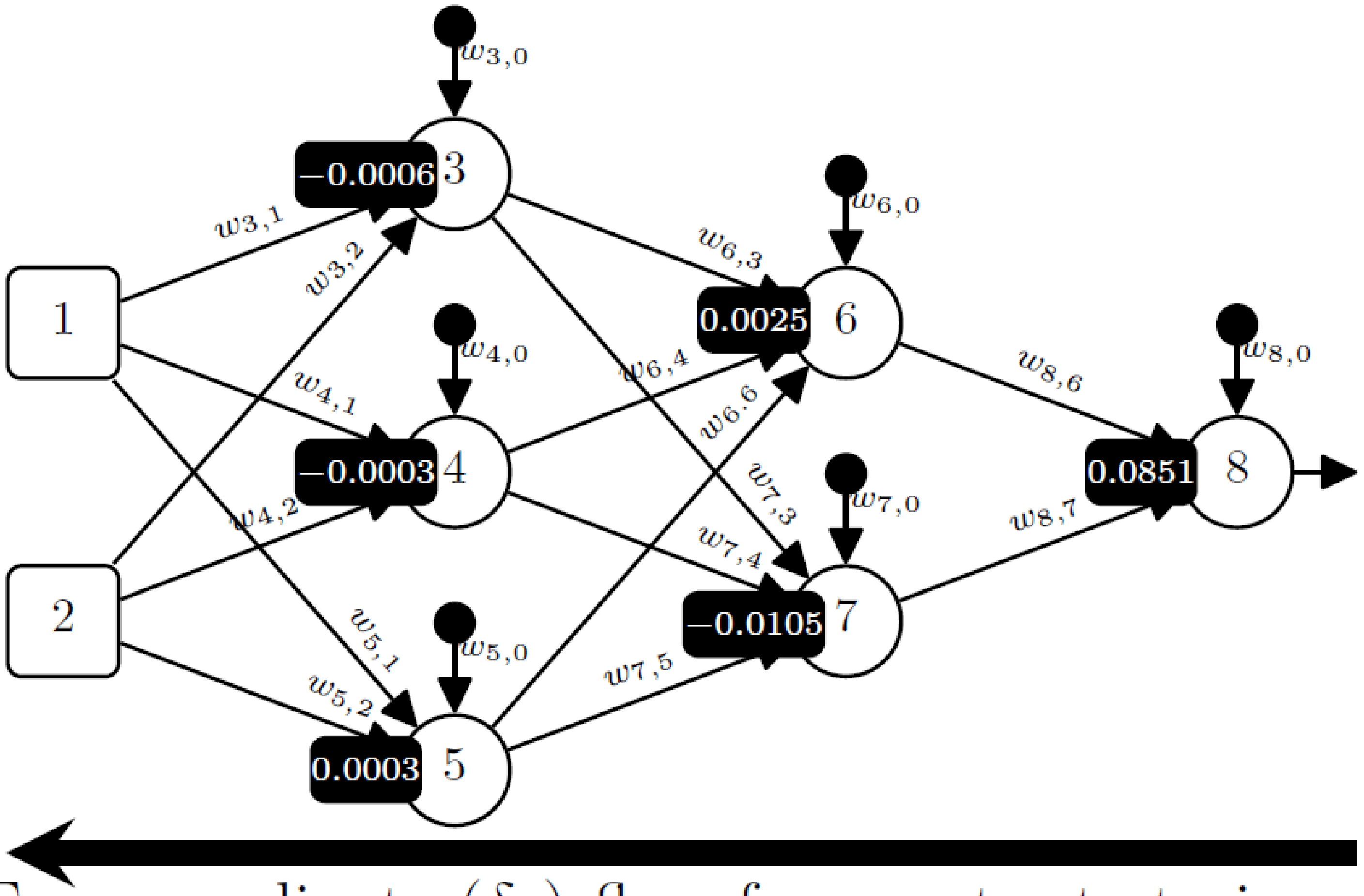
$$
\begin{aligned}
\delta_3 &= \frac{\partial \mathcal{E}}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \\
&= \left(\sum \delta_i \times w_{i,3}\right) \times \frac{\partial a_3}{\partial z_3} \\
&= ((\delta_6 \times w_{6,3}) + (\delta_7 \times w_{7,3})) \times \frac{\partial a_3}{\partial z_3} \\
&= ((0.0025 \times -0.04) + (-0.0105 \times 0.20)) \times 0.2500 \\
&= -0.0006
\end{aligned}
$$

