Deep Learning

Prof. Ahmed Guessoum and Dr. Mohamed Brahimi

Big Idea

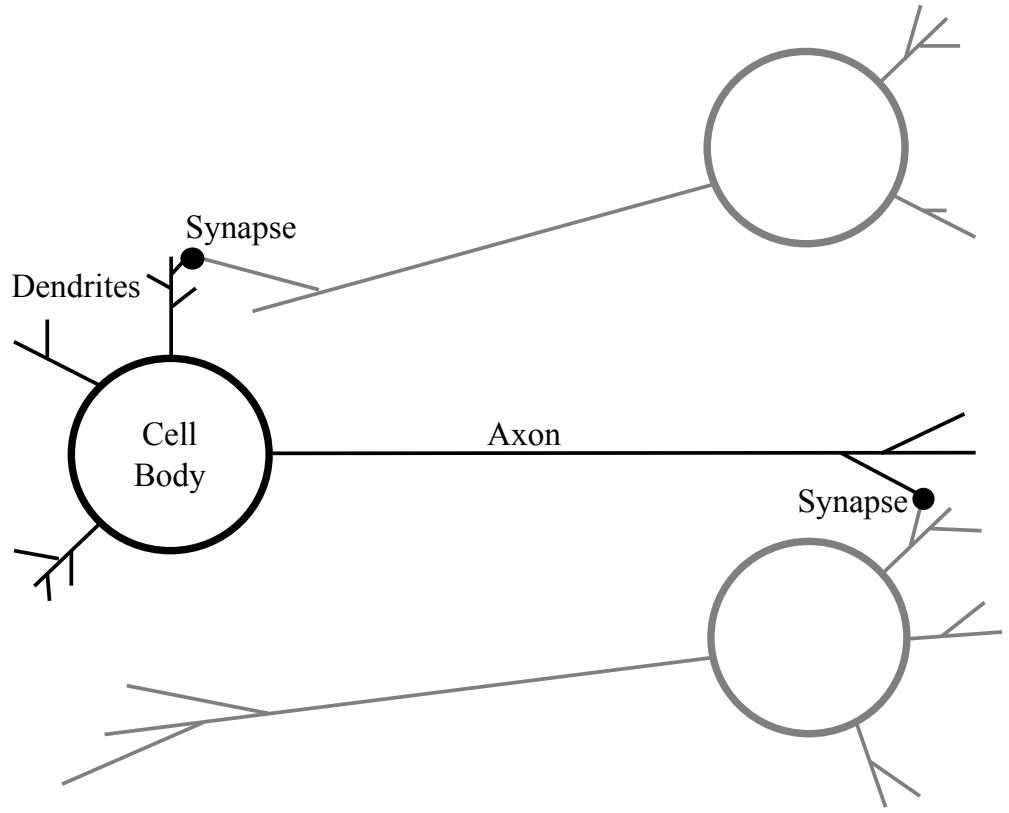


Figure 1: A high-level schematic of the structure of a neuron. This figure illustrates three interconnected neurons; the middle neuron is highlighted in black, and the major structural components of this neuron are labeled **cell body**, **dendrites**, and **axon**. Also marked are the **synapses** connecting the axon of one neuron and the dendrite of another, which allow **signals** to pass between the neurons.

Fundamentals

- The fundamental building block of a neural network is a computational model known as an artificial neuron.
- First defined by McCulloch and Pitts (1943) who were trying to develop a model of the activity in the human brain based on propositional logic.
- Their inspiration for this work was linking neurons in the brain to propositional logic using a Boolean representation.
- Neurons are somewhat similar, insofar as they act as a switch that responds to a set of inputs by outputting either a high activation (1) or no activation (0).
- So they designed a model of the neuron that would take in multiple inputs and then output either a high signal (a 1) or a low signal (a 0).

The McCulloch and Pitts model works in two phases:

1. A **weighted sum** calculation: each input is multiplied by a weight, and the results of these multiplications are then added together yielding z.
2. If z is greater than or equal to a manually preset **threshold**, the artificial neuron outputs a 1 (high activation); otherwise it outputs a 0 (low activation).

The McCulloch and Pitts model works in two phases:

1. A **weighted sum** calculation: each input is multiplied by a weight, and the results of these multiplications are then added together yielding z.
2. If z is greater than or equal to a manually preset **threshold**, the artificial neuron outputs a 1 (high activation); otherwise it outputs a 0 (low activation).

The term **activation function** is used as a general term to refer to whatever function is employed in the second stage of an artificial neuron, because the function maps the weighted sum value, z, into the output value, or activation, of the neuron.

$$z = \underbrace{\mathbf{w}[0] \times \mathbf{d}[0] + \mathbf{w}[1] \times \mathbf{d}[1] + \cdots + \mathbf{w}[m] \times \mathbf{d}[m]}_{weighted\ sum} \tag{1}$$

$$= \sum_{j=0}^{m} \mathbf{w}[j] \times \mathbf{d}[j]$$

$$= \underbrace{\mathbf{w} \cdot \mathbf{d}}_{dot\ product} = \underbrace{\mathbf{w}^T \mathbf{d}}_{matrix\ product} = [w_0, w_1, \ldots, w_m] \begin{bmatrix} d_0 \\ d_2 \\ \vdots \\ d_m \end{bmatrix} \tag{2}$$

where d is a vector of $m+1$ descriptive features, $d[0], \ldots d[m]$ and $w[0] \ldots w[m]$ are $m+1$ weights.

$$z = \underbrace{\mathbf{w}[0] \times \mathbf{d}[0] + \mathbf{w}[1] \times \mathbf{d}[1] + \cdots + \mathbf{w}[m] \times \mathbf{d}[m]}_{weighted\ sum} \tag{1}$$

$$= \sum_{j=0}^{m} \mathbf{w}[j] \times \mathbf{d}[j]$$

$$= \underbrace{\mathbf{w} \cdot \mathbf{d}}_{dot\ product} = \underbrace{\mathbf{w}^T \mathbf{d}}_{matrix\ product} = [w_0, w_1, \ldots, w_m] \begin{bmatrix} d_0 \\ d_2 \\ \vdots \\ d_m \end{bmatrix} \tag{2}$$

where d is a vector of $m+1$ descriptive features, $d[0], \ldots d[m]$ and $w[0] \ldots w[m]$ are $m+1$ weights.

N.B.: $w[0]$, the **bias parameter**: when no other input, the output of the weighted sum is biased to be $w[0]$.

As such, the model can define lines that do not go through the origin of the input space.

The McCulloch and Pitts model used the following **threshold activation function**.

$$\mathbb{M}_{\mathbf{w}}(\mathbf{d}) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & otherwise \end{cases} \tag{3}$$

The McCulloch and Pitts model used the following **threshold activation function**.

$$\mathbb{M}_{\mathbf{w}}(\mathbf{d}) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & otherwise \end{cases} \tag{3}$$

Until recently, one of the most popular functions used in artificial neurons was the **logistic function** introduced in Chapter 7 (see the figure on the next slide).

The McCulloch and Pitts model used the following **threshold activation function**.

$$\mathbb{M}_{\mathbf{w}}(\mathbf{d}) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & otherwise \end{cases} \tag{3}$$

Until recently, one of the most popular functions used in artificial neurons was the **logistic function** introduced in Chapter 7 (see the figure on the next slide).

Today, the most popular choice of function for an activation function is the **rectified linear activation function** or **rectifier**, defined as follows:

$$rectifier(z) = max(0, z) \tag{4}$$

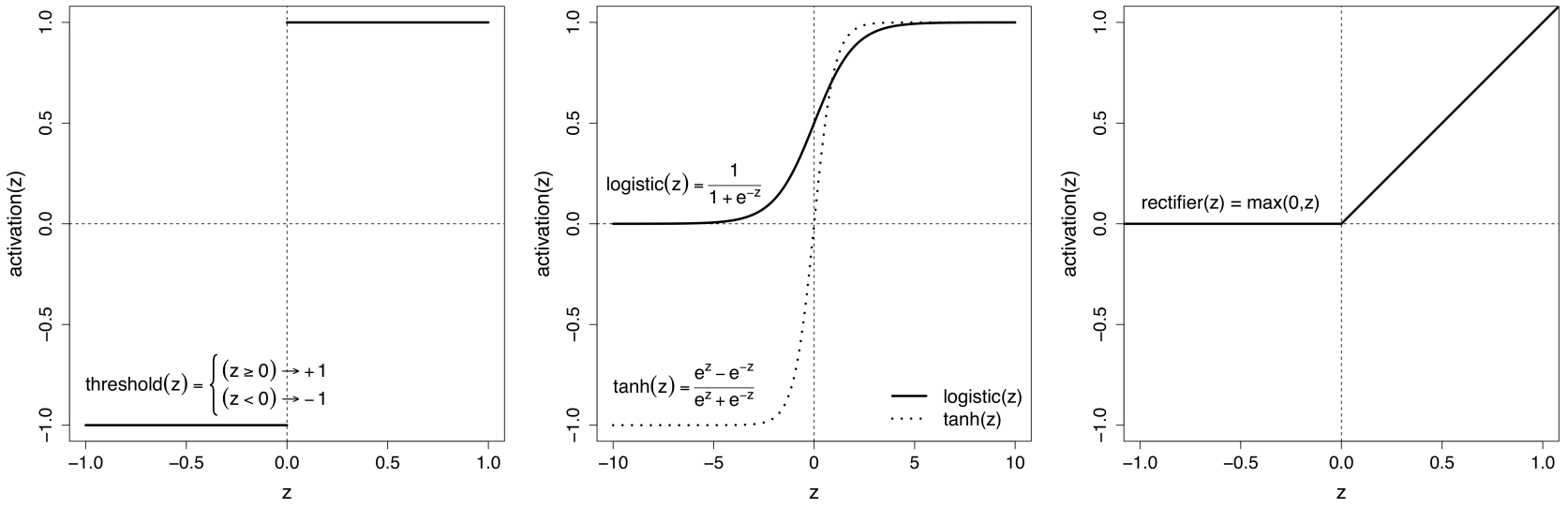


Figure 2: Plots for activation functions that have been popular in the history of neural networks.

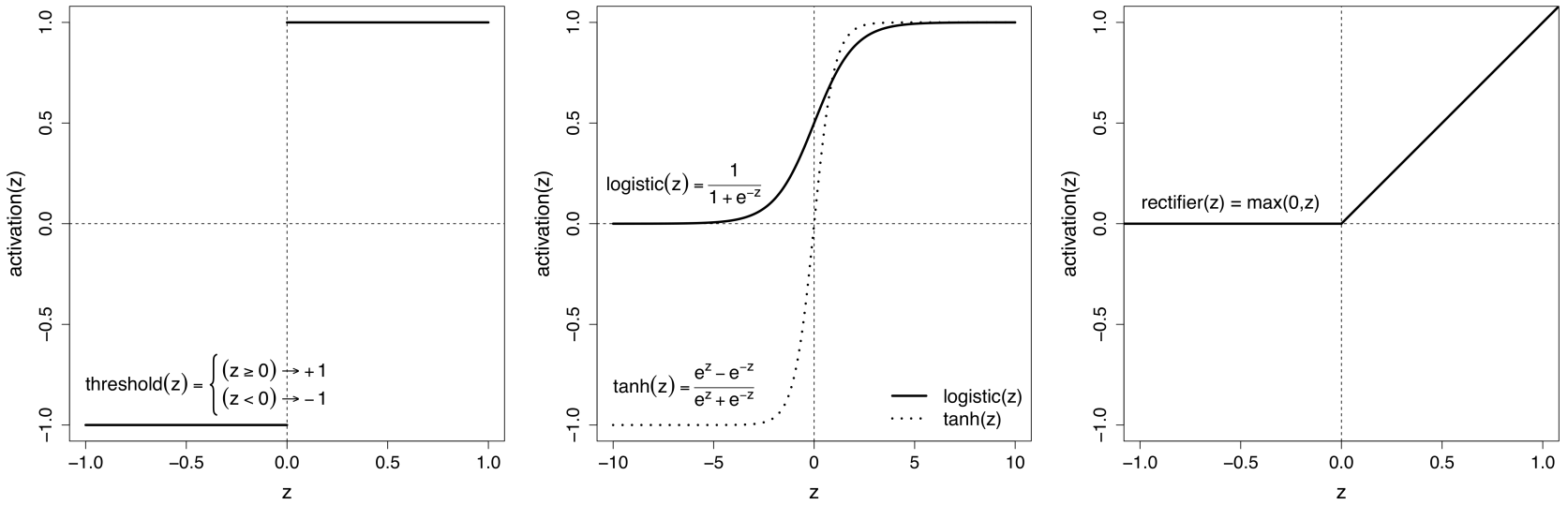


Figure 2: Plots for activation functions that have been popular in the history of neural networks.

- Clearly, a common characteristic of all of these activation functions is that they are not linear functions.
- It is the introduction of a non-linearity into the input to output mapping defined by a neuron that enables an artificial neural network (ANN) to learn complex non-linear mappings, thus making them powerful models.

If we use the symbol φ to represent the activation function of a neuron, we can mathematically define an artificial neuron as follows:

$$\begin{aligned}\mathbb{M}_{\mathbf{w}}(\mathbf{d}) &= \varphi\left(\mathbf{w}\left[0\right] \times \mathbf{d}\left[0\right] + \mathbf{w}\left[1\right] \times \mathbf{d}\left[1\right] + \cdots + \mathbf{w}\left[m\right] \times \mathbf{d}\left[m\right]\right) \\ &= \varphi\left(\sum_{i=0}^{m} w_i \times d_i\right) = \varphi\left(\underbrace{\mathbf{w} \cdot \mathbf{d}}_{dot\ product}\right) \\ &= \varphi\left(\underbrace{\mathbf{w}^T\mathbf{d}}_{matrix\ product}\right) = \varphi\left([w_0, w_1, \ldots, w_m]\begin{bmatrix} d_0 \\ d_2 \\ \vdots \\ d_m \end{bmatrix}\right)\end{aligned} \tag{5}$$

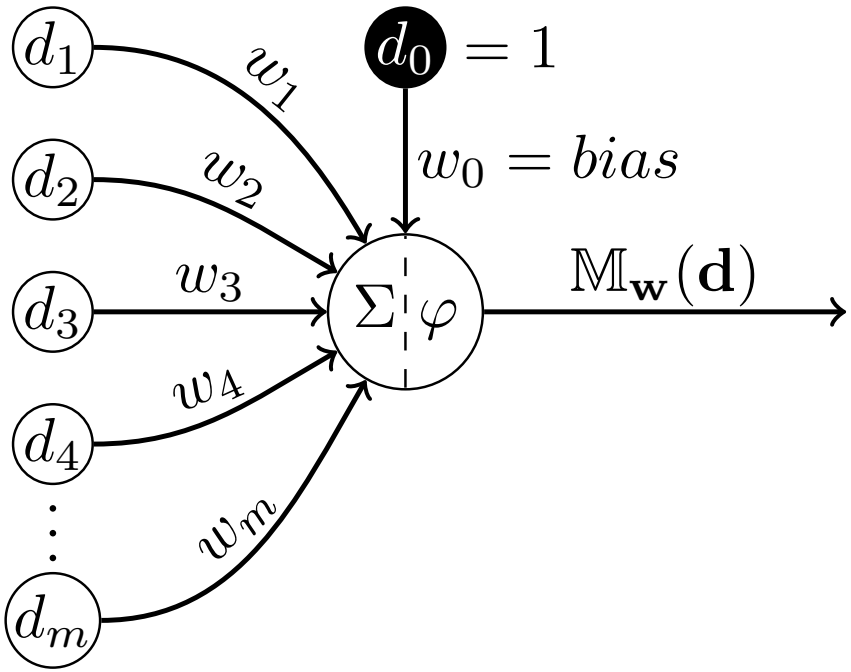


Figure 3: A schematic of an artificial neuron.

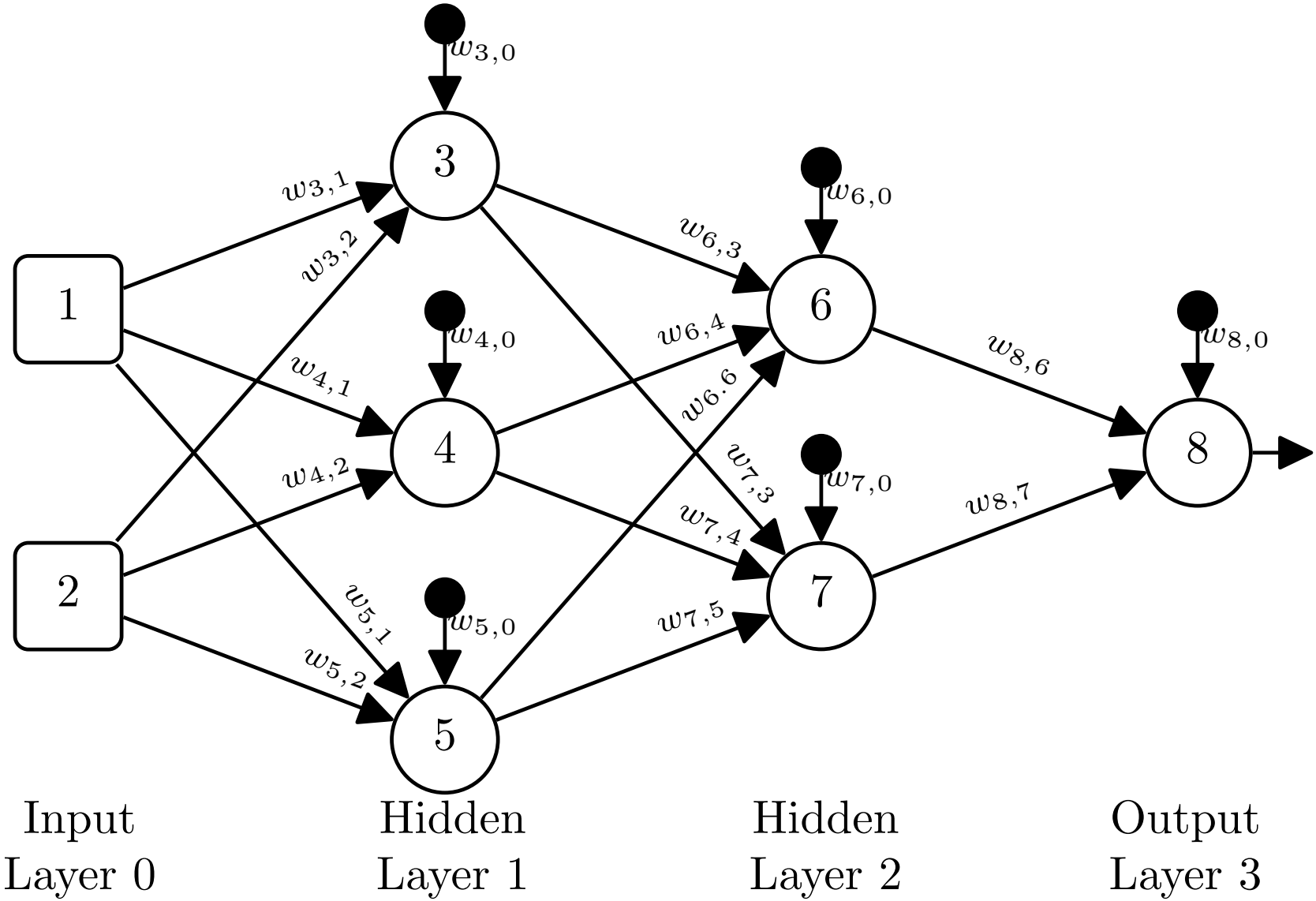


Figure 4: A schematic of a feedforward artificial neural network.

- An ANN consists of a network of interconnected artificial neurons organized into a sequence of layers.
- An ANN can have any structure, but an organization into layers of neurons is common.
- The previous network architecture is an example of a **feedforward network**.
  - A feedforward network contains no loops (cycles) in the network connections that would allow the output of a neuron to flow back as input into the neuron (even indirectly).
  - So in a feedforward network the activations always flow forward through the sequence of layers.
  - This network is also a **fully connected network** because each of its neurons is connected such that all the neurons of any layer are connected to all the neurons of the next one.
  - We will later see network architectures that are not feedforward and architectures that are not fully connected.

- The **depth** of a neural network is equal to the number of hidden layers plus the output layer.
- Cybenko (1988) proved that a network with three layers of (processing) neurons (i.e., two hidden layers and an output layer) can approximate any function to arbitrary accuracy.
- So we will consider that the minimum number of hidden layers for a Deep network is 2.
- Today, some deep ANNs may have tens or even hundreds of hidden layers.