$$
\begin{aligned}
\delta_4 &= \frac{\partial \mathcal{E}}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \\
&= \left(\sum \delta_i \times w_{i,4}\right) \times \frac{\partial a_4}{\partial z_4} \\
&= ((\delta_6 \times w_{6,4}) + (\delta_7 \times w_{7,4})) \times \frac{\partial a_4}{\partial z_4} \\
&= ((0.0025 \times 0.10) + (-0.0105 \times 0.14)) \times 0.2478 \\
&= -0.0003
\end{aligned}
$$

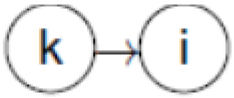
$$
\begin{aligned}
\delta_5 &= \frac{\partial \mathcal{E}}{\partial a_5} \times \frac{\partial a_5}{\partial z_5} \\
&= \left(\sum \delta_i \times w_{i,5}\right) \times \frac{\partial a_5}{\partial z_5} \\
&= ((\delta_6 \times w_{6,5}) + (\delta_7 \times w_{7,5})) \times \frac{\partial a_5}{\partial z_5} \\
&= ((0.0025 \times 0.06) + (-0.0105 \times -0.09)) \times 0.2491 \\
&= 0.0003
\end{aligned}
$$

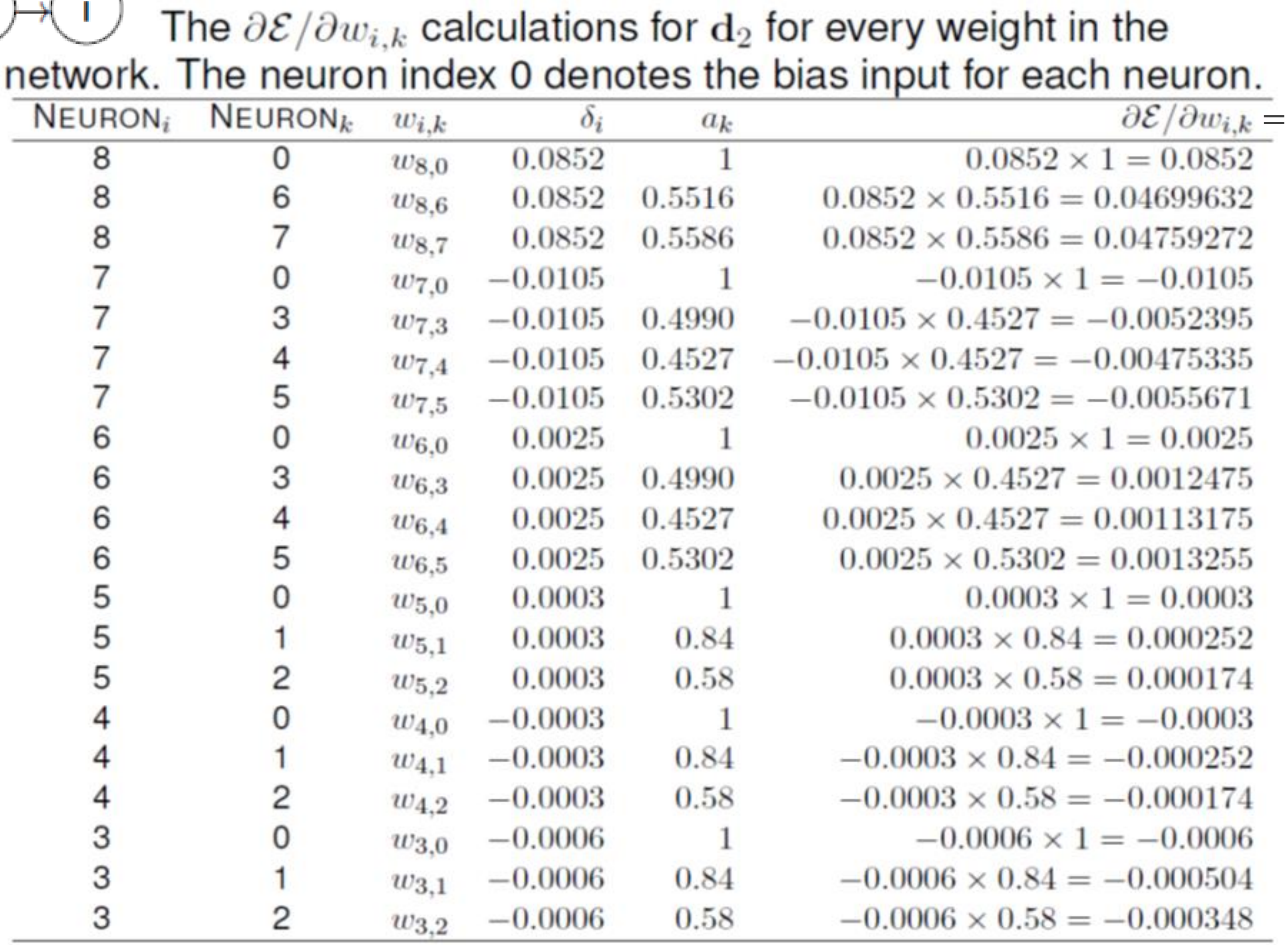
The Backward Pass of $\mathbf{d}_2$

$\mathbf{d}_2$
0.84
0.58



Error gradients (δs) flow from outputs to inputs





The $\partial\mathcal{E}/\partial w_{i,k}$ calculations for $\mathbf{d}_2$ for every weight in the network. The neuron index 0 denotes the bias input for each neuron.

NEURON$_i$	NEURON$_k$	$w_{i,k}$	δ_i	a_k	$\partial\mathcal{E}/\partial w_{i,k} = \boldsymbol{\delta_i \times a_k}$