We use the convention of a bold capital letter to denote a matrix and a superscript in parentheses to list the relevant layer.
Using this notation, Let us consider the second layer of neurons in a network as an example.
Let us denote:

- the matrix containing the weights on the edges into Layer 2 as $\boldsymbol{W}^{(2)}$,
- the column vector of activations coming from the neurons in Layer 1 as $\boldsymbol{a}^{(1)}$, and
- the column vector of weighted sums for the neurons in Layer 2 as $\boldsymbol{z}^{(2)}$,

then the order of the matrices in the multiplication operation that we use in this explanation is

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} \tag{6}$$

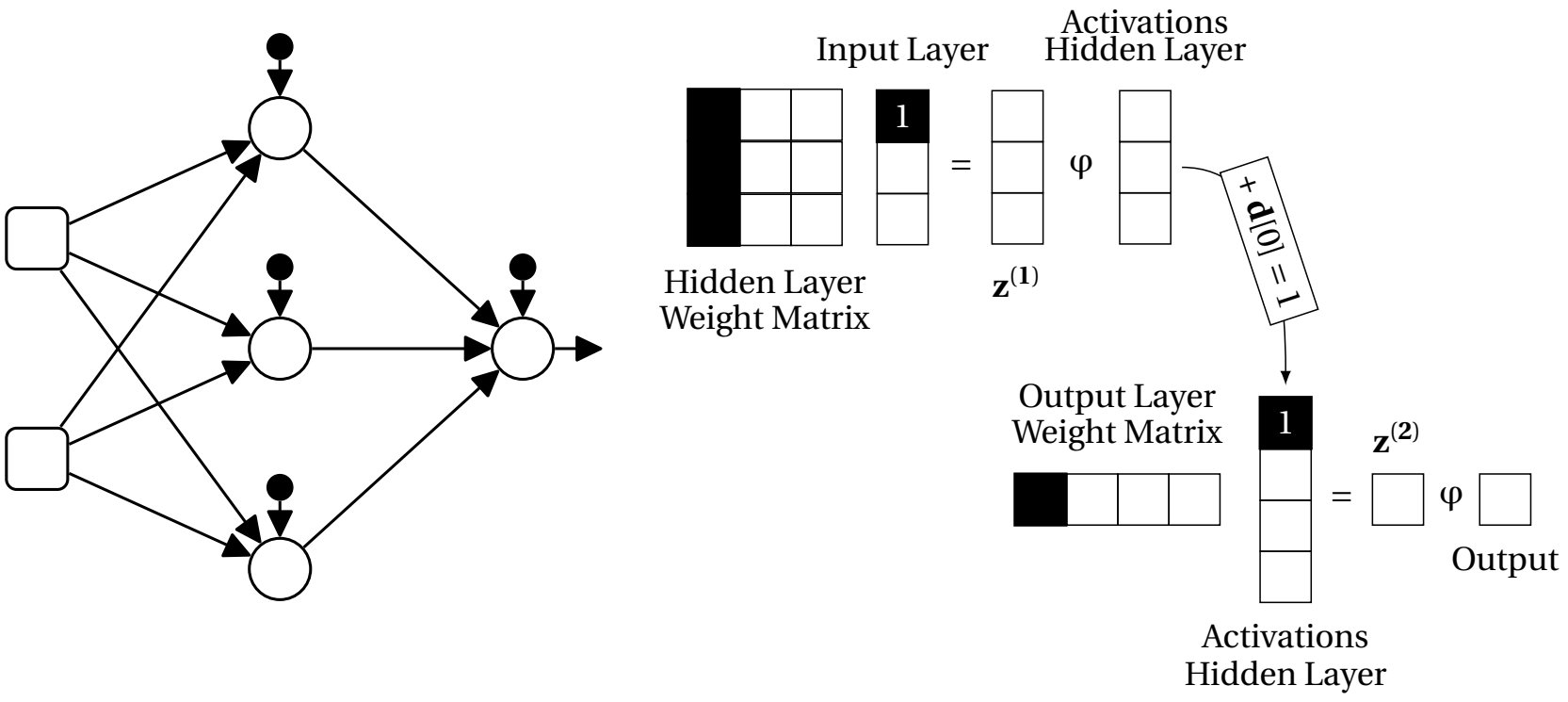


Figure 5: An illustration of the correspondence between graphical and matrix representations of a neural network.

The fact that we can use a sequence of matrix operations to implement how a neural network processes a single example can be generalized to processing a number of examples in parallel.

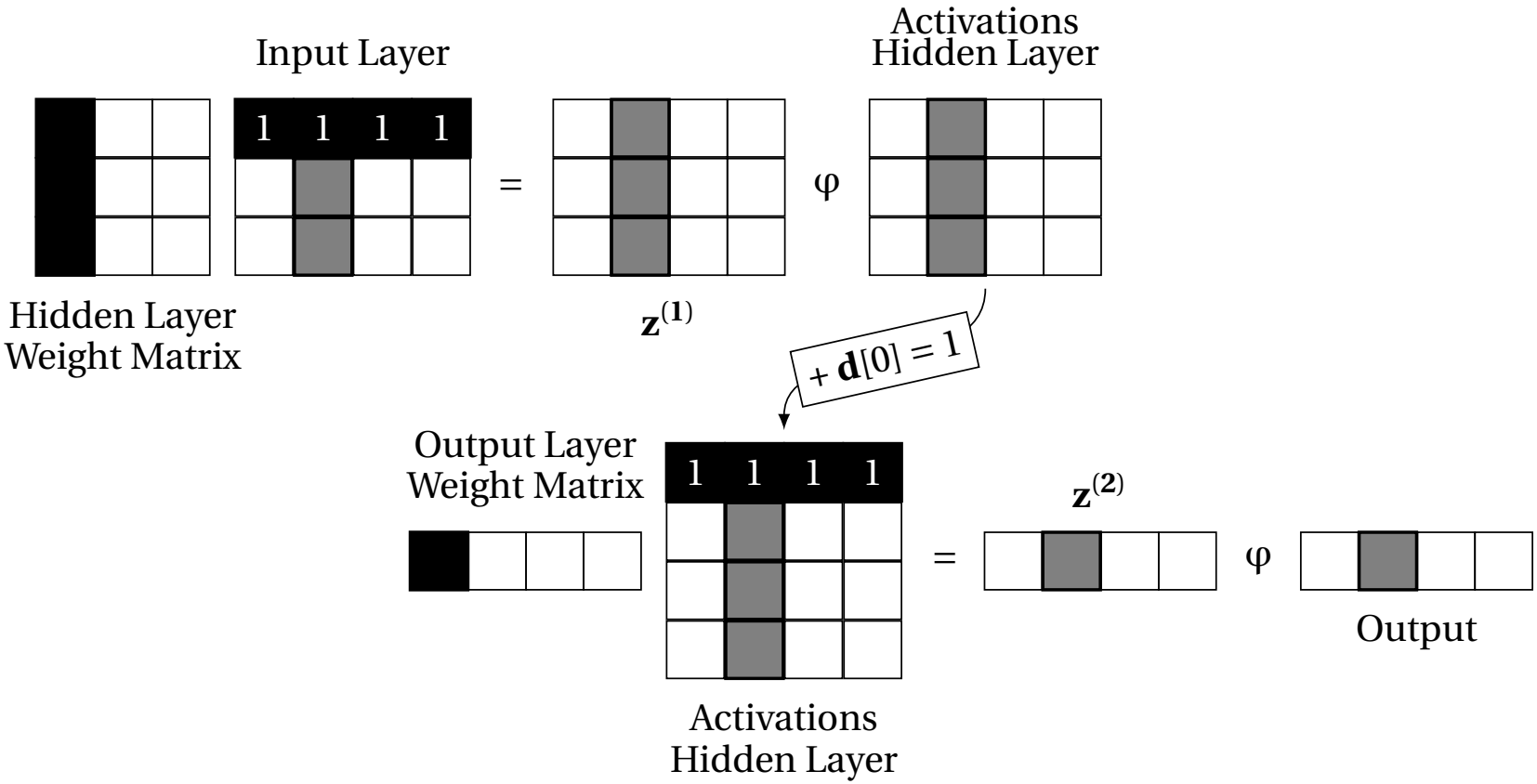


Figure 6: An illustration of how a batch of examples can be processed in parallel using matrix operations

- Implementing a network as a sequence of matrix multiplications:
  1. speeds up the calculation of a weighted sum across each layer in the network, and
  2. enables the network to parallelize the processing of examples.
- Both these speedups enable us to remove expensive for loops from the implementation and training of a network.
- A further benefit of implementing an ANN using matrix operations is that it enables the use of specialized hardware known as graphical processing units (GPUs) which are designed to carry out matrix operations very quickly.

A multi-layer feedforward ANN that uses only linear neurons (i.e., neurons that do not include a non-linear activation function) is equivalent to a single-layer network with linear neurons. In other words, it can represent only a linear mapping on the inputs.

$$\mathbf{A}^{(1)} = \mathbf{W}^{(1)}\mathbf{A}^{(0)} \tag{7}$$

$$\mathbf{A}^{(2)} = \mathbf{W}^{(2)}\mathbf{A}^{(1)} \tag{8}$$

$$\mathbf{A}^{(2)} = \mathbf{W}^{(2)}\left(\mathbf{W}^{(1)}\mathbf{A}^{(0)}\right) \tag{9}$$

$$\mathbf{A}^{(2)} = \left(\mathbf{W}^{(2)}\mathbf{W}^{(1)}\right)\mathbf{A}^{(0)} \tag{10}$$

$$\mathbf{A}^{(2)} = \mathbf{W}'\mathbf{A}^{(0)} \tag{11}$$

- The product of weight matrices of the linear layers, will also implement a linear transformation on the input data.
- Therefore, this rewriting shows that the output of this two-layer network with linear neurons is equivalent to a single-layer network with linear neurons.
- This analysis shows that adding layers to a network without including a non-linear activation function between the layers appears to add complexity to the network, but in reality the network remains equivalent to a single-layer linear network.
- Fortunately, we do not need to add complex non-linearities between the layers; introducing simple non-linearities (e.g. the logistic or rectifier functions) is sufficient to enable neural networks to represent arbitrarily complex functions, as long as the network is deep enough.

- What types of functions can a network that has only a single layer of processing neurons capable of representing?

- What types of functions can a network that has only a single layer of processing neurons capable of representing?
- A thresholded unit (also known as perceptron) is identical to a multivariate linear regression model with a threshold applied to it.
- So, similar to the thresholded multivariate linear regression models, a perceptron is able to represent a function that distinguishes between two classes of inputs if these two classes are linearly separable.

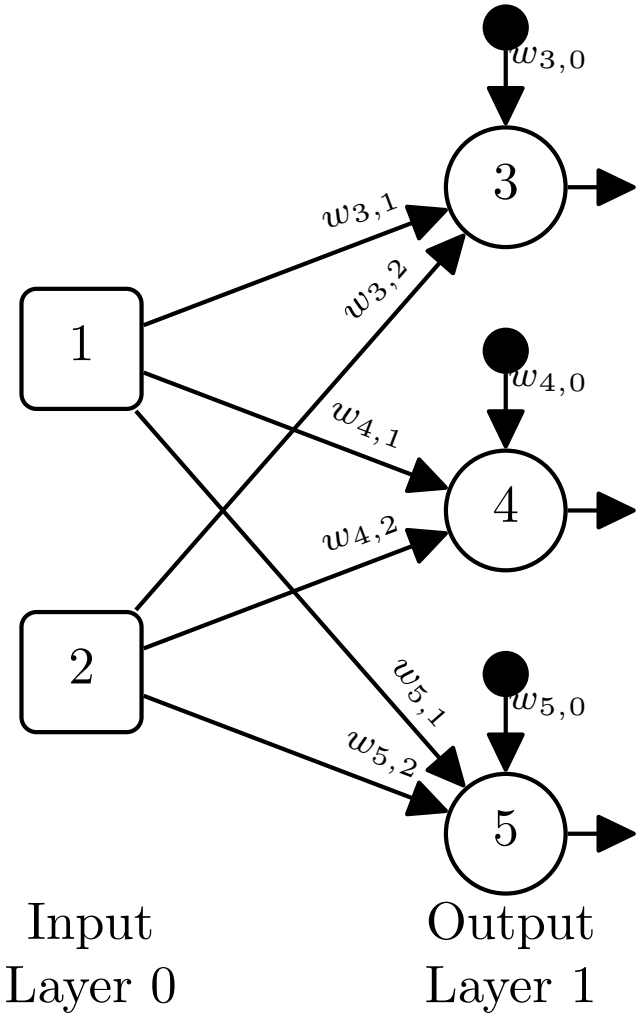


Figure 7: A single-layer network.

- Each of the neurons in the output layer (Neurons 3, 4, and 5) are independent of each other; they receive no information from each other.
- In fact, the network is equivalent to three separate neurons, each receiving the same input vector. (We will assume that each of the 3 neurons uses a threshold activation function. So each of them is equivalent to the McCulloch and Pitts neuron.)

- Each of the neurons in the output layer (Neurons 3, 4, and 5) are independent of each other; they receive no information from each other.
- In fact, the network is equivalent to three separate neurons, each receiving the same input vector. (We will assume that each of the 3 neurons uses a threshold activation function. So each of them is equivalent to the McCulloch and Pitts neuron.)
- The fact that our single-layer network contains 3 independent neurons means that the network has the potential to represent three separate linear decision boundaries.
- However, none of the neurons in the output layer are capable of representing a non-linear decision boundary on the inputs, so the network as a whole cannot represent a non-linear function.

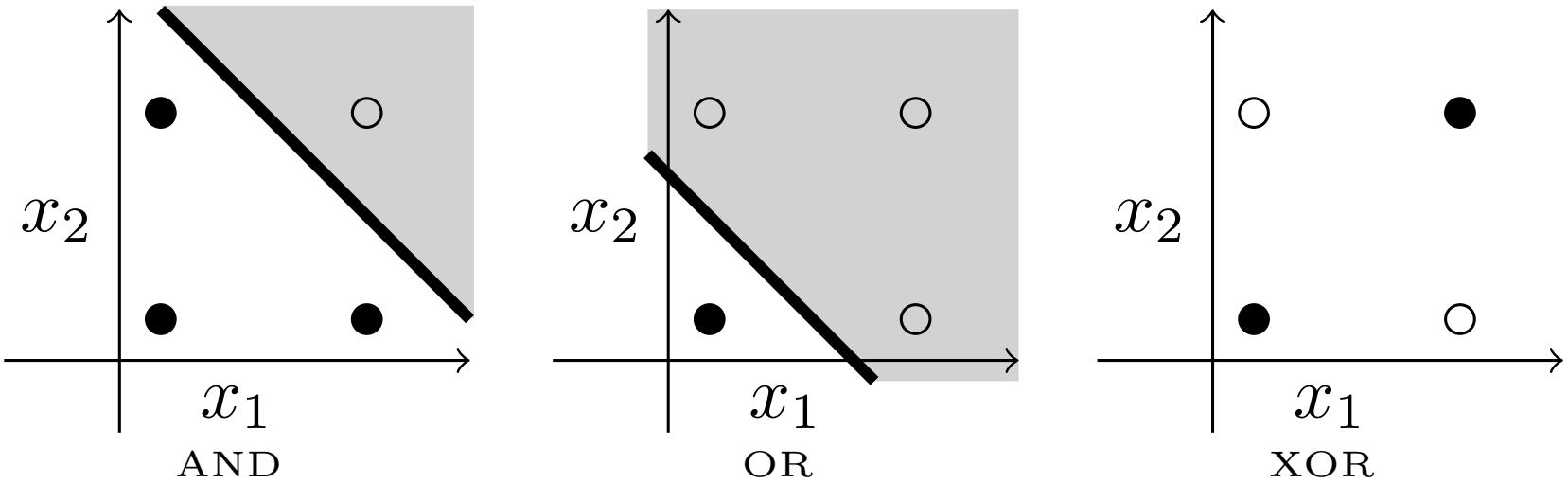


Figure 8: The logical AND and OR functions are linearly separable, but the XOR is not.

Although the XOR function is simple, a *perceptron* cannot represent it.

All the neurons in this network use the following threshold activation function:

$$\mathbb{M}_{\mathbf{w}}(\mathbf{d}) = \begin{cases} 1 & \text{if } z \geq 1 \\ 0 & otherwise \end{cases} \tag{12}$$

The representational limitation of the single-layer network can be overcome by adding a single hidden layer to the network.

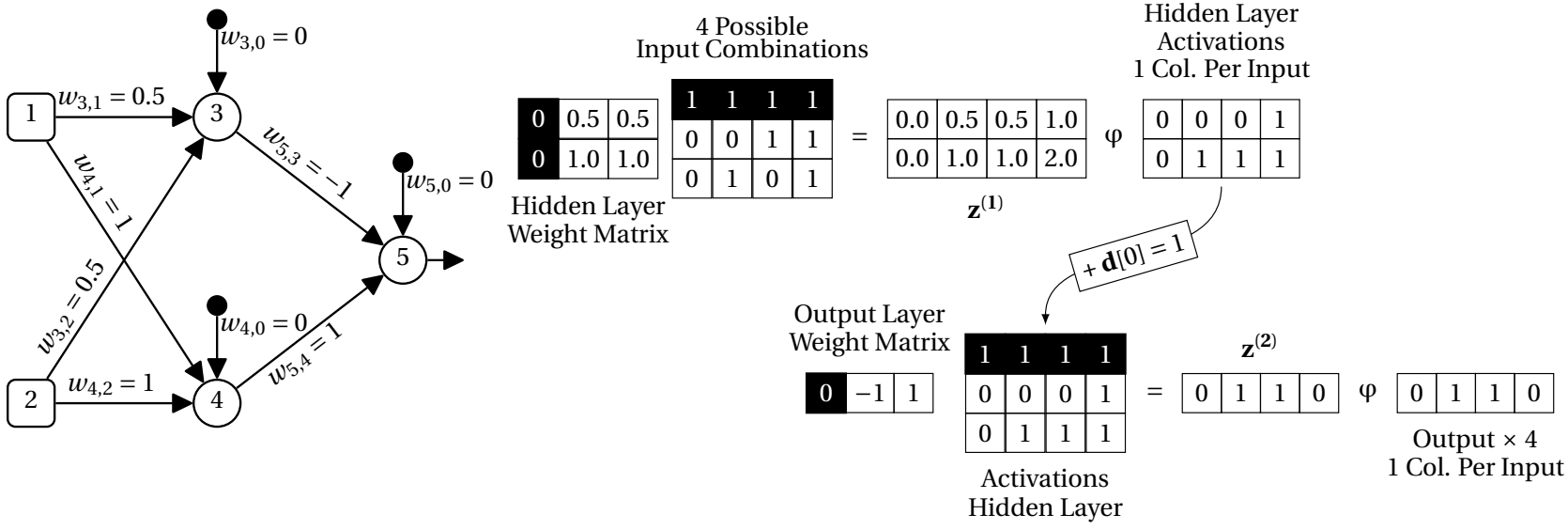


Figure 9: (left) The XOR function implemented as a two-layer neural network. (right) The network processing the four possible input combinations, one combination plus bias input per column: [*bias*, *FALSE*,*FALSE*] $\rightarrow$ [1,0,0]; [*bias*, *FALSE*,*TRUE*] $\rightarrow$ [1,0,1]; [*bias*, *TRUE*,*FALSE*] $\rightarrow$ [1,1,0]; [*bias*, *TRUE*,*TRUE*] $\rightarrow$ [1,1,1].