8	0	$w_{8,0}$	0.0852	1	$0.0852 \times 1 = 0.0852$
8	6	$w_{8,6}$	0.0852	0.5516	$0.0852 \times 0.5516 = 0.04699632$
8	7	$w_{8,7}$	0.0852	0.5586	$0.0852 \times 0.5586 = 0.04759272$
7	0	$w_{7,0}$	-0.0105	1	$-0.0105 \times 1 = -0.0105$
7	3	$w_{7,3}$	-0.0105	0.4990	$-0.0105 \times 0.4527 = -0.0052395$
7	4	$w_{7,4}$	-0.0105	0.4527	$-0.0105 \times 0.4527 = -0.00475335$
7	5	$w_{7,5}$	-0.0105	0.5302	$-0.0105 \times 0.5302 = -0.0055671$
6	0	$w_{6,0}$	0.0025	1	$0.0025 \times 1 = 0.0025$
6	3	$w_{6,3}$	0.0025	0.4990	$0.0025 \times 0.4527 = 0.0012475$
6	4	$w_{6,4}$	0.0025	0.4527	$0.0025 \times 0.4527 = 0.00113175$
6	5	$w_{6,5}$	0.0025	0.5302	$0.0025 \times 0.5302 = 0.0013255$
5	0	$w_{5,0}$	0.0003	1	$0.0003 \times 1 = 0.0003$
5	1	$w_{5,1}$	0.0003	0.84	$0.0003 \times 0.84 = 0.000252$
5	2	$w_{5,2}$	0.0003	0.58	$0.0003 \times 0.58 = 0.000174$
4	0	$w_{4,0}$	-0.0003	1	$-0.0003 \times 1 = -0.0003$
4	1	$w_{4,1}$	-0.0003	0.84	$-0.0003 \times 0.84 = -0.000252$
4	2	$w_{4,2}$	-0.0003	0.58	$-0.0003 \times 0.58 = -0.000174$
3	0	$w_{3,0}$	-0.0006	1	$-0.0006 \times 1 = -0.0006$
3	1	$w_{3,1}$	-0.0006	0.84	$-0.0006 \times 0.84 = -0.000504$
3	2	$w_{3,2}$	-0.0006	0.58	$-0.0006 \times 0.58 = -0.000348$

* 0 indicates the bias term.