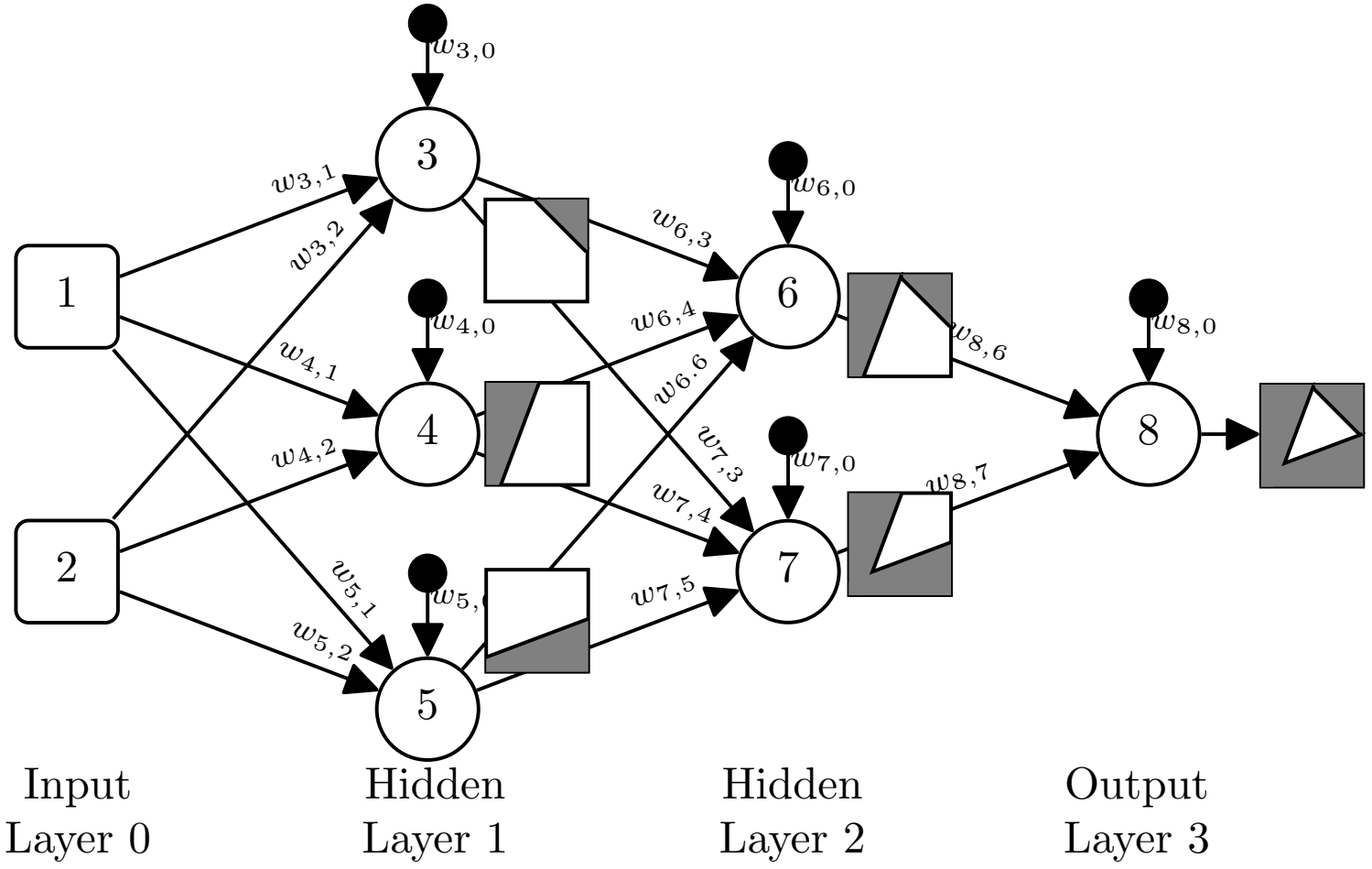


Figure 10: An illustration of how the representational capacity of a network increases as more layers are added to the network.

- Neural networks with no hidden layer (i.e., perceptrons) cannot represent non-linearly separable functions.
- Neural networks with a single hidden layer have been proven to be capable of universal approximation of (bounded) continuous functions as long as either
  1. complex functions are integrated into the structure of the neurons, or
  2. the hidden layer of the network is sufficiently (potentially exponentially) wide.
- Neural networks with two hidden layers and using smooth activation functions can represent any function and generally can do so using fewer neurons than networks with only a single hidden layer.

- As layers are added to a network, neurons in subsequent layers are able to use the representations learned by the preceding layer as building blocks to construct more complex functions.
- Overall there is a general trend that deeper networks have better performance than shallower networks, and that deeper networks are often more efficient in terms of the number of neurons they require.
- However, as networks become deeper they can become more difficult to train.

Standard Approach: Backpropagation and Gradient Descent

- A neuron is structurally equivalent to a logistic regression model.
- Indeed, when a neuron uses the logistic function as an activation function, then these two models are identical.
- As a consequence, if a neuron uses a logistic activation function, then we can train the neuron in the same way that we train a logistic regression function: using the gradient descent algorithm (introduced in Chapter 7) with the weight update rule.
- if the neuron implements a different activation function, then so long as the function is differentiable, we modify the weight update rule by replacing the derivative of the logistic function with the derivative of the new activation function, and apply the gradient descent algorithm.

- Training a neuron by initializing its weights and then iteratively updating these weights to reduce the error of the neuron on training examples fits with Hebb?s Postulate (See Section 8.1) that learning occurs in the brain through a process involving changes in the connections between neurons.
- Although in a neural network it is simple to calculate the gradient of the error for neurons in the output layer by directly comparing the activations of these neurons with the expected outputs, it is not possible to directly compare the activation of a **hidden** neuron with the expected activation for that neuron, and so we cannot directly calculate an error gradient for hidden neurons.

- We must calculate a measure of how the neuron contributed to the overall error of the network at the output layer (**blame assignment problem**).
- The **backpropagation algorithm** solves the blame assignment problem. Once we have used the backpropagation algorithm to solve the blame assignment problem for all the neurons in the network, we can then use the weight update rule from the gradient descent algorithm to update the weights for each of the neurons in the network.

- The backpropagation algorithm begins by initializing the weights of the network. We assume that the weights are initialized to random values close to zero.
- Then we can train the network to implement a useful function by iteratively presenting examples to it and using its error on the examples to update the weights so that it converges on a set of weights that implement a useful function relative to the patterns in the training data.
- The key step in this iterative weight update process is solving the blame assignment problem.
- The general structure of the backpropagation algorithm is a two-step process.

1. **Forward Pass:** An input pattern is presented to the network, and the activations flow forward through the network until an output is generated. The activations of all neurons are recorded during the forward pass. (11[44])
2. **Backward Pass:** The error of the network is calculated by comparing the output generated by the forward pass with the target output specified in the dataset. This error is **backpropagated** through the network on a layer-by-layer basis until the input layer is reached. During this pass, an error gradient δ_i for each neuron i is calculated. These error gradients can then be used by the gradient descent weight update rule to update the weights for each neuron. (12[45])

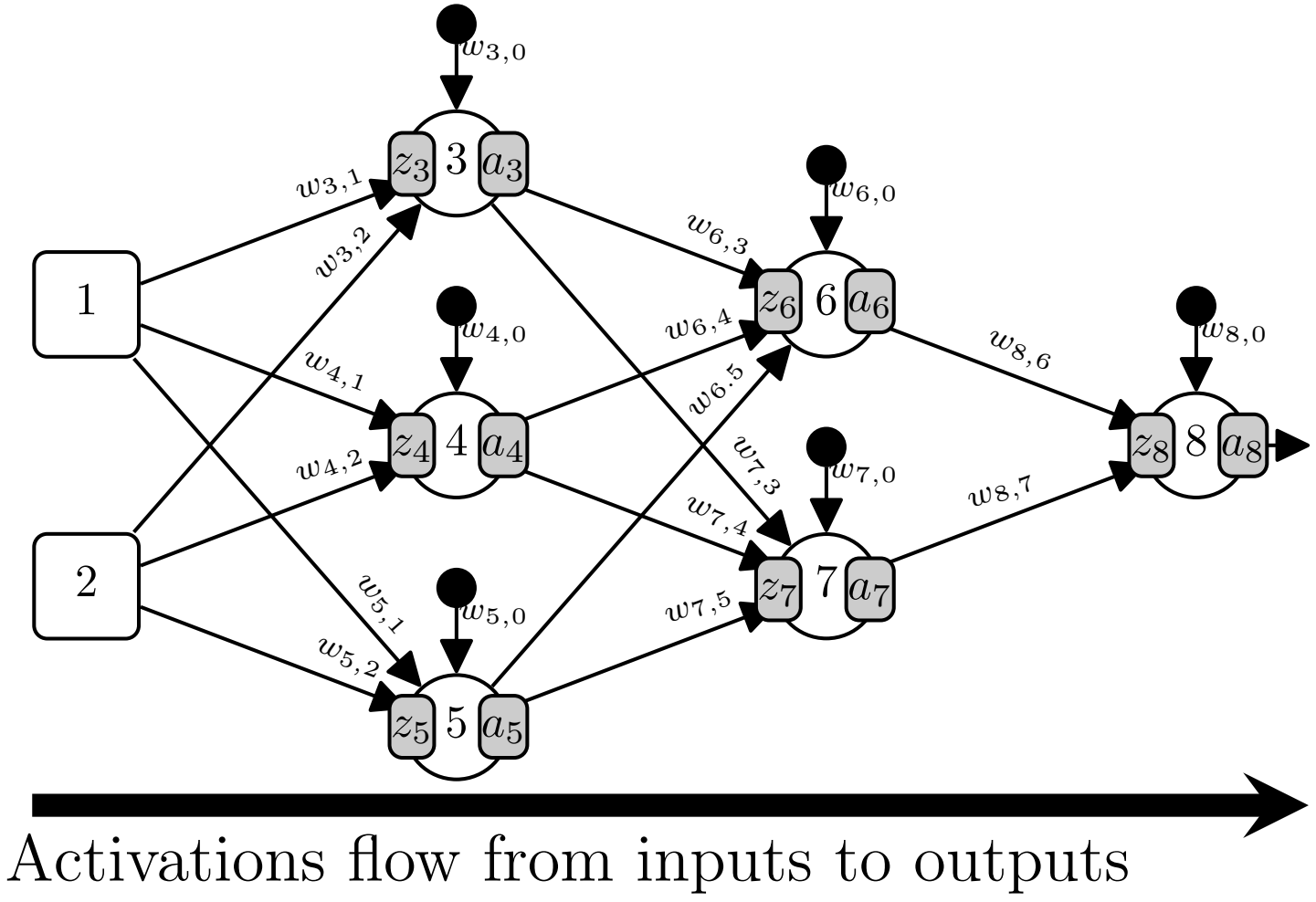


Figure 11: The calculation of the z values and activations of each neuron during the forward pass of the backpropagation algorithm. This figure is based on Figure 6.5 of (**?**).

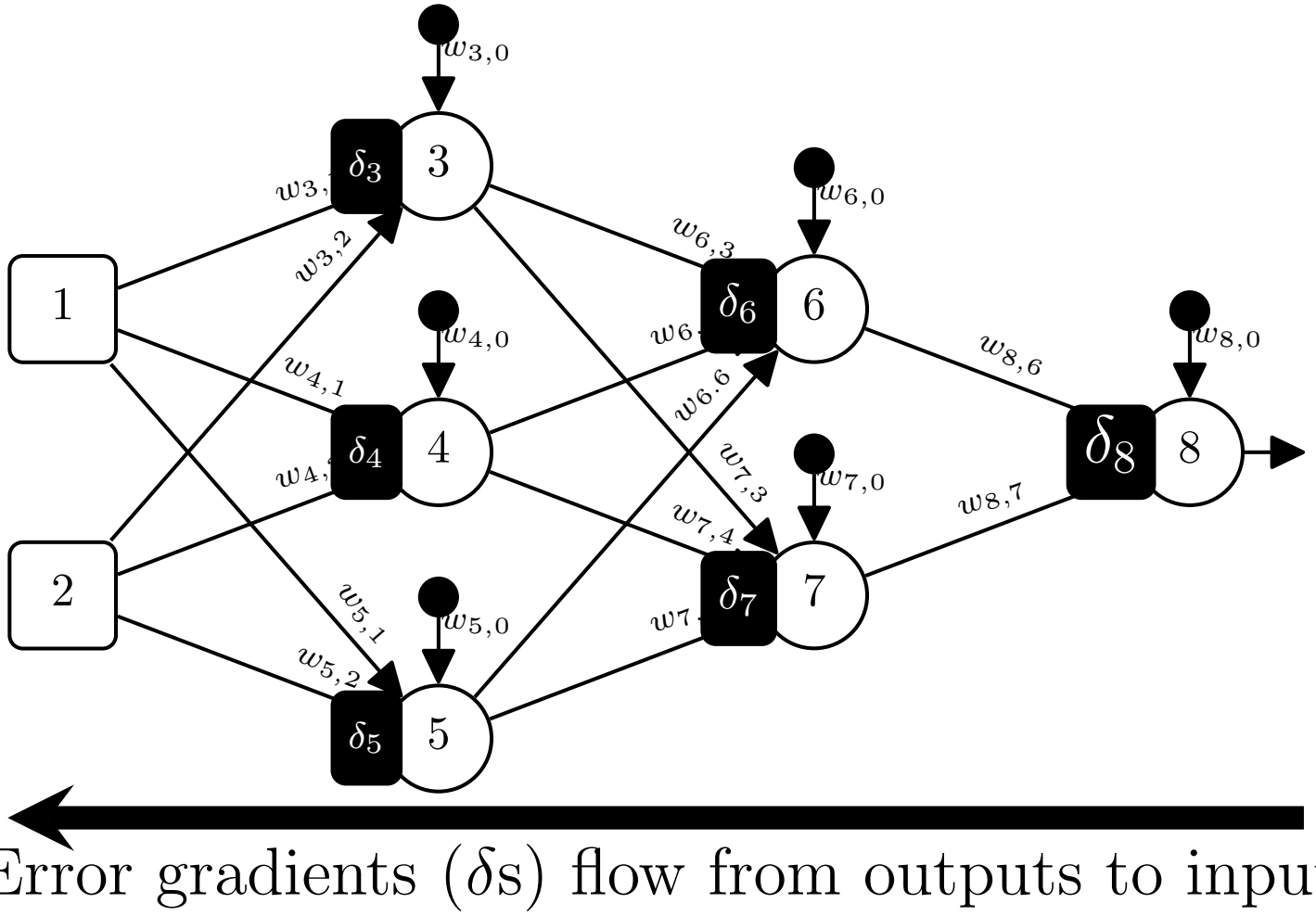


Figure 12: The backpropagation of the δ values during the backward pass of the backpropagation algorithm. This figure is based on Figure 6.6 of (**?**).

- The δ term for a neuron describes the rate of change of the error (i.e., the error gradient) of the network with respect to changes in the weighted sum calculated by the neuron.
- Let $\mathcal{E}$ denote the error of the network at the output layer, and z_k denote the weighted sum calculation in neuron k.
- the δ for a neuron k can be mathematically defined:

$$\boldsymbol{\delta}_k = \frac{\partial \mathcal{E}}{\partial z_k} \tag{13}$$

- The δs for ALL neurons in a network are calculated as the product of two terms:

$$\boldsymbol{\delta}_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k} \tag{14}$$

1. $\frac{\partial \mathcal{E}}{\partial a_k}$: the rate of change of the error of the network with respect to changes in the activation of the neuron; and
2. $\frac{\partial a_k}{\partial z_k}$: the rate of change of the activation of the neuron with respect to changes in the weighted sum calculated at the neuron.

The backpropagation algorithm assumes that the neurons in the network use differentiable activation functions; as such, the same process is used to calculate $\frac{\partial a_k}{\partial z_k}$ for all neurons as follows:

$$\frac{d}{dz} logistic(z) = logistic(z) \times (1 - logistic(z)) \tag{15}$$

The backpropagation algorithm assumes that the neurons in the network use differentiable activation functions; as such, the same process is used to calculate $\frac{\partial a_k}{\partial z_k}$ for all neurons as follows:

$$\frac{d}{dz} logistic(z) = logistic(z) \times (1 - logistic(z)) \tag{15}$$

- The graph of the logistic function is relatively flat for large (positive or negative) values.
- Regions of curves that are flat are said to be **saturated**.
- In these saturated regions the derivative of the logistic function is approximately 0.

- The weighted sum for each neuron is stored in the forward pass of the algorithm: it is used to calculate the $\frac{\partial a_k}{\partial z_k}$ term during the backpropagation process. (Using logistic activation functions made it relatively easy to implement the backpropagation algorithm.)

$$\begin{aligned}\frac{d}{dz}logistic(z=0) &= logistic(0) \times (1 - logistic(0)) \\ &= 0.5 \times (1 - 0.5) \\ &= 0.25\end{aligned} \tag{16}$$

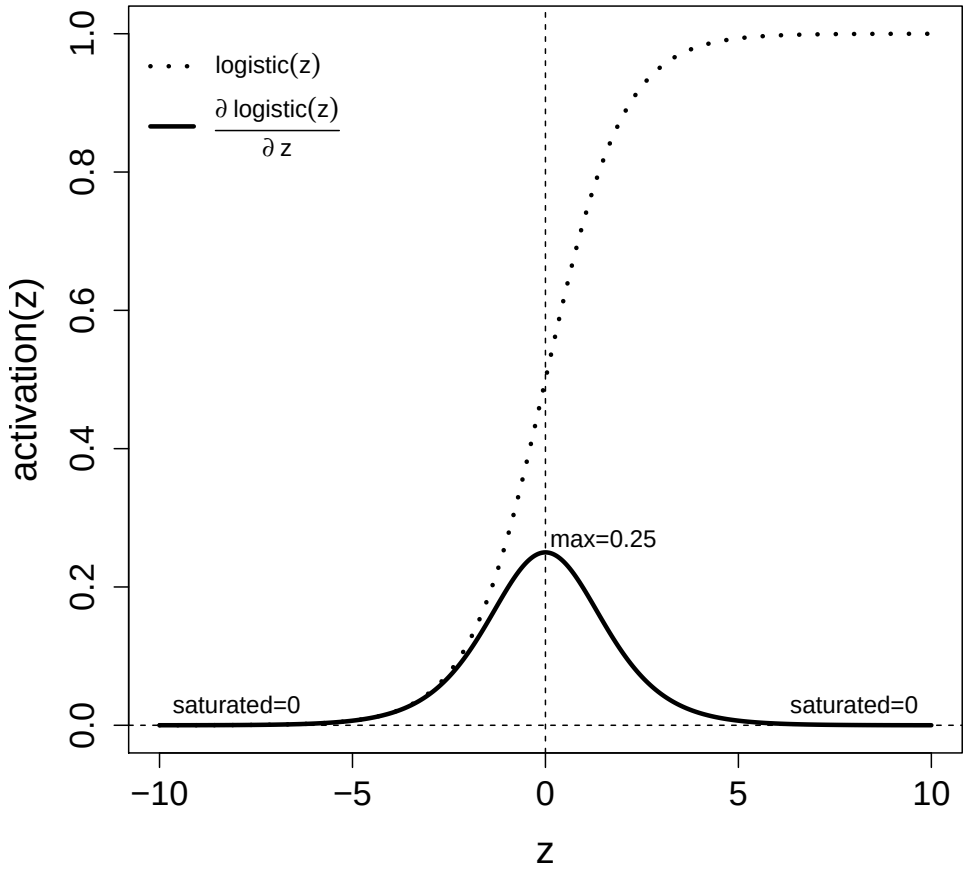


Figure 13: Plots of the logistic function and its derivative. This figure is Figure 4.6 of (**?**) and is used here with permission.

- **Recall from chapter 7**: the gradient of the error surface for multivariate linear regression is given as the partial derivative of L_2 with respect to each weight, $\mathbf{w}[j]$:

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{w}[j]} L_2\left(\mathbb{M}_{\mathbf{w}}, \mathcal{D}\right) &= \frac{\partial}{\partial \mathbf{w}[j]}\left(\frac{1}{2}\left(t - \mathbb{M}_{\mathbf{w}}\left(\mathbf{d}\right)\right)^2\right) && (17)\\
&= \left(t - \mathbb{M}_{\mathbf{w}}(\mathbf{d})\right) \times \frac{\partial}{\partial \mathbf{w}[j]}\left(t - \mathbb{M}_{\mathbf{w}}(\mathbf{d})\right) && (18)\\
&= \left(t - \mathbb{M}_{\mathbf{w}}(\mathbf{d})\right) \times \frac{\partial}{\partial \mathbf{w}[j]}\left(t - (\mathbf{w} \cdot \mathbf{d})\right) && (19)\\
&= \left(t - \mathbb{M}_{\mathbf{w}}(\mathbf{d})\right) \times -\mathbf{d}[j] && (20)
\end{aligned}$$

$$L_2(\mathbb{M}_{\mathbf{w}}, \mathcal{D}) = \frac{1}{2} \sum_{i=1}^{n} \left(t_i - \mathbb{M}_{\mathbf{w}} \left(\mathbf{d}_i\right)\right)^2 \tag{21}$$

Calculating $\frac{\partial \mathcal{E}}{\partial a_k}$, the rate of change of the error of the network with respect to changes in the activation of the neuron:

$$\frac{\partial}{\partial \mathbf{w}\left[j\right]} L_2\left(\mathbb{M}_{\mathbf{w}}, \mathbf{d}\right) = \left(t - \mathbb{M}_{\mathbf{w}}(\mathbf{d})\right) \times -\mathbf{d}\left[j\right] \tag{22}$$

- This derivation was based on the **chain rule** and is the product of the rate of change of the error of the model with respect to its output, the term $\left(t - \mathbb{M}_{\mathbf{w}}(\mathbf{d})\right)$ and the rate of change of the output of the linear regression model with respect to a change in the weight j, the term $-\mathbf{d}\left[j\right]$.

We want to define $\frac{\partial \mathcal{E}}{\partial a_k}$ the rate of change of the error of a neuron (a model) with respect to its activation (output). For this we need only the first term from the previous product.

$$\frac{\partial \mathcal{E}}{\partial a_k} = \frac{\partial L_2\left(\mathbb{M}_{\mathbf{w}}, \mathbf{d}\right)}{\partial \mathbb{M}_{\mathbf{w}}(\mathbf{d}))} = t - \mathbb{M}_{\mathbf{w}}(\mathbf{d}) = t_k - a_k \tag{23}$$

We want to define $\frac{\partial \mathcal{E}}{\partial a_k}$ the rate of change of the error of a neuron (a model) with respect to its activation (output). For this we need only the first term from the previous product.

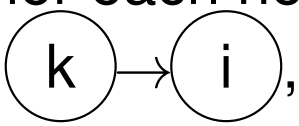
$$\frac{\partial \mathcal{E}}{\partial a_k} = \frac{\partial L_2\left(\mathbb{M}_{\mathbf{w}}, \mathbf{d}\right)}{\partial \mathbb{M}_{\mathbf{w}}(\mathbf{d}))} = t - \mathbb{M}_{\mathbf{w}}(\mathbf{d}) = t_k - a_k \tag{23}$$

However, the direction of the calculated gradient is toward the highest value on the error surface, and therefore to move down the error surface we should multiply it by -1. Hence:

$$\frac{\partial \mathcal{E}}{\partial a_k} = -\left(t_k - a_k\right) \tag{24}$$

Calculation of δ_k for neuron k in the output layer (making use of Equation 14[46]):

$$
\begin{aligned}
\boldsymbol{\delta}_k &= \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k} \\
&= \frac{\partial a_k}{\partial z_k} \times -(t_k - a_k) \\
&= \underbrace{\frac{d}{dz} logistic(z)}_{\text{Assuming a logistic activation function}} \times -(t_k - a_k) \\
&= \underbrace{(logistic(z) \times (1 - logistic(z)))}_{\text{Assuming a logistic activation function}} \times -(t_k - a_k) \qquad (25)
\end{aligned}
$$

- Calculating $\frac{\partial \mathcal{E}}{\partial a_k}$ for a hidden neuron.
- for each neuron i that neuron k connects forward to 

, the δ for neuron i connects the z_i value for i to the error of the network $\mathcal{E}$.

$$\frac{\partial \mathcal{E}}{\partial a_k} = \sum_{i=1}^{n} w_{i,k} \times \boldsymbol{\delta}_i \tag{26}$$

$$\boldsymbol{\delta}_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k} \tag{27}$$

$$\boldsymbol{\delta}_k = \frac{\partial a_k}{\partial z_k} \times \left(\sum_{i=1}^{n} w_{i,k} \times \boldsymbol{\delta}_i \right)$$

$$= \underbrace{\frac{d}{dz} logistic(z)}_{\text{Assuming a logistic activation function}} \times \left(\sum_{i=1}^{n} w_{i,k} \times \boldsymbol{\delta}_i \right)$$

$$= \underbrace{(logistic(z) \times (1 - logistic(z)))}_{\text{Assuming a logistic activation function}} \times \left(\sum_{i=1}^{n} w_{i,k} \times \boldsymbol{\delta}_i \right) \quad (28)$$

- The basic principle of weights adjustments: a weight should be updated in proportion to the sensitivity of the network error to changes in this weight.
- The rate of change of the network error with respect to changes in a weight is denoted $\frac{\partial \mathcal{E}}{\partial w_{i,k}}$.
- Using the chain rule, we can rewrite this term as a product of three terms:

$$\frac{\partial \mathcal{E}}{\partial w_{i,k}} = \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i} \times \frac{\partial z_i}{\partial w_{i,k}} \tag{29}$$

The previous equation uses the product of the following:

- the rate of change of the weighted sum function with respect to changes in one of the weights ($\frac{\partial z_i}{\partial w_{i,k}}$);
- the rate of change of the activation function with respect to changes in the weighted sum ($\frac{\partial a_i}{\partial z_i}$); and
- the rate of change of the error of the network with respect to changes in the activation function ($\frac{\partial \mathcal{E}}{\partial a_i}$).

The previous equation uses the product of the following:

- the rate of change of the weighted sum function with respect to changes in one of the weights ($\frac{\partial z_i}{\partial w_{i,k}}$);
- the rate of change of the activation function with respect to changes in the weighted sum ($\frac{\partial a_i}{\partial z_i}$); and
- the rate of change of the error of the network with respect to changes in the activation function ($\frac{\partial \mathcal{E}}{\partial a_i}$).

This indeed covers the impact of weight changes all the way till the network error.

Since we know from Equation 14[46] that

$$\boldsymbol{\delta}_k = \frac{\partial a_k}{\partial z_k} \times \frac{\partial \mathcal{E}}{\partial a_k} \tag{30}$$

$$\begin{aligned} \frac{\partial \mathcal{E}}{\partial w_{i,k}} &= \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i} \times \frac{\partial z_i}{\partial w_{i,k}} \\ &= \boldsymbol{\delta}_i \times \frac{\partial z_i}{\partial w_{i,k}} \end{aligned} \tag{31}$$

And since

$$\frac{\partial z_i}{\partial w_{i,k}} = a_k \tag{32}$$

Then

$$\frac{\partial \mathcal{E}}{\partial w_{i,k}} = \boldsymbol{\delta}_i \times a_k \tag{33}$$

The weight update rule is

$$w_{i,k} \leftarrow w_{i,k} - \alpha \times \boldsymbol{\delta}_i \times a_k \tag{34}$$

where α is the learning rate hyper-parameter and has the same function as the learning rate in gradient descent.

- It states that the updated weight after processing a training example is equal to the weight used to process the training example minus α times the sensitivity of the error of the network with respect to changes in the weight.
- Updating weights after each training example is known as on-line, sequential, or stochastic gradient descent.

The weight update rule is

$$w_{i,k} \leftarrow w_{i,k} - \alpha \times \boldsymbol{\delta}_i \times a_k \tag{34}$$

where α is the learning rate hyper-parameter and has the same function as the learning rate in gradient descent.

- It states that the updated weight after processing a training example is equal to the weight used to process the training example minus α times the sensitivity of the error of the network with respect to changes in the weight.
- Updating weights after each training example is known as on-line, sequential, or stochastic gradient descent.
- **Problem with stochastic gradient descent:** the error gradient calculated on a single example is likely to be a noisy approximation of the true gradient over the entire dataset.
- But, typically, stochastic gradient descent still works on the entire dataset.

- Ideally, we want to descend the true error gradient for the entire dataset.
- Batch gradient descent involves calculating the error gradients for each weight $w_{i,k}$ over one complete pass for all the m examples in a dataset and summing the gradients for each such weight, and only then updating the weights using the summed error gradients calculated over the entire dataset.

$$\Delta w_{i,k} = \sum_{j=1}^{m} \boldsymbol{\delta}_{i,j} \times a_{k,j} \tag{35}$$

$$w_{i,k} \leftarrow w_{i,k} - \alpha \times \Delta w_{i,k} \tag{36}$$

1. **Epoch**:
   - An epoch is a single pass through all the examples in the training dataset.
   - If the m dataset training examples are processed in parallel, a single epoch is completed in each iteration of the algorithm.
   - There is a single weight update per epoch.

1. **Epoch**:
   - An epoch is a single pass through all the examples in the training dataset.
   - If the m dataset training examples are processed in parallel, a single epoch is completed in each iteration of the algorithm.
   - There is a single weight update per epoch.
2. **Iteration**
   - The term iteration is used to refer to a single forward and backward pass plus weight update of the backpropagation algorithm.
   - If we have m examples in stochastic gradient descent, it would take m iterations to complete a single epoch.
   - This epoch would involve m weight updates.

Some advantages of batch gradient descent:

- A neural network, implemented as a sequence of matrix operations, can process multiple examples in parallel.
- Calculating a direction of descent by taking an average over a set of noisy examples can make the descent of the error surface smoother.
- A smoother descent means that we can use a larger learning rate with batch gradient descent.
- This combination of multiple examples processed in parallel and a larger learning rate can result in much faster training times using batch gradient descent.

Drawback of batch gradient descent

- The entire dataset must be processed between each weight update.
- Modern datasets can be very large (quite possible to have millions or even billions of examples).
- Even with modern computational power and with processing examples in parallel, batch gradient descent can take a long time, resulting in network training taking a long time.

Approach to mitigate the drawback of batch gradient descent: Use **mini-batch gradient descent**, a standard practice in deep learning.

- The dataset is split into multiple subsets of the same size called **mini-batches** or **batches**.
- Ideally, each mini-batch should be created by random sampling from the dataset.
- At the start of training, the dataset is shuffled and then split into a sequence of mini-batches.
- The first iteration of training is done on the first mini-batch in the sequence, the second iteration on the second mini-batch, and so on until the epoch is completed.
- At the start of the second epoch, the sequence of mini-batches is shuffled and the training iterations are carried out on this new sequence of mini-batches.

Approach to mitigate the drawback of batch gradient descent – choice of the batch size:

- Larger batches provide a more accurate estimate of the true gradient for the entire dataset;
- Hardware constraints may necessitate the use of smaller batches. E.g., if parallel processing of mini-batches, then the larger the batch size, the larger is the memory requirement.
- If all the examples in a mini-batch are to be processed in parallel, then the larger the batch size, the larger is the memory requirement.
- Popular batch sizes include 32, 64, 128, and even 256 examples.
- Finding a good combination of values for the batch size and learning rate hyper-parameters involves trial-and-error experimentation.

Require: set of training instances $\mathcal{D}$
Require: a learning rate α that controls how quickly the algorithm converges
Require: a batch size B specifying the number of examples in each batch
Require: a convergence criterion

Shuffle $\mathcal{D}$ and create the mini-batches: $[(\mathbf{X}^{(1)}, \mathbf{Y}^{(1)}), \ldots, (\mathbf{X}^{k}, \mathbf{Y}^{k})]$
Initialize the weight matrices for each layer: $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}$
repeat ▷ Each repeat loop is one epoch
    for t=1 to number of mini-batches **do** ▷ Each for loop is one iteration
        $\mathbf{A}^{(0)} \leftarrow \mathbf{X}^{(t)}$
        for l=1 to L **do**
            $\mathbf{v} \leftarrow [1_0, \ldots 1_m]$ ▷ Create $\mathbf{v}$ the vector of bias terms
            $\mathbf{A}^{(l-1)} \leftarrow [\mathbf{v}; \mathbf{A}^{(l-1)}]$ ▷ Insert $\mathbf{v}$ into the activation matrix
            $\mathbf{Z}^{(l)} \leftarrow \mathbf{W}^{l}\mathbf{A}^{(l-1)}$
            $\mathbf{A}^{(l)} \leftarrow \varphi(\mathbf{Z}^{(l)})$ ▷ Elementwise application of φ to $\mathbf{Z}^{(l)}$
        end for
        for each weight $w_{i,k}$ in the network **do**
            $\Delta w_{i,k} = 0$
        end for
        for each example in the mini-batch **do** ▷ Backpropagate the δs
            for each neuron i in the output layer **do**
                $\delta_i = \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i}$ ▷ See Equation (25)[55]
            end for
            for l = L-1 to 1 **do**
                for each neuron i in the layer l **do**
                $\delta_i = \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i}$ ▷ See Equation (28)[57]
                end for
            end for
            for each weight $w_{i,k}$ in the network **do**
                $\Delta w_{i,k} = \Delta w_{i,k} + (\delta_i \times a_k)$ ▷ Equation (35)[64]
            end for
        end for
        for each weight $w_{i,k}$ in the network **do**
            $w_{i,k} \leftarrow w_{i,k} - \alpha \times \Delta w_{i,k}$ ▷ Equation (36)[64]
        end for
    end for
    shuffle($[(\mathbf{X}^{(1)}, \mathbf{Y}^{(1)}), \ldots, (\mathbf{X}^{k}, \mathbf{Y}^{k})]$)
until convergence occurs

Table 1: 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

- **Normalization** of descriptive features is a standard data preprocessing practice for ANNs.
- Range normalization or standardization ensures that all weight updates are brought to a similar scale.
- If large differences in different descriptive feature values, then the connection weights from input layer to first hidden layer will impact the learning since weight updates are scaled by feature values.
- Same for weight updates for bias inputs (always 1) and weights on inputs with values much larger than 1.
- Large weight updates can result in instability in model training.
- If a model has a relatively large weight on one input feature, then the model output can be very sensitive to small changes in the value of high-value features. So model outputs can be very different for similar input vectors.

- In a regression problem with continuous target, normalization often applied to both descriptive and target features.
- One reason: many activation functions have a small output range (e.g., logistic function outputs $\in [0, 1]$), so appropriate that target feature matches the activation function output range.
- With normalized target feature, during training, the network error on an example can be calculated by directly comparing the network output with normalized target feature value.
- So, to retrieve the predicted target value in the original range of the target feature, the network output must be mapped back to the original target feature scale.
- E.g. if target feature originally $\in [min, max]$ and range normalization used (range $[0, 1]$), then output layer activation a_i of a logistic unit can be mapped to value in original target feature range by $max \times a_i$.

- Alternatively, one can use linear units in the output layer (i.e., units do not use an activation function and simply output the weighted sum z), so the network can have the same range as the non-normalized target feature.
- Advantage: outputs of the network do not have to be transformed back into the original target feature range.
- Drawback: if the target feature has a large range, then during training the error of the network on an example can be very large $\implies$ very large error gradients $\implies$ large weight updates $\implies$ unstable learning process.

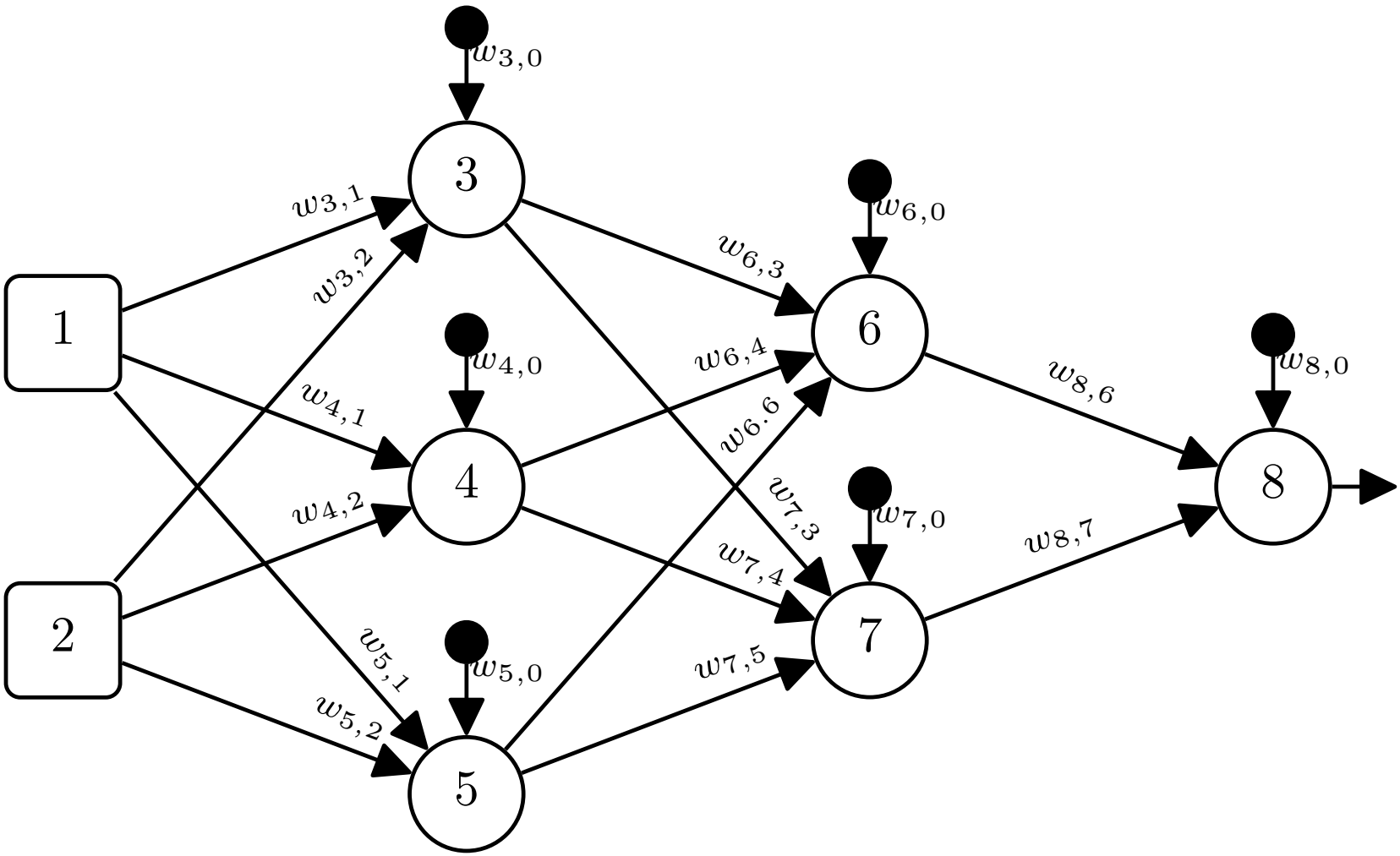
Table 2: The minimum and maximum values for the AMBIENT TEMPERATURE, RELATIVE HUMIDITY, and ELECTRICAL OUTPUT features in the power plant dataset.

	AMBIENT TEMPERATURE	RELATIVE HUMIDITY	ELECTRICAL OUTPUT
Min	$1.81°$C	25.56%	420.26MW
Max	$37.11°$C	100.16%	495.76MW

Table 3: The *range-normalized* hourly samples of ambient factors and full load electrical power output of a combined cycle power plant, rounded to two decimal places.