Weight Update

- Weight update for an example $\mathbf{d}_2$ (Suppose α =0.2)

$$
\begin{aligned}
w_{7,5} = & \; w_{7,5} - \alpha \times \boldsymbol{\delta}_7 \times a_5 \\
= & \; w_{7,5} - \alpha \times \frac{\partial \mathcal{E}}{\partial w_{i,k}} \\
= & -0.09 - 0.2 \times -0.0055671 \\
= & -0.08888658
\end{aligned}
$$

$$\begin{tikzcd} k \arrow[r] & i \end{tikzcd}$$

$$
\begin{aligned}
\boldsymbol{w_{i,k}} &= \boldsymbol{w_{i,k}} - \boldsymbol{\alpha} \times \boldsymbol{\delta_i} \times \boldsymbol{a_k} \\
&= w_{i,k} - \alpha \times \frac{\partial \mathcal{E}}{\partial w_{i,k}}
\end{aligned}
$$

- Weight update for four examples in the dataset

MINI-BATCH EXAMPLE	$\frac{\partial \mathcal{E}}{\partial w_{7,5}}$
$\mathbf{d}_1$	0.00730080
$\mathbf{d}_2$	−0.00556710
$\mathbf{d}_3$	0.00157020
$\mathbf{d}_4$	−0.00176664
$\Delta w_{7,5}$=	0.00153726

$$\Delta \boldsymbol{w_{i,k}} = \sum_{j=1}^{m} \boldsymbol{\delta_{i,j}} \times \boldsymbol{a_{k,j}} = \sum_{j=1}^{m} \frac{\partial \mathcal{E}}{\partial w_{i,k.j}}$$

, where j is an example, i and k are neurons

$$\boldsymbol{w_{i,k}} \leftarrow \boldsymbol{w_{i,k}} - \boldsymbol{\alpha} \times \Delta \boldsymbol{w_{i,k}}$$

$$
\begin{aligned}
w_{7,5} &= w_{7,5} - \alpha \times \Delta w_{i,k} \\
&= -0.09 - 0.2 \times 0.00153726 \\
&= -0.0903074520
\end{aligned}
$$

Network Error after Weight Update

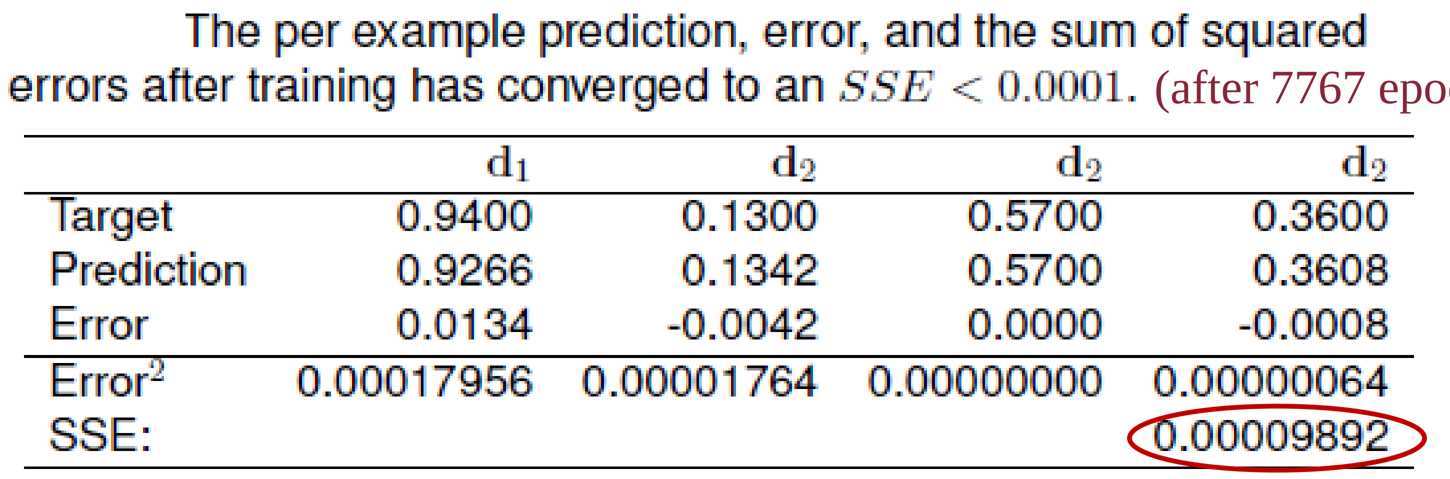
- The per example error after the forward pass, the per example $\partial\mathcal{E}/\partial a_8$, and the **sum of squared errors** for the model over the dataset of four examples.

	$\mathbf{d}_1$	$\mathbf{d}_2$	$\mathbf{d}_3$	$\mathbf{d}_4$
Target	0.9400	0.1300	0.5700	0.3600
Prediction	0.4715	0.4718	0.4717	0.4716
Error	0.4685	-0.3418	0.0983	-0.1116
$\partial\mathcal{E}/\partial a_8$: Error $\times -1$	-0.4685	0.3418	-0.0983	0.1116
Error^2	0.21949225	0.11682724	0.00966289	0.01245456
SSE:				0.17921847

- The per example error after each weight has been updated once, the per example $\partial\mathcal{E}/\partial a_8$, and the **sum of squared errors** for the model.

	$\mathbf{d}_1$	$\mathbf{d}_2$	$\mathbf{d}_3$	$\mathbf{d}_4$
Target	0.9400	0.1300	0.5700	0.3600
Prediction	0.4738	0.4741	0.4740	0.4739
Error	0.4662	-0.3441	0.0960	-0.1139
$\partial\mathcal{E}/\partial a_8$: Error $\times -1$	-0.4662	0.3441	-0.0960	0.1139
Error^2	0.21734244	0.11840481	0.009216	0.01297321
SSE:				0.17896823

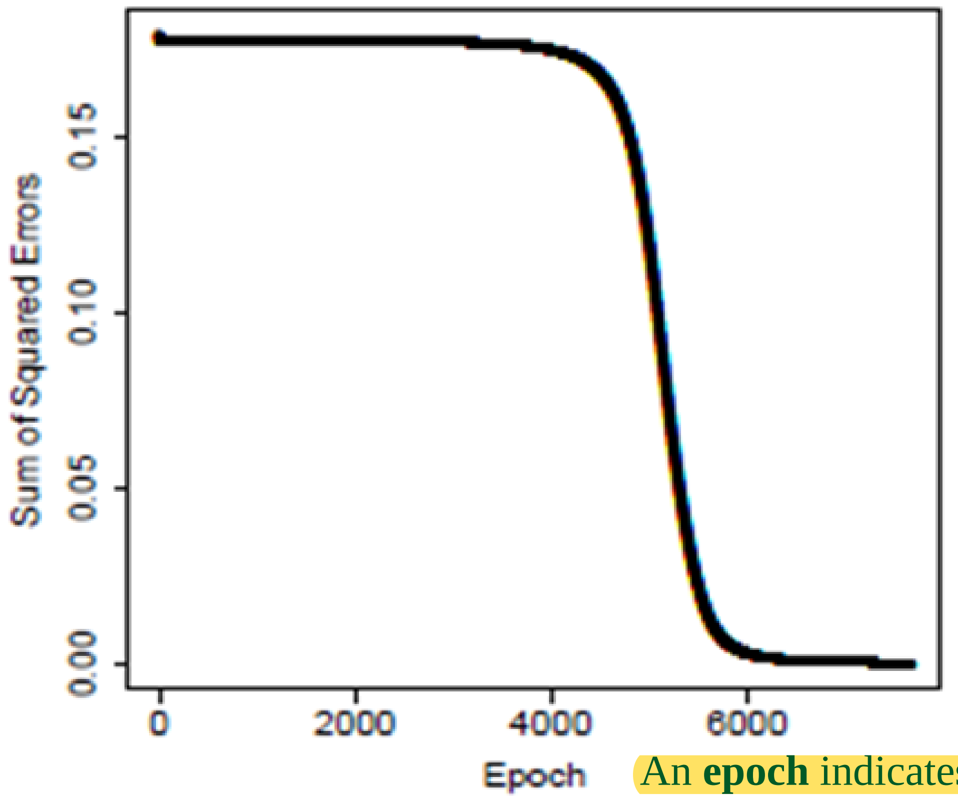
Network Error After Training has Converged



- The per example prediction, error, and the sum of squared errors after training has converged to an $SSE < 0.0001$. (after 7767 epochs)

	d_1	d_2	d_2	d_2
Target	0.9400	0.1300	0.5700	0.3600
Prediction	0.9266	0.1342	0.5700	0.3608
Error	0.0134	-0.0042	0.0000	-0.0008
Error^2	0.00017956	0.00001764	0.00000000	0.00000064
SSE:				0.00009892

Change of Network Errors



An **epoch** indicates a single pass through all the examples in the training dataset.

Figure: A plot showing how the sum of squared errors of the network changed during training