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

Network architecture used as the structure for the model to be trained:



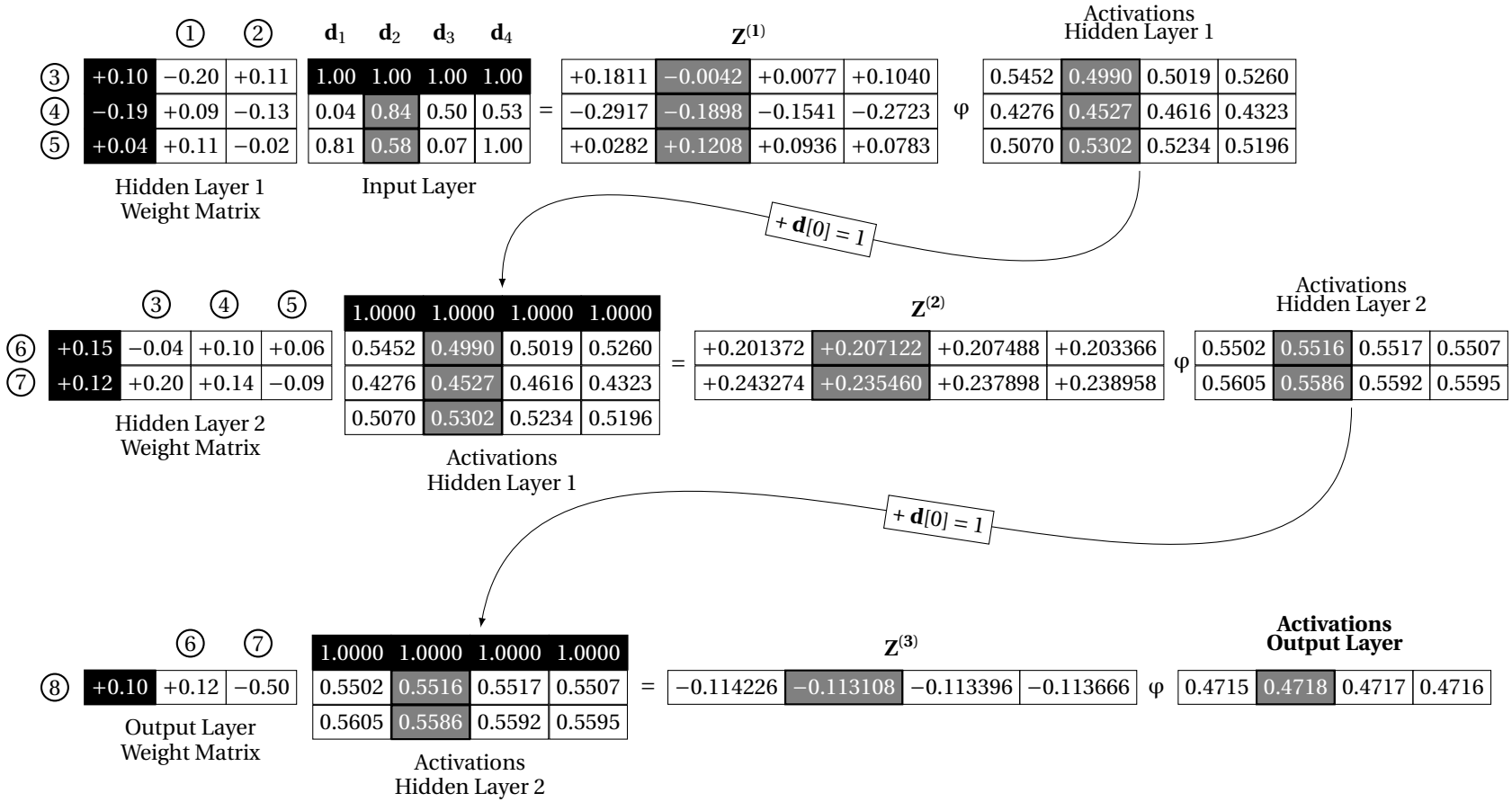


Figure 14: The forward pass of the examples listed in Table 3[78] through the network in Figure 4[18]. The weight matrices were randomly initialized from the range $[-0.5, 0.5]$

Table 4: The per example error after the forward pass illustrated in Figure 14[80], 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

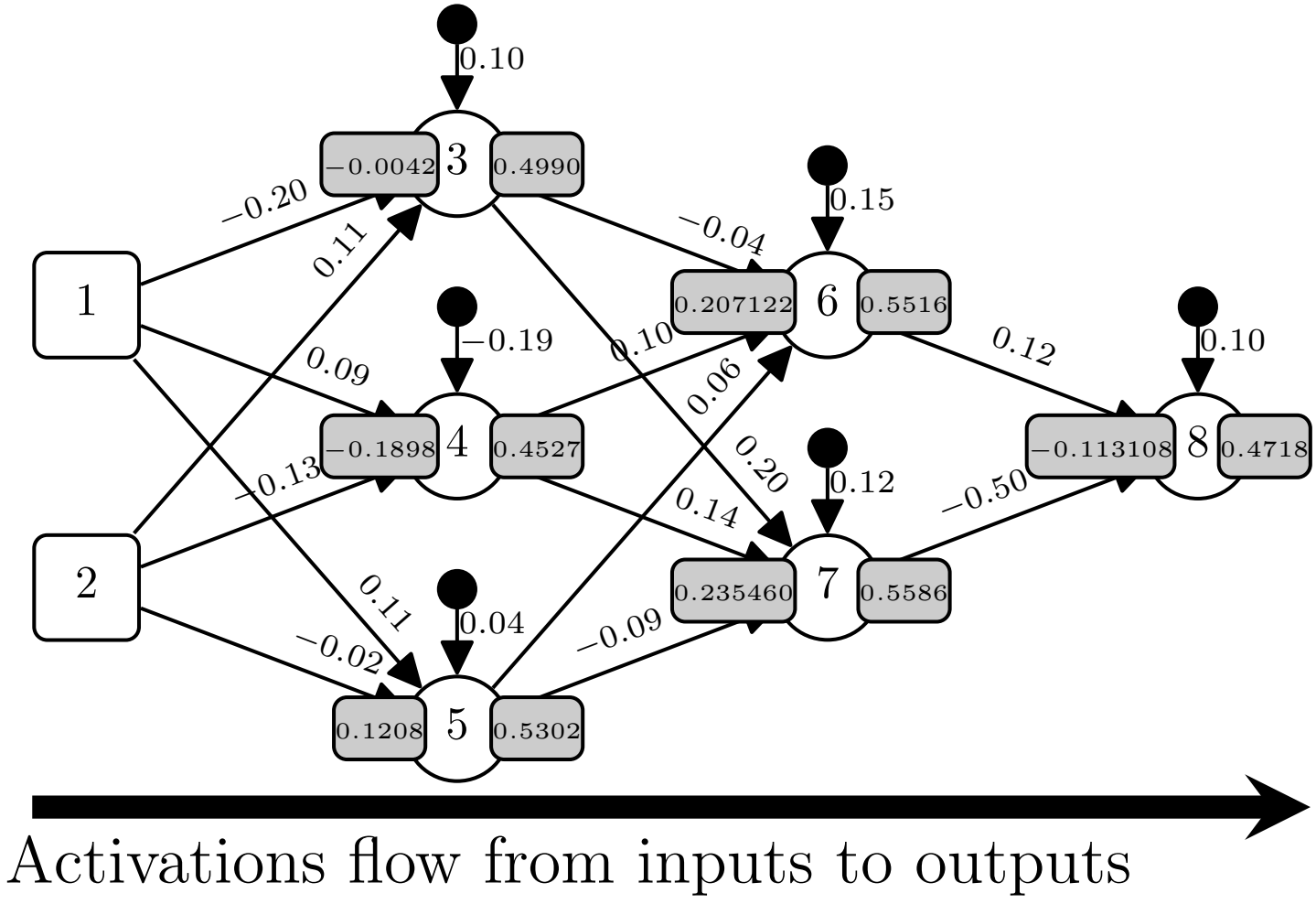


Figure 15: An illustration of the forward propagation of $\mathbf{d}_2$ through the network showing the weights on each connection, and the weighted sum z and activation a value for each neuron in the network.

$$\frac{\partial \mathcal{E}}{\partial a} = 0.3418 \tag{37}$$

$$\begin{aligned} \delta_8 = & \frac{\partial \mathcal{E}}{\partial a_8} \times \frac{\partial a_8}{\partial z_8} \\ =& 0.3418 \times 0.2492 \\ =& 0.0852 \end{aligned} \tag{38}$$

Table 5: The $\partial a/\partial z$ for each neuron for Example 2 rounded to four decimal places.

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

$$
\begin{aligned}
\delta_6 = & \frac{\partial \mathcal{E}}{\partial a_6} \times \frac{\partial a_6}{\partial z_6} \\
= & \left(\sum \delta_i \times w_{i,6} \right) \times \frac{\partial a_6}{\partial z_6} \\
= & \left(\delta_8 \times w_{8,6} \right) \times \frac{\partial a_6}{\partial z_6} \\
= & (0.0852 \times 0.12) \times 0.2473 \\
= & \quad 0.0025
\end{aligned} \tag{39}
$$

$$
\begin{aligned}
\delta_7 = & \frac{\partial \mathcal{E}}{\partial a_7} \times \frac{\partial a_7}{\partial z_7} \\
= & \left(\sum \delta_i \times w_{i,7} \right) \times \frac{\partial a_7}{\partial z_7} \\
= & \left(\delta_8 \times w_{8,7} \right) \times \frac{\partial a_6}{\partial z_6} \\
= & (0.0852 \times -0.50) \times 0.2466 \\
= & -0.0105
\end{aligned} \tag{40}
$$

$$
\begin{aligned}
\delta_3 = & \frac{\partial \mathcal{E}}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \\
= & \left(\sum \delta_i \times w_{i,3} \right) \times \frac{\partial a_3}{\partial z_3} \\
= & \left((\delta_6 \times w_{6,3}) + (\delta_7 \times w_{7,3}) \right) \times \frac{\partial a_3}{\partial z_3} \\
= & ((0.0025 \times -0.04) + (-0.0105 \times 0.20)) \times 0.2500 \\
= & -0.0006
\end{aligned} \tag{41}
$$

$$
\begin{aligned}
\delta_4 = & \frac{\partial \mathcal{E}}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \\
= & \left(\sum \delta_i \times w_{i,4} \right) \times \frac{\partial a_4}{\partial z_4} \\
= & \left((\delta_6 \times w_{6,4}) + (\delta_7 \times w_{7,4}) \right) \times \frac{\partial a_4}{\partial z_4} \\
= & ((0.0025 \times 0.10) + (-0.0105 \times 0.14)) \times 0.2478 \\
= & -0.0003
\end{aligned} \tag{42}
$$

$$
\begin{aligned}
\delta_5 = & \frac{\partial \mathcal{E}}{\partial a_5} \times \frac{\partial a_5}{\partial z_5} \\
= & \left(\sum \delta_i \times w_{i,5} \right) \times \frac{\partial a_5}{\partial z_5} \\
= & \left((\delta_6 \times w_{6,5}) + (\delta_7 \times w_{7,5}) \right) \times \frac{\partial a_5}{\partial z_5} \\
= & ((0.0025 \times 0.06) + (-0.0105 \times -0.09)) \times 0.2491 \\
= & \quad 0.0003 \qquad (43)
\end{aligned}
$$

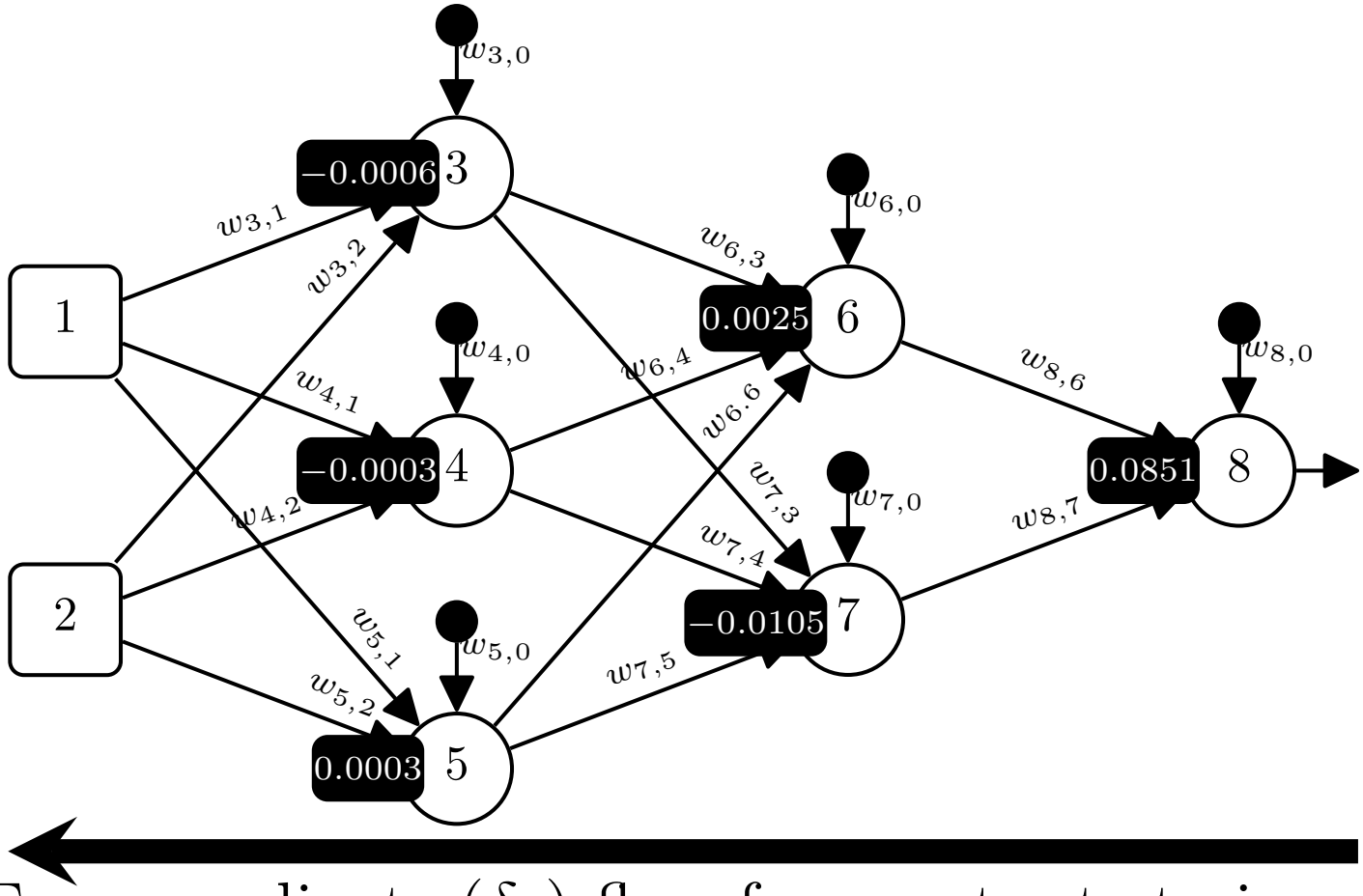


Figure 16: The δs for each of the neurons in the network for Ex. 2.

Table 6: 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	$\partial \mathcal{E}/\partial w_{i,k}$
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	$-0.0006 \times 0.58 = -0.000348$

$$
\begin{aligned}
w_{7,5} &= w_{7,5} - \alpha \times \boldsymbol{\delta}_7 \times a_5 \\
&= w_{7,5} - \alpha \times \frac{\partial \mathcal{E}}{\partial w_{i,k}} \\
&= -0.09 - 0.2 \times -0.0055671 \\
&= -0.08888658
\end{aligned} \tag{44}
$$

Table 7: The calculation of $\Delta w_{7,5}$ across our four examples.

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

$$\begin{aligned} w_{7,5} &= w_{7,5} - \alpha \times \Delta w_{i,k} \\ &= -0.09 - 0.2 \times 0.00153726 \\ &= -0.0903074520 \end{aligned} \tag{45}$$

Table 8: 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

Table 9: The per example prediction, error, and the sum of squared errors after training on the four examples has converged after 7656 epochs to an $SSE < 0.0001$.

	$\mathbf{d}_1$	$\mathbf{d}_2$	$\mathbf{d}_3$	$\mathbf{d}_4$
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

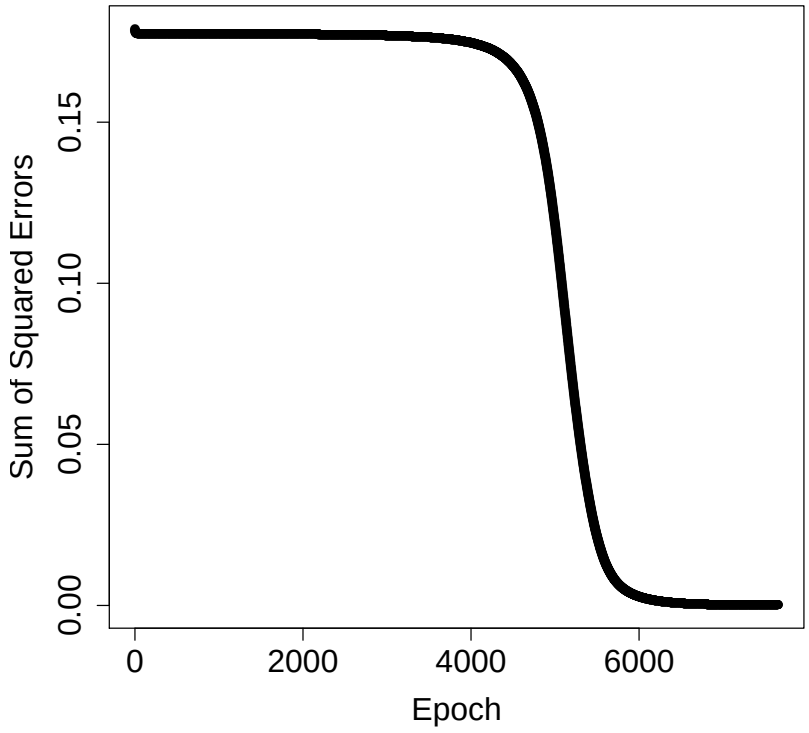


Figure 17: A plot showing how the sum of squared errors of the network changed during training.

- Other texts on neural networks and deep learning: (**????**)
- Programming focused introductions to deep learning: (**??**)
- Introduction to neural networks for natural language processing: (**?**)
- Computer architecture perspective on deep learning: (**?**)
- Recent developments in the field: **batch normalization** (**?**), adaptively learning algorithms such as **Adam** (**?**), **Generative Adversarial Networks** (**?**), and attention-based architectures such as the **Transformer** (**?**).

Extensions and Variations

Recall the network we have worked on:

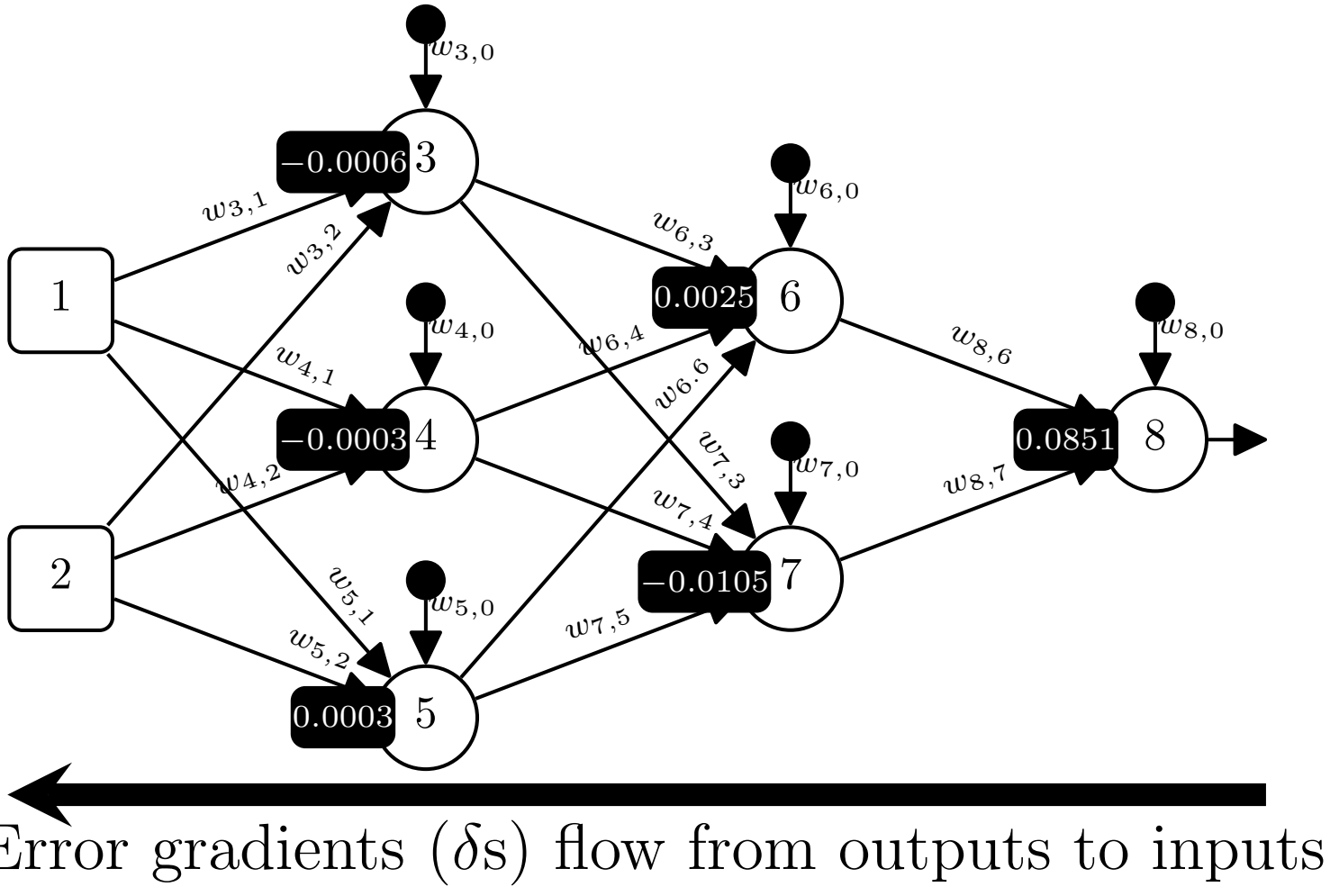


Figure 18: The δs for each of the neurons in the network for Ex. 2.

- One of the biggest challenges in training a deep neural network is to ensure that the flow of error gradients back through its layers is stable during training.
- In the previous network, if we compare the δ for Neuron 8 with the δs for the neurons in first hidden layer (Neurons 3, 4, and 5), it becomes clear that the δ values become smaller as they are propagated back from the output layer to the earlier layers.

- One of the biggest challenges in training a deep neural network is to ensure that the flow of error gradients back through its layers is stable during training.
- In the previous network, if we compare the δ for Neuron 8 with the δs for the neurons in first hidden layer (Neurons 3, 4, and 5), it becomes clear that the δ values become smaller as they are propagated back from the output layer to the earlier layers.
- This phenomenon is known as the **vanishing gradient problem**: in deep networks the δ terms tend toward zero and vanish as the δs are propagated back to the early layers.
- This is a serious challenge for training deep networks: as the δs vanish, the learning signal attenuates, and this can slow down the rate at which the early layers of the network learn.

Table 10: 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.

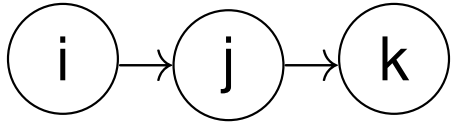
NEURON$_i$	NEURON$_k$	$w_{i,k}$	δ_i	a_k	$\partial \mathcal{E}/\partial w_{i,k}$
8	0	$w_{8,0}$	0.0852	1	$0.0852 \times 1 = 0.0852$
8	6	$w_{8,6}$	0.0852	0.5516	$0.0852 \times 0.5516 = 0.04699632$
8	7	$w_{8,7}$	0.0852	0.5586	$0.0852 \times 0.5586 = 0.04759272$
7	0	$w_{7,0}$	-0.0105	1	$-0.0105 \times 1 = -0.0105$
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	$-0.0006 \times 0.58 = -0.000348$

- In Vanishing Gradient, the weights in the earlier layers in the network are updated by smaller amounts than the neurons in the later layers. See the right-hand column of the next table (already seen in the worked example.)
- This means that the neurons in the first hidden layer will learn more slowly than the output layer neurons.
- Also the deeper the network becomes, the slower the earlier layers learn.
- A serious problem because the ability of a deep neural network to learn a useful representation of the inputs works by the earlier layers of the network extracting low-level features from the raw data and then the later layers learning to combine these features in useful ways.

- The vanishing gradient problem can cause deep networks to take a very long time to train.
- It is a direct consequence of the fact that the backpropagation algorithm is based on the chain rule.
- Using backpropagation, the error gradient at any point in the network is a product of the gradients up to that point.

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_{i,k}} &= \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i} \times \frac{\partial z_i}{\partial w_{i,k}} \\ &= \boldsymbol{\delta}_i \times \frac{\partial z_i}{\partial w_{i,k}}\end{aligned} \tag{46}$$

Consider a simple feedforward network with just three neurons, i, j, and k, arranged so that i feeds forward into j and j into k.



$$
\begin{aligned}
\delta_i &= \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i} \\
&= \overbrace{w_{j,i} \times \delta_j}^{\frac{\partial \mathcal{E}}{\partial a_i}} \times \frac{\partial a_i}{\partial z_i} \\
&= w_{j,i} \times \overbrace{w_{k,j} \times \delta_k \times \frac{\partial a_j}{\partial z_j}}^{\frac{\partial \mathcal{E}}{\partial a_j}} \times \frac{\partial a_i}{\partial z_i} \\
&= w_{j,i} \times w_{k,j} \times \overbrace{\frac{\partial \mathcal{E}}{\partial a_k} \times \frac{\partial a_k}{\partial z_k}}^{\delta_k} \times \frac{\partial a_j}{\partial z_j} \times \frac{\partial a_i}{\partial z_i}
\end{aligned}
\tag{47}
$$

- Multiplying a number by a number less than 1 makes the number smaller.
- Previous Equation shows that during backpropagation the error gradient is repeatedly multiplied by a derivative of activation function, one multiplication for each neuron the error gradient is backpropagated through.
- When a neuron uses a logistic function as its activation, then the maximum value the derivative can take is 0.25.
- $\implies$ backpropagating the error gradient through a neuron that uses a logistic function involves multiplying the error term by a value ≤ 0.25 and that is ≈ 0 for z values in the saturated regions of the logistic function.
- The error gradient will get smaller and smaller as it is backpropagated through the networks layers.
- The scaling down of the gradient is particularly severe for neurons whose z value is in the saturated region of the activation function.

One way to address the vanishing gradient problem is to use a different activation function, namely the rectified linear function:

$$rectifier(z) = max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & otherwise \end{cases} \tag{48}$$

Its derivative is:

$$\frac{d}{dz} rectifier(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & otherwise \end{cases} \tag{49}$$

- During backpropagation for some neurons in a network, the δ value will be pushed to zero.
- This is not necessarily a problem: if we consider the backpropagation process as error gradients δs flowing through a layer of neurons then, although for some neurons in the layer the δs will go to zero, for others the gradients will pass through unscaled.

- With ReLu, the error gradients in the backward pass can propagate along the paths of active neurons in the network, and because $\frac{\partial a_k}{\partial z_k} = 1$ along these paths, the gradients can flow back through a deeper network. (Neurons that use a rectified linear activation function are known as ReLUs, short for rectified linear units.)
- Technically, the derivative of the rectifier function is undefined when $z_k = 0$. So, strictly, the rectifier function is not an appropriate choice for gradient descent and backpropagation, which assume differentiable functions.
- A common practice in neural networks is to choose a derivative value for $z_k = 0$ of zero.
- This has been found to generally work well.

- With ReLu, the error gradients in the backward pass can propagate along the paths of active neurons in the network, and because $\frac{\partial a_k}{\partial z_k} = 1$ along these paths, the gradients can flow back through a deeper network. (Neurons that use a rectified linear activation function are known as ReLUs, short for rectified linear units.)
- Technically, the derivative of the rectifier function is undefined when $z_k = 0$. So, strictly, the rectifier function is not an appropriate choice for gradient descent and backpropagation, which assume differentiable functions.
- A common practice in neural networks is to choose a derivative value for $z_k = 0$ of zero.
- This has been found to generally work well.
- The following Figure shows the forward propagation of the previous worked example through the same network, this time with all the neurons being now ReLUs.

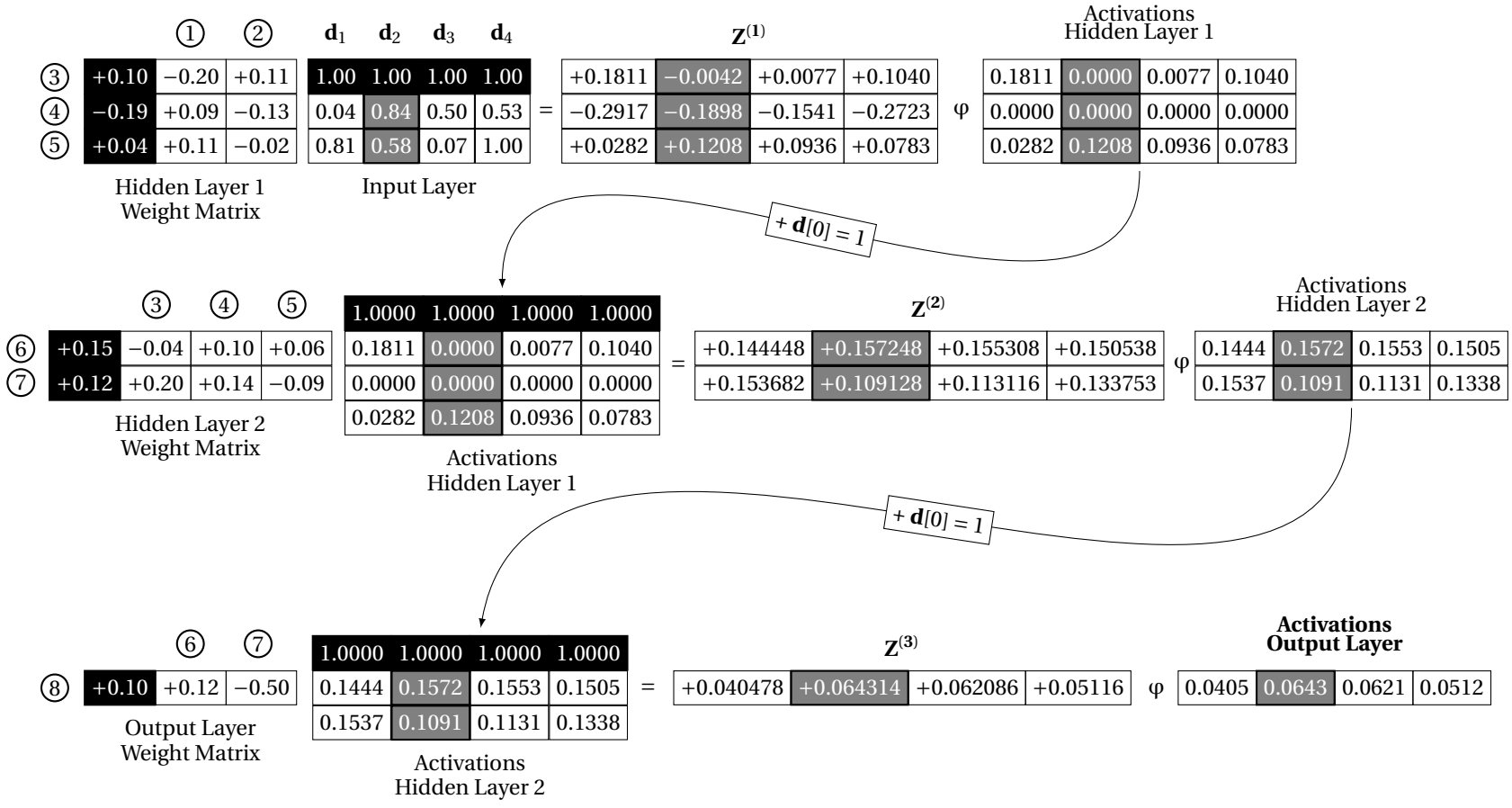


Figure 19: The forward pass of the examples listed in Table 14[160] through the network in Figure 4[18] when all the neurons are ReLUs.

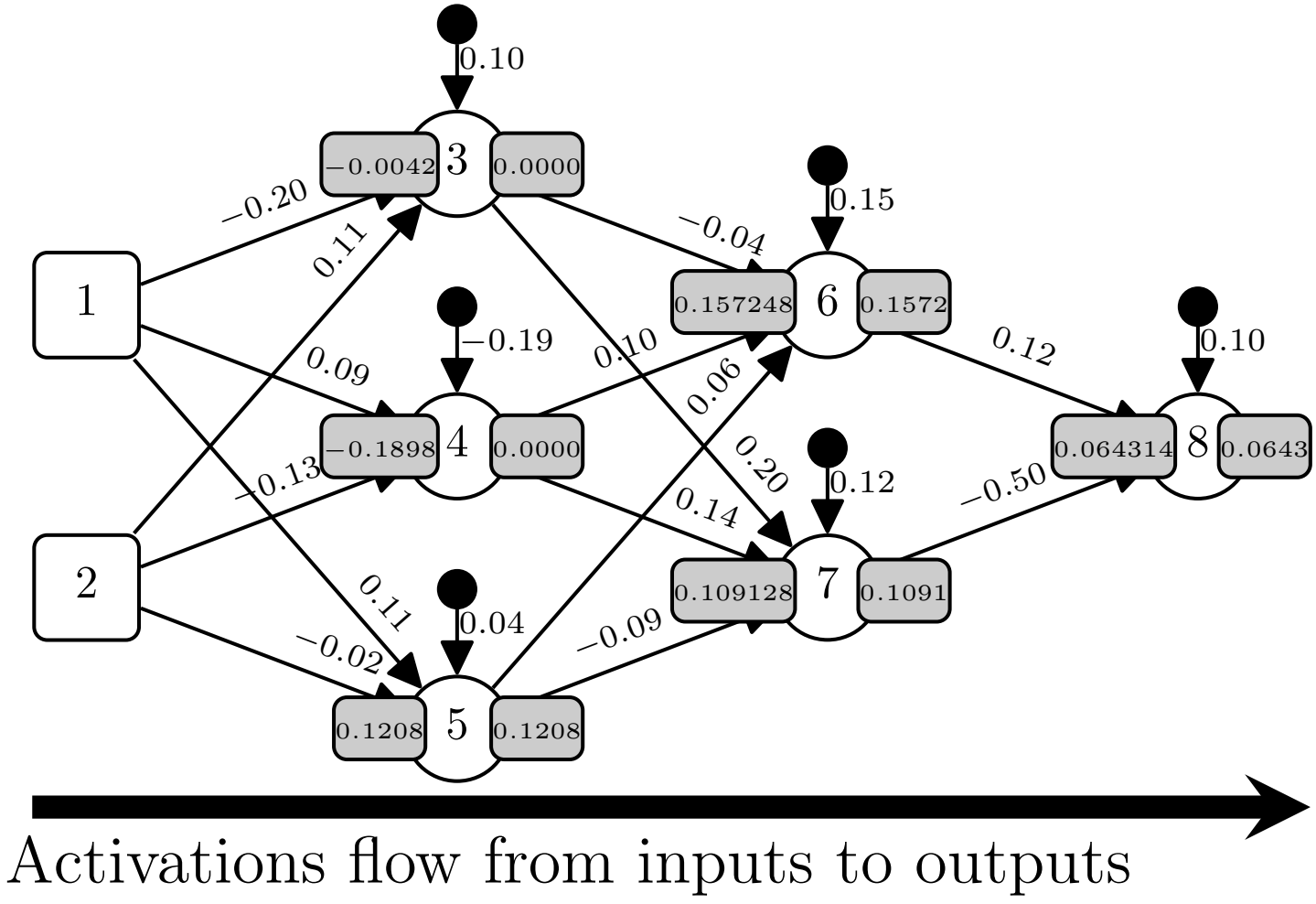


Figure 20: An illustration of the forward propagation of $\mathbf{d}_2$ through the ReLU network showing the weights on each connection, and the weighted sum z and activation a value for each neuron in the network.

Table 11: The per example error of the ReLU network after the forward pass illustrated in Figure 19[111], the per example $\partial \mathcal{E}/\partial a_8$, and the **sum of squared errors** for the ReLU model.

	$\mathbf{d}_1$	$\mathbf{d}_2$	$\mathbf{d}_3$	$\mathbf{d}_4$
Target	0.9400	0.1300	0.5700	0.3600
Prediction	0.0405	0.0643	0.0621	0.0512
Error	0.8995	0.0657	0.5079	0.3088
$\partial \mathcal{E}/\partial a_8$: Error $\times -1$	-0.8995	-0.0657	-0.5079	-0.3088
Error^2	0.80910025	0.00431649	0.25796241	0.09535744
SSE:				0.58336829

- In the previous figure, both Neurons 3 and 4 have an activation of zero for d_2. The reason is that both these neurons happen to have large negative weights on at least one of their inputs.
- However, the forward propagation through the network still occurs along an active path through Neuron 5.
- We can thus see that the error gradient will be able to flow backward through Neurons 8, 7, and 6 without being scaled down by the multiplication by $\frac{\partial a}{\partial z}$ in the δ calculations.

In order to calculate and backpropagate the δs for d_2 through the network, we need the $\frac{\partial a}{\partial z}$ for each neuron in the network for this example:

Table 12: The $\partial a/\partial z$ for each neuron for $\mathbf{d}_2$ rounded to four decimal places.

NEURON	z	$\partial a/\partial z$
3	-0.004200	0
4	-0.189800	0
5	0.120800	1
6	0.157248	1
7	0.109128	1
8	0.064314	1

- We can see that the error gradient will be able to flow backward through Neurons 8, 7, and 6 without being scaled down by the multiplication by $\frac{\partial a}{\partial z}$ in the δ calculations.
- Comparing the δs for the ReLU network with those calculated for the logistic network for the same example shows that, where the derivative of the ReLU has not pushed the δs to 0, the drop in the magnitude of the δs as we move back through the network is not as severe.
- The fact that both Neurons 3 and 4 had an activation of 0 in response to d_2 resulted in both of these neurons having a $\delta = 0$. This means that weights for these neurons will not be updated for this example.
- Initializing the network with the same initial set of weights (as with logistic activation) and same data, training regime, and learning rate ($\alpha = 0.2$), then the network ReLU version converges to an $SSE < 0.0001$ in just 424 epochs, as compared with the 7656 epochs for the logistic network.

$$
\begin{aligned}
\delta_k &= \frac{\partial \mathcal{E}}{\partial a_k} \times \frac{\partial a_k}{\partial z_i} \\
\delta_8 &= -0.0657 \times 1.0 \\
&= -0.0657 \\
\delta_7 &= \left(\delta_8 \times w_{8,7}\right) \times \frac{\partial a_7}{\partial z_7} \\
&= (-0.0657 \times -0.50) \times 1 \\
&= 0.0329 \\
\delta_6 &= \left(\delta_8 \times w_{8,6}\right) \times \frac{\partial a_6}{\partial z_6} \\
&= (-0.0657 \times 0.12) \times 1 \\
&= -0.0079 \\
\delta_5 &= \left(\left(\delta_6 \times w_{6,5}\right) + \left(\delta_7 \times w_{7,5}\right)\right) \times \frac{\partial a_5}{\partial z_5} \\
&= ((-0.0079 \times 0.06) + (0.0329 \times -0.09)) \times 1 \\
&= -0.0034 \\
\delta_4 &= \left(\left(\delta_6 \times w_{6,4}\right) + \left(\delta_7 \times w_{7,4}\right)\right) \times \frac{\partial a_4}{\partial z_4} \\
&= ((-0.0079 \times 0.10) + (0.0329 \times 0.14)) \times 0 \\
&= 0 \\
\delta_3 &= \left(\left(\delta_6 \times w_{6,3}\right) + \left(\delta_7 \times w_{7,3}\right)\right) \times \frac{\partial a_3}{\partial z_3} \\
&= ((-0.0079 \times -0.04) + (0.0329 \times 0.20)) \times 0 \\
&= 0
\end{aligned}
$$

(50)

Table 13: The ReLU network's per example prediction, error, and the sum of squared errors after training has converged to an $SSE < 0.0001$.

	$\mathbf{d}_1$	$\mathbf{d}_2$	$\mathbf{d}_3$	$\mathbf{d}_4$
Target	0.9400	0.1300	0.5700	0.3600
Prediction	0.9487	0.1328	0.5772	0.3679
Error	-0.0087	-0.0028	-0.0072	-0.0079
Error^2	0.00007569	0.00000784	0.00005184	0.00006241
SSE:				0.00009889

- After about 150 epochs, the rate of decrease of the SSE increases, but training also becomes a bit unstable with the network SSE sometimes increasing and decreasing dramatically. (See figure 21[121].)
- This type of instability can be indicative that the learning rate is too high.
- In fact, because the rectifier linear function is unbounded, it is often the case that smaller learning rates are necessary using ReLUs than logistic units.

- After about 150 epochs, the rate of decrease of the SSE increases, but training also becomes a bit unstable with the network SSE sometimes increasing and decreasing dramatically. (See figure 21[121].)
- This type of instability can be indicative that the learning rate is too high.
- In fact, because the rectifier linear function is unbounded, it is often the case that smaller learning rates are necessary using ReLUs than logistic units.
- Figure 22[122] plots the SSE of the ReLU network across the training epochs when we use a smaller learning rate, in this case $\alpha = 0.1$.
- The learning is more stable (i.e., smoother) and the network training actually converges on the stop criterion of $SSE < 0.0001$ in fewer epochs, 412 instead of 424.
- This shows how sensitive a deep NN training can be to a range of hyper-parameters, e.g. learning rate and activation function.

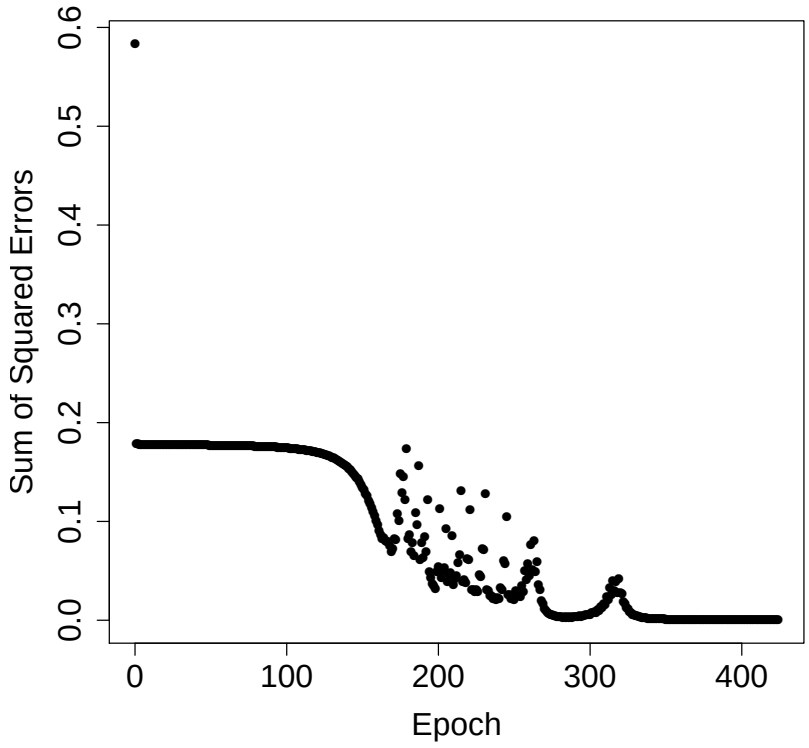


Figure 21: A plot showing how the sum of squared errors of the ReLU network changed during training when $\alpha = 0.2$.

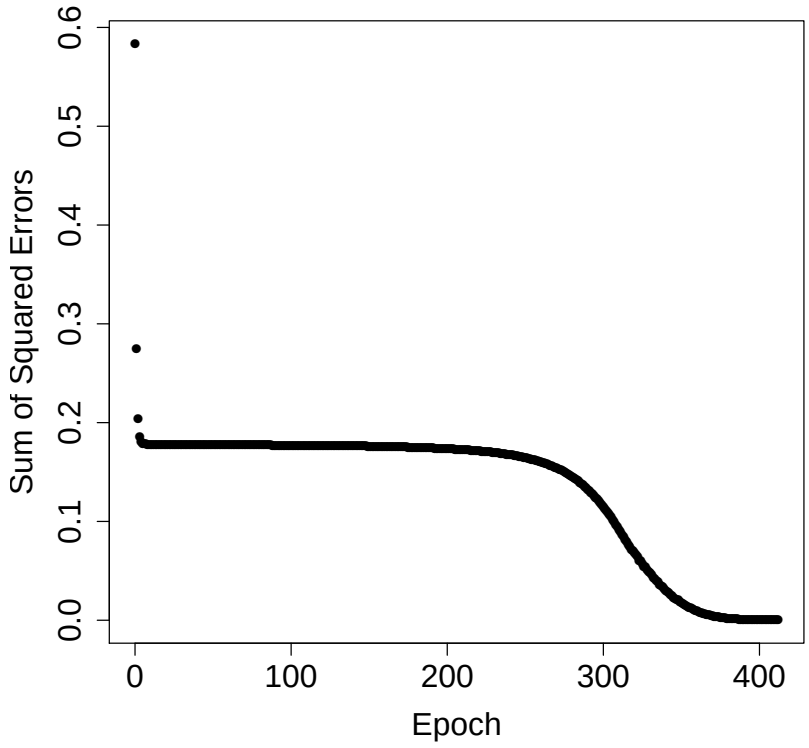


Figure 22: A plot showing how the sum of squared errors of the ReLU network changed during training when $\alpha = 0.1$.

- switching a network from a logistic activation function to a rectified linear activation function can speed up training.
- However, it also tends to make the representations learned by a network sparse, i.e. for any given input vector, only a subset of the neurons in the network will activate (i.e., $a_i > 0$).
- Consider a feedforward network that has been initialized with weights randomly sampled with uniform probability from a range such as $[-0.5, 0.5]$.
- In such a network, immediately after initialization, approximately half the hidden neurons in the network will have activations equal to zero.
- Using a sparse representation can reduce the energy consumed by a network.
- However, if the representations become too sparse, then the performance of the network may deteriorate.

Consider Neuron 4 in our example network.

- The fact that the ReLU activation function for Neuron 4 is saturated for all four examples means that Neuron 4 is essentially dead during the forward pass of the algorithm (it does not activate for any example).
- This reduces the representational capacity of the network.
- Because the derivative for Neuron 4 takes value 0, no matter how long we train the network, Neuron 4 will remain in this dead state.
- This dynamic of a ReLU being in a state where it is inactive for all (or nearly all) inputs and consequently it is never updated and so never becomes active is known as the **dying ReLU problem**.

Consider Neuron 4 in our example network.

- If too many neurons in a network are dead, then the network will not converge during training.
- if Neuron 8 is dead, the network will never converge on the training stop criterion, because no error gradients will be backpropagated to the earlier layers and so no training will occur.
- Similarly, if Neurons 6 and 7 are dead, then no error gradients will be backpropagated past this layer of neurons and the network is essentially reduced to a perceptron network composed of just Neuron 8.

- A simple heuristic that is sometimes used to try to avoid dead ReLUs is to initialize all the bias weights of a network to small positive values, such as 0.1, because this increases the likelihood that most of the neurons will initially be active for most of the training examples and so these neurons can learn from these examples
- This heuristic is not guaranteed to avoid dead ReLUs.

Another approach to avoiding dead ReLUs is to modify the rectified linear function so that it does not saturate for $z < 0$. One such variant is the Leaky ReLU which has a small (predefined) non-zero gradient when $z < 0$:

$$rectifier_{leaky}(z) = \begin{cases} z & \text{if } z > 0 \\ 0.01 \times z & otherwise \end{cases} \tag{51}$$

and its derivative is:

$$\frac{d}{dz} rectifier_{leaky}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0.01 & otherwise \end{cases} \tag{52}$$

Another approach to avoiding dead ReLUs is to modify the rectified linear function so that it does not saturate for $z < 0$. One such variant is the Leaky ReLU which has a small (predefined) non-zero gradient when $z < 0$:

$$rectifier_{leaky}(z) = \begin{cases} z & \text{if } z > 0 \\ 0.01 \times z & otherwise \end{cases} \tag{51}$$

and its derivative is:

$$\frac{d}{dz} rectifier_{leaky}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0.01 & otherwise \end{cases} \tag{52}$$

- Using Leaky ReLUs instead of ReLUs, we sacrifice the potential benefits in terms of energy efficiency of sparse representations for a gradient that may potentially be more robust during training.
- A leaky ReLU with large negative weight will still learn very slowly.

Another alternative to ReLU is Parametric ReLU (PReLU) for which the main distinction from the Leaky ReLU is that rather than using a fixed predefined gradient for $z \leq 0$, this gradient can be learned as a parameter for each neuron in the network. i.e. each neuron learns a separate gradient for its activation function for the region $z \leq 0$:

$$rectifier_{parametric}(z_i) = \begin{cases} z_i & \text{if } z_i > 0 \\ \lambda_i \times z_i & otherwise \end{cases} \tag{53}$$

Its derivative is:

$$\frac{d}{dz} rectifier_{parametric}(z_i) = \begin{cases} 1 & \text{if } z_i > 0 \\ \lambda_i & otherwise \end{cases} \tag{54}$$

The parameter λ is learned in tandem with the weights of the network.
As for the weights of the network, the λ parameter is initialized to a value and then iteratively updated as training progresses.

- Updating a λ is proportional to the error gradient of the network with respect to changes in that parameter:

$$\frac{\partial \mathcal{E}}{\partial \lambda_i} = \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial \lambda_i} \tag{55}$$

- The (1st term) rate of change of the network error with respect to changes in the activation function is calculated as for the network weights.
- The second term in the product $\frac{\partial a_i}{\partial \lambda_i}$ is the activation function gradient with respect to changes in λ_i. It is given by:

$$\frac{\partial a_i}{\partial \lambda_i} = \begin{cases} 0 & \text{if } z_i > 0 \\ z_i & otherwise \end{cases} \tag{56}$$

Once $\frac{\partial a_i}{\partial \lambda_i}$ is calculated, λ_i is updated using the following rule:

$$\lambda_i \leftarrow \lambda_i - \alpha \times \frac{\partial \mathcal{E}}{\partial \lambda_i} \tag{57}$$

- So far we have been initializing the bias terms and weights in our worked examples by sampling from a uniform distribution with a range of [-0.5, 0.5].
- Main advantage: simplicity of the approach.

- So far we have been initializing the bias terms and weights in our worked examples by sampling from a uniform distribution with a range of [-0.5, 0.5].
- Main advantage: simplicity of the approach.
- Some of the problems that arise when weights are set naively: vanishing gradients and dead neurons.
- If we wish to train a deep network, we want the behavior of the network, in terms of the variance of the layer's z values, activations, and error gradients, to be similar across all the layers of the network.
- Reason: being able to add more layers to the network.
- However, naively initializing weights $\implies$ unstable behavior within the dynamics of a network during training $\implies$ saturated activation functions (as a consequence of z values becoming too large or small) or unstable error gradients.

- First way a naive weight initialization can result in instability during training: the weights on the connections into a neuron are too large. $\implies$ the z value for the neuron will be large $\implies$ activation function for the neuron becoming saturated (large positive or negative z values).
- So derivative is zero $\implies$ adjusting the weights using small increments that are scaled by the activation function derivative. $\implies$ neurons with saturated activation functions can get stuck: their weights never change substantially.
- This is avoided by initializing the weights to be close to 0.

- Second way that a naive weight initialization can lead to instability during training is that very small or large weights can result in unstable gradients.
- The error term $\frac{\partial \mathcal{E}}{\partial a_k}$ is multiplied by two weights: $w_{j,i}$ and $w_{k,j}$.
  1. If both are > 1,
     - The error gradient term will get larger each time it is multiplied (known as **exploding gradients problem**)
     - Error gradient (the derivative) too large $\implies$ weights will be updated by a large amount $\implies$ changes in the output of a neuron, from one iteration to the next, will be so large that the training will become unstable.
  2. If both these weights are < 1,
     - The error gradient term will get smaller each time it is multiplied (**Vanishing gradients problem**).
     - Weights are very small (too close to 0) $\implies$ error gradient will tend to vanish $\implies$ the weight updates will be so small $\implies$ training the network will take too long.

Exploding and vanishing gradients can be understood as examples of the more general challenge of **unstable gradients**.

$$\delta_i = \underbrace{w_{j,i} \times w_{k,j} \times}_{\substack{\text{extreme weights} \\ \rightarrow \\ \text{unstable gradients}}} \frac{\partial \mathcal{E}}{\partial a_k} \underbrace{\times \frac{\partial a_k}{\partial z_k} \times \frac{\partial a_j}{\partial z_j} \times \frac{\partial a_i}{\partial z_i}}_{\substack{\text{extreme weights} \\ \rightarrow \\ \text{saturated activations} \\ \rightarrow \\ \text{vanishing gradients}}} \tag{58}$$

- Third way in which naive weights initialization can cause unstable gradients: the variance of the output of a weighted sum is a function of three factors: the number of inputs to the weighted sum, the variance of the inputs, and the variance of the weights.
- Consequently, if the relationship between the number of inputs to a weighted sum and the variance of the weights is incorrect, then the result of a weighted sum can have either a larger or a smaller variance than the variance of its inputs.
- This property of weighted sum calculations can result in unstable dynamics in both the forward and backward pass.

Example to show how adjusting the sample distribution variance used to initialize the network weights affects its dynamics during training:

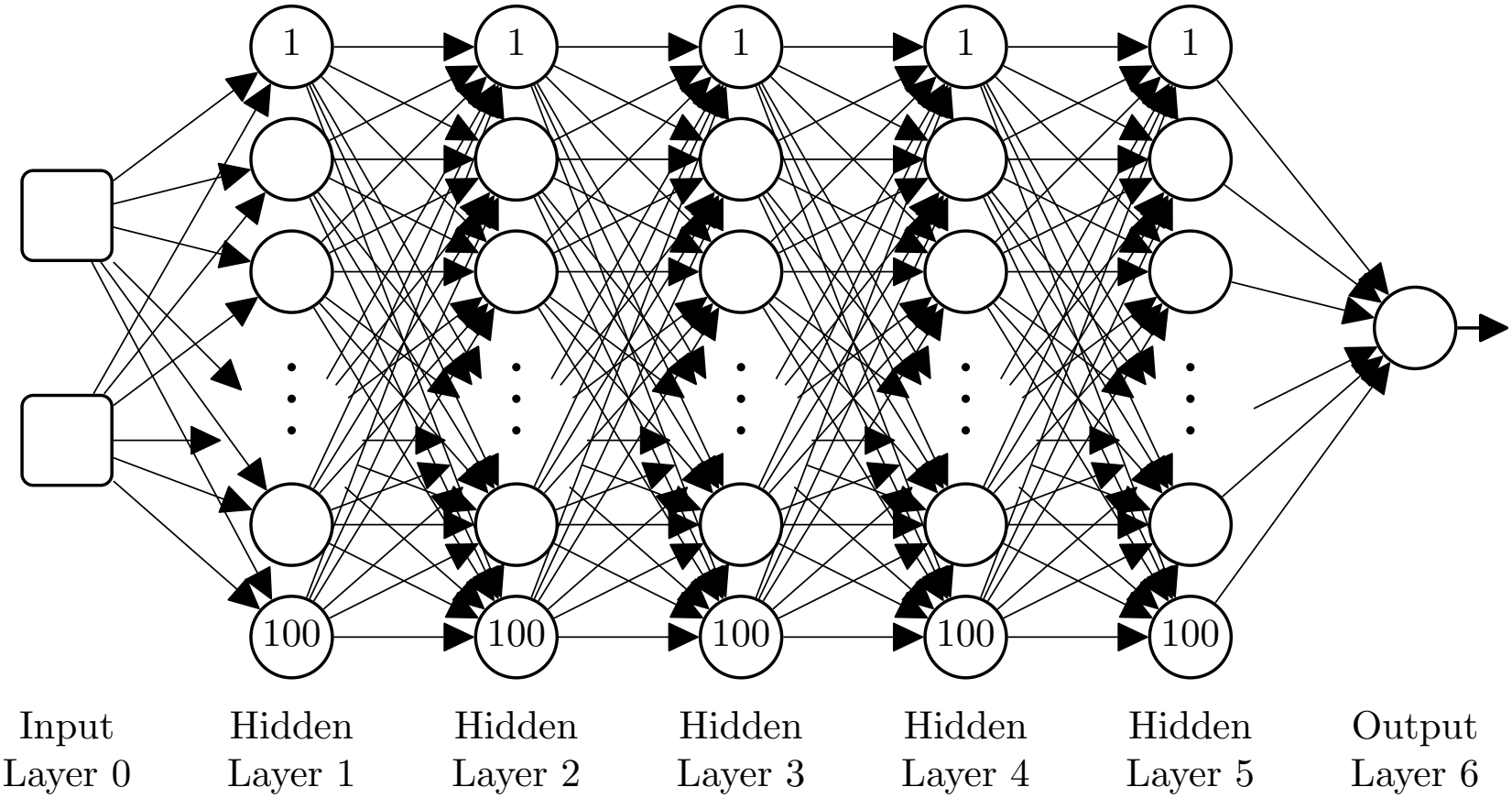


Figure 23: NN architecture used in the weight initialization experiments. (Neurons use a Linear activation function: $a_i = z_i$.)

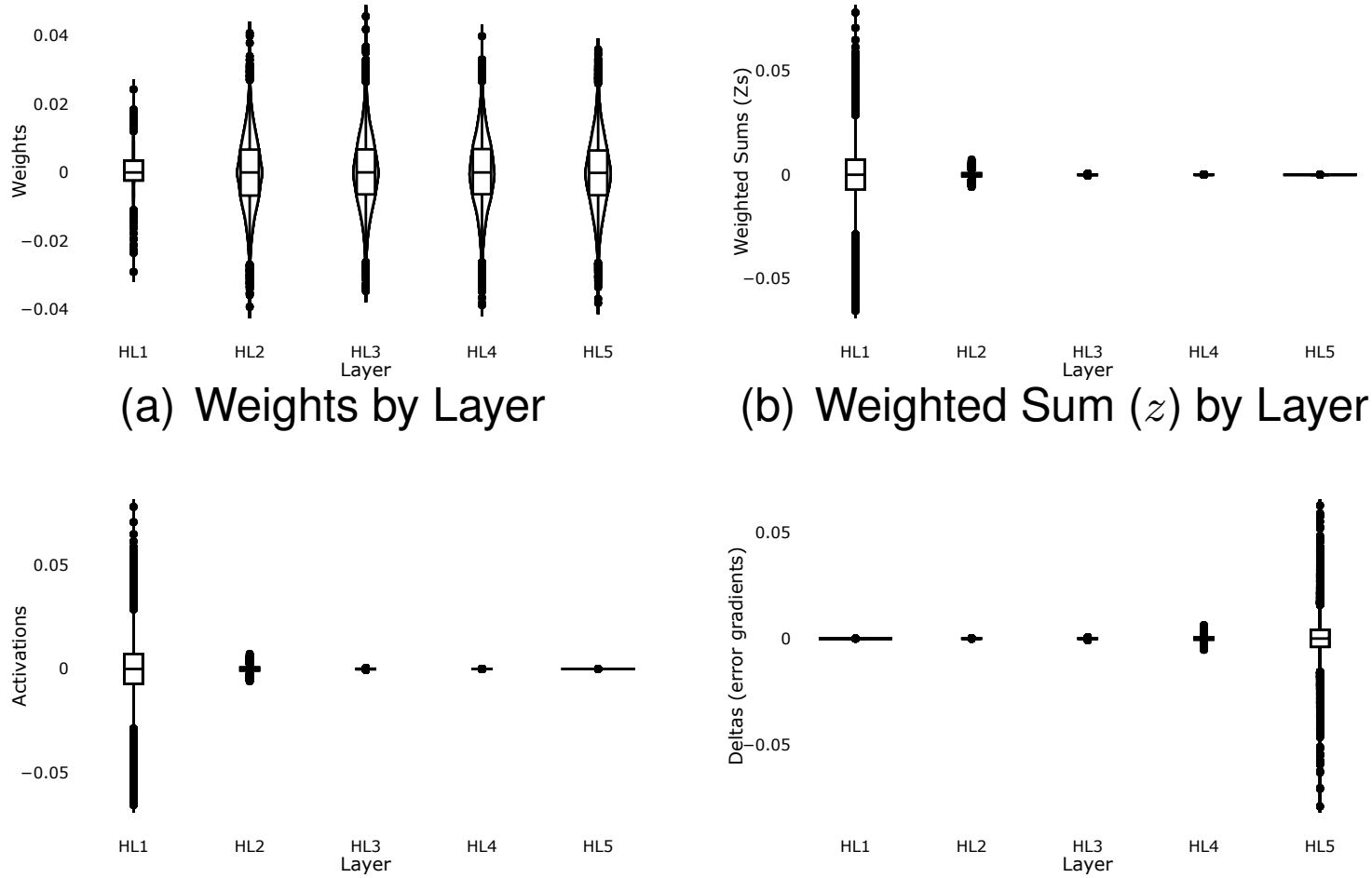
- In the previous network, the neurons use a linear activation function $a_i = z_i$.
- The derivative of this activation function with respect to z is always 1.
- Consequently, the gradients in this network will not be affected by saturated activation functions.
- We also increase the size of the training data to a sample of 100 examples, and we standardize all features to have μ of 0 and a σ of 1.

- Question: how will we initialize the weights of the previous network?

- Question: how will we initialize the weights of the previous network?
- Most weight initialization processes are based on heuristics that try to ensure that the weights are neither too big nor too small.
- Typically, bias terms are initialized to 0. But, in some instances we may wish to set bias terms to non-zero values. (e.g. ReLUs saturate when $z < 0$, and so to avoid dead ReLUs, the heuristic of initializing the bias terms for ReLU units to a small positive number (such as 0.1) is sometimes used.
- We will assume here that biases are initialized to 0.

- Most frequent heuristic used to initialize the weights: randomly sampling values from a normal or uniform distribution with a mean of 0. (e.g. μ of 0 and a σ of 0.01).
- When weights are initialized this way, we can control the initial scale of the weights by controlling the variance of the normal distribution. (See Figure 24[143])

- Most frequent heuristic used to initialize the weights: randomly sampling values from a normal or uniform distribution with a mean of 0. (e.g. μ of 0 and a σ of 0.01).
- When weights are initialized this way, we can control the initial scale of the weights by controlling the variance of the normal distribution. (See Figure 24[143])
- We can try to avoid this problem by making the network weights larger.
- Figure 25[144] shows the impact of increasing the standard deviation of the distribution from which we sample, in this instance a normal distribution with a μ of 0 and a σ of 0.2.



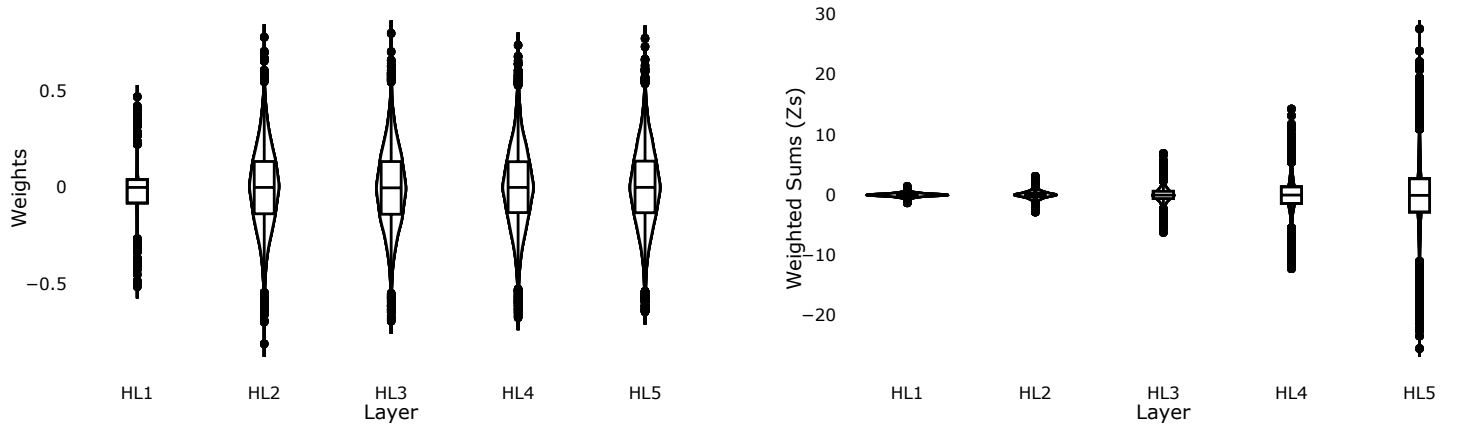
(a) Weights by Layer

(b) Weighted Sum (z) by Layer

(c) Activations by Layer

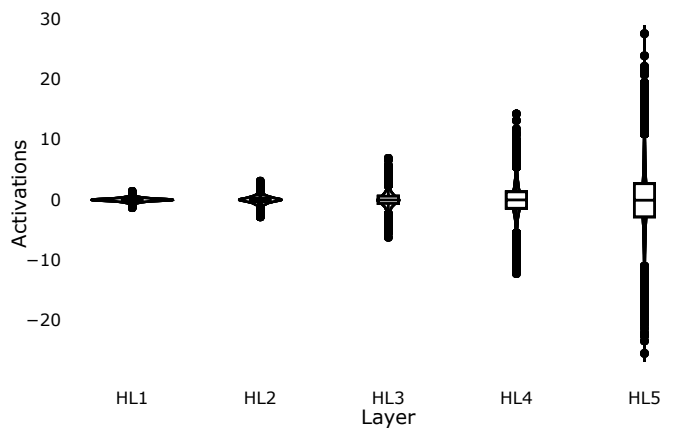
(d) δs by Layer

Figure 24: The internal dynamics of the network in Figure 23[137] during the first training iteration when the weights were initialized using a normal distribution with $\mu{=}0.0, \sigma{=}0.01$.

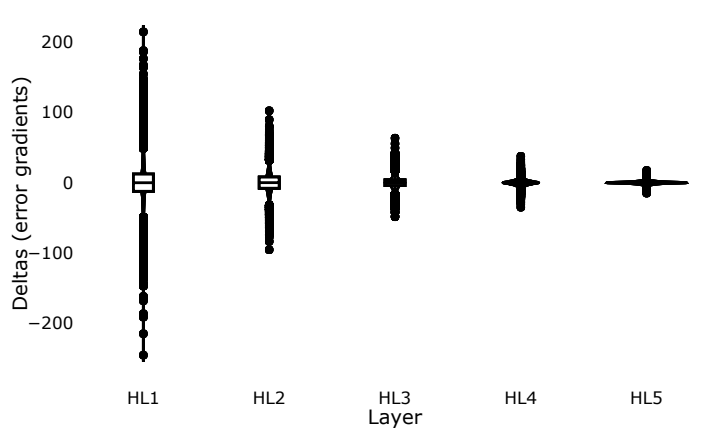


(a) Weights by Layer

(b) Weighted Sum (z) by Layer



(c) Activations by Layer



(d) δs by Layer

Figure 25: The internal dynamics of the network in Figure 23[137] during the first training iteration when the weights were initialized using a normal distribution with $\mu = 0.0$ and $\sigma = 0.2$.

- The previous Figure shows that the network now has an **exploding gradient** dynamic during backpropagation, with the variance of δ values rapidly increasing as they are backpropagated through the network.
- The exploding gradients is caused by a similar process to the vanishing z values. The connection between these three processes is that they all involve a weighted sum.

$$z = (w_1 \times d_1) + (w_2 \times d_2) + \cdots + (w_{n_{in}} \times d_{n_{in}}) \tag{59}$$

The starting point to explain the relationship between weighted sum calculations and vanishing and exploding z and δ values is the **Bienaymé formula** from statistics, which states that the variance of the sum of uncorrelated random variables is the sum of their variances:

$$var\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} var\left(X_i\right) \tag{60}$$

Assuming that all the weights and the inputs are independent and identically distributed, then the products in the weighted sum can be considered uncorrelated and we can state the variance of z as follows:

$$\begin{aligned} var(z) &= var\left((w_1 \times d_1) + (w_2 \times d_2) + \ldots (w_{n_{in}} \times d_{n_{in}})\right) \\ &= \sum_{i=1}^{n_{in}} var(w_i \times d_i) \end{aligned} \tag{61}$$

Assuming that each weight w_i is independent of the corresponding input d_i, then the variance of each of these products is:

$$var(w \times d) = [E(\mathbf{W})]^2\, var(\mathbf{d}) + [E(\mathbf{d})]^2\, var(\mathbf{W}) + var(\mathbf{W})\, var(\mathbf{d}) \tag{62}$$

$$var(w \times d) = var(\mathbf{W}) var(\mathbf{d}) \tag{63}$$

Since we have sampled our weights from a distribution with mean 0, then $E(W) = 0$, and if the inputs have been standardized, then $E(\mathbf{d}) = 0$, and so the Equation simplifies to:

$$var(z) = \sum_{i=1}^{n_{in}} var(w_i \times d_i) = n_{in}\ var(\mathbf{W})\ var(\mathbf{d}) \tag{64}$$

- The previous Equation states that the variance of z is equal to the variance of the inputs ($var(\mathbf{d})$) scaled by $n_{in}\ var(\mathbf{W})$.
- So the variance of z is dependent on the number n_{in} of inputs the neuron receives. (in general, the larger the number of inputs, the larger the variance).
- It also says that the scaling of the variance of z is dependent on the product $n_{in}var(\mathbf{W})$.
- Consequently, we can counteract this scaling by the number of inputs by setting $var(\mathbf{W}) = \frac{1}{n_{in}}$; i.e. setting the variance of the distribution of the neuron weights to $\frac{1}{n_{in}}$.
- When $var(\mathbf{W}) = \frac{1}{n_{in}}$ then $\frac{1}{n_{in}}var(\mathbf{W}) = 1$ and the variance of z for the neuron is solely dependent on the variance of the inputs, which if standardized will have a variance of 1.
- For fully connected network: n_{in} is the same for all the neurons in a layer, so we can set the variance of the distribution of the neuron weights on a layer-by-layer basis.

The weights in the network were sampled from a normal distribution with $\mu = 0.0$ and $\sigma = 0.01$. So the weights in each layer have a variance of $var(\mathbf{W}) = \sigma^2 = 0.01^2 = 0.0001$. So the variance of the z values across the neurons in layer $HL1$ is:

$$\begin{aligned} var(Z^{(HL1)}) &= n_{in}^{(HL1)} \times var(\mathbf{W}^{(HL1)}) \times var(\mathbf{d}^{(HL1)}) \\ &= 2 \times 0.0001 \times 1 \\ &= 0.0002 \end{aligned} \tag{65}$$

- note that a variance of $\sigma^2 = 0.0002$ is equivalent to a standard deviation of $\sigma \approx 0.014$. Since a linear activation function is used, $var(Z(HL1))$ is also the variance of the activations propagated forward to the next hidden layer, hence the variance of the inputs to the next layer: $var(Z(HL1)) = var(A(HL1)) = var(\mathbf{d}(HL2))$.
- For $HL2$, we know that $n_{in}^{(HL2)} = 100$ and $var(\mathbf{W}^{(HL2)}) = 0.0001$, so the variance of z can be calculated across the neurons in layer $HL2$ as follows:

- Note that a variance of $\sigma^2 = 0.0002$ is equivalent to a standard deviation of $\sigma \approx 0.014$.
- Since a linear activation function is used, $var(Z(HL1))$ is also the variance of the activations propagated forward to the next hidden layer, hence the variance of the inputs to the next layer:
  $var(Z(HL1)) = var(A(HL1)) = var(\mathbf{d}(HL2))$.
- For $HL2$, we know that $n_{in}^{(}HL2) = 100$ and $var(\mathbf{W}^{(}HL2)) = 0.0001$, so the variance of z can be calculated across the neurons in layer $HL2$ as follows:

$$\begin{aligned} var(Z^{(HL2)}) &= n_{in}^{(HL2)} \times var(\mathbf{W}^{(HL2)}) \times var(\mathbf{d}^{(HL2)}) \\ &= 100 \times 0.0001 \times 0.0002 \\ &= 0.000002 \end{aligned} \tag{66}$$

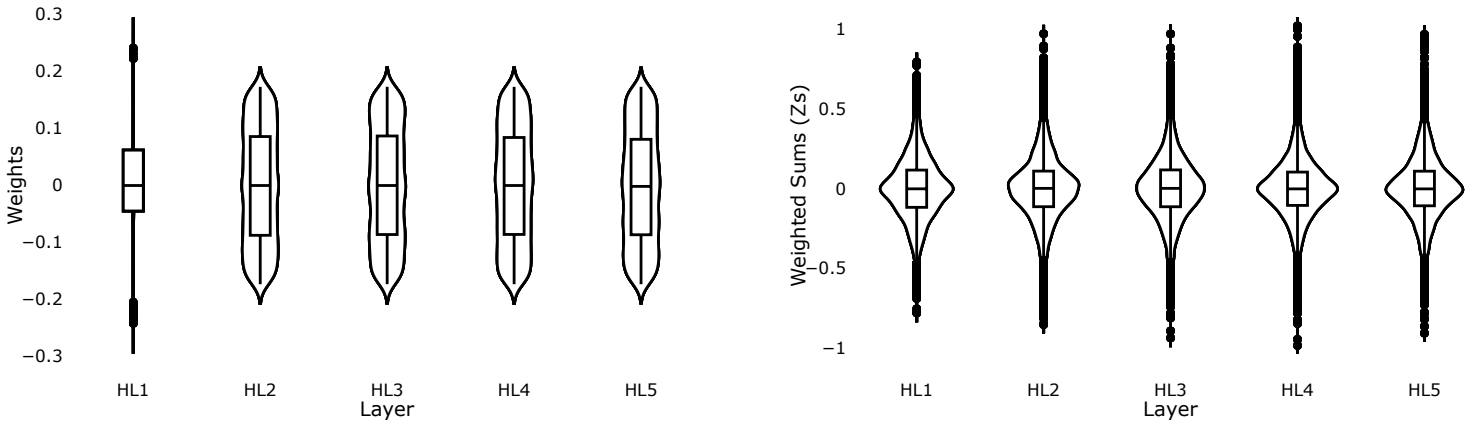
- Recall that in order to train a deep network, it is important to keep the internal behavior of the network (i.e., the variance of the z values, activations, and δs) similar across all the layers during training, because doing so allows us to add more layers to the network while avoiding saturated units (by avoiding extreme z values) and exploding or vanishing δs.
- We also saw that the variance of output of a weighted sum is dependent on the number of inputs to the weighted sum (be it n_{in} during forward propagation or n_{out} during backward propagation).
- Different weight initialization regimes have been developed and vary depending on whether they take both n_{in} and n_{out} into account and the activation functions with which they work best.

- One of the best known of these weight initialization regimes is called **Xavier initialization**.
- The original version of Xavier initialization considered both the forward activation through the network and the backward propagation of gradients, and so the calculation of the variance of the distribution from which the weights for a layer are sampled takes both n_{in} and n_{out} into account

$$var(\mathbf{W}^{(k)}) = \frac{2}{n_{in}^{(k)} + n_{out}^{(k)}} \tag{67}$$

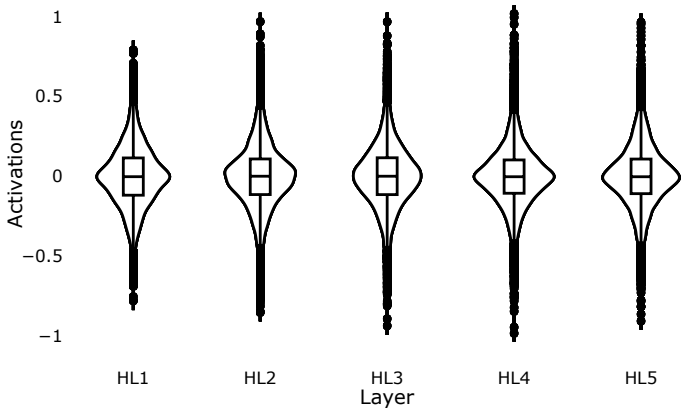
Often in practice, with a fully connected feedforward network, a simpler variant of Xavier initialization is used that just considers $n_{in}^{(k)}$:

$$var(\mathbf{W}^{(k)}) = \frac{1}{n_{in}^{(k)}} \tag{68}$$

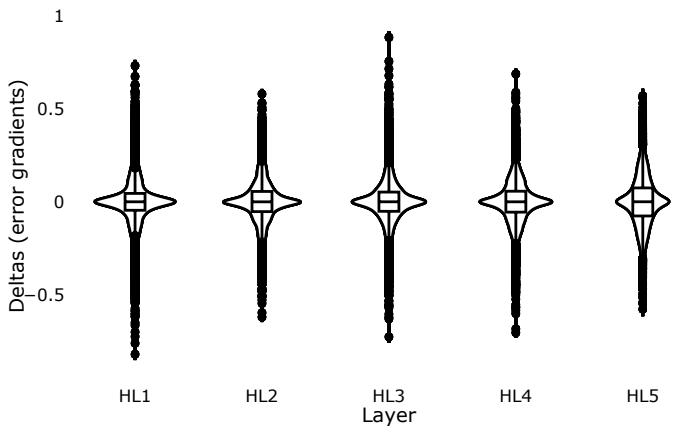


(a) Weights by Layer

(b) Weighted Sum (z) by Layer



(c) Activations by Layer



(d) δs by Layer

Figure 26: The internal dynamics of the network in Figure 23[137] during the first training iteration when the weights were initialized using Xavier initialization.

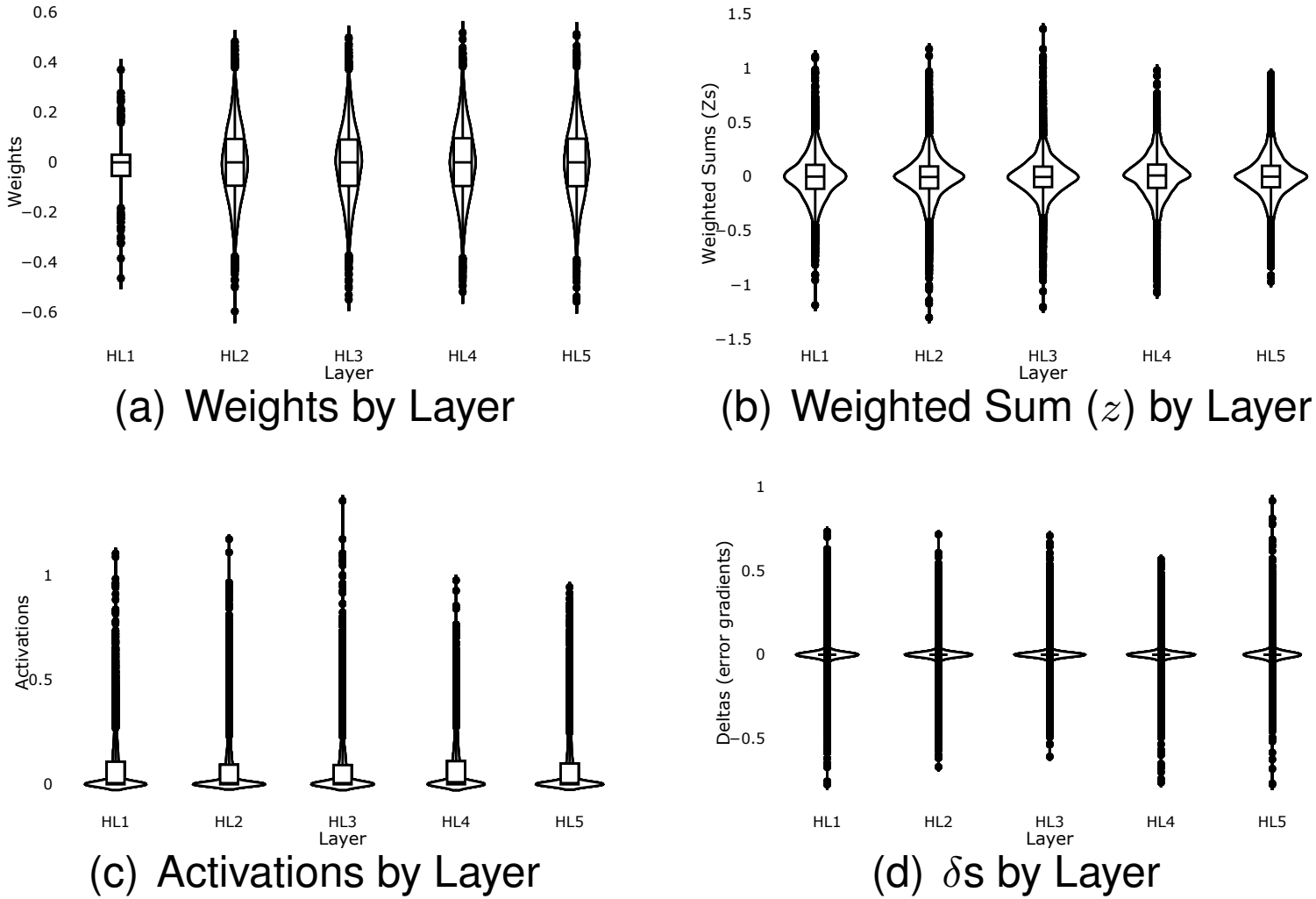
- Xavier initialization has empirically been shown to often lead to faster training and is one of the most popular weight initialization approaches in deep learning.
- It is generally used when networks use *logistic* or *tanh* activation functions.
- However, a modified version of this weight initialization heuristic is recommended when the network uses ReLUs.
- This weight initialization heuristic is known as **He initialization** (or sometimes called **Kaiming initialization**)

$$var(\mathbf{W}^{(k)}) = \frac{2}{n_{in}^{(k)}} \tag{69}$$

Sometimes a blend of **Xavier** and **He initialization** is used: Xavier for the weights in the first layer, because the rectified function has not been applied to the inputs, and then He initialization could then be used for the later layers in the network.

Example: for a fully connected three-layer ReLU network with 100 inputs, 80 neurons in the first hidden layer, 50 neurons in the second hidden layer, and 5 neurons in the output layer, the weight matrix for each layer would be initialized as follows:

$$
\begin{aligned}
W^{(1)} &\sim \mathcal{N}\left(0, \sqrt{\frac{1}{100}}\right) \\
W^{(2)} &\sim \mathcal{N}\left(0, \sqrt{\frac{2}{80}}\right) \\
W^{(3)} &\sim \mathcal{N}\left(0, \sqrt{\frac{2}{50}}\right)
\end{aligned}
\tag{70}
$$



(a) Weights by Layer

(b) Weighted Sum (z) by Layer

(c) Activations by Layer

(d) δs by Layer

Figure 27: The internal dynamics of the network in Figure 23[137], using ReLUs, during the first training iteration when the weights were initialized using He initialization.

We have focused so far on regression problems. To create a neural network that can predict a multi-level categorical feature, we make three adjustments:

1. represent the target feature using **one-hot encoding**;
2. change the output layer of the network to be a **softmax layer**; and
3. change the error (or loss) function we use for training to be the **cross-entropy** function.

Table 14: The *range-normalized* hourly samples of ambient factors and full load electrical power output of a combined cycle power plant, rounded to two decimal places, and with the (binned) target feature represented using one-hot encoding ($low \leq 0.33, medium \leq 0.66, high > 0.66$).

ID	Ambient Temperature °C	Relative Humidity %	Electrical Output *low*	*medium*	*high*
1	0.04	0.81	0	0	1
2	0.84	0.58	1	0	0
3	0.50	0.07	0	1	0
4	0.53	1.00	0	1	0

In a softmax output layer there is a single neuron for each level of the target feature.
The activation function used by the neurons in a softmax layer is the softmax function; for an output layer with m neurons, the softmax activation function is:

$$\varphi_{sm}\left(z_i\right) = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_m}} \tag{71}$$

In a softmax output layer there is a single neuron for each level of the target feature.
The activation function used by the neurons in a softmax layer is the softmax function; for an output layer with m neurons, the softmax activation function is:

$$\varphi_{sm}\left(z_i\right) = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_m}} \tag{71}$$

- The sum of the softmax layer neurons activations is 1.
- With softmax, the activation of each neuron is dependent on the size of its z value relative to the z values of the other output layer neurons.
- Softmax returns a strictly positive value for every neuron.
- In the context of a softmax layer, the non-normalized z values are often referred to as ***logits***.

Table 15: The calculation of the softmax activation function φ_{sm} over a vector of three logits $\mathbf{l}$.

	$\mathbf{l}_0$	$\mathbf{l}_1$	$\mathbf{l}_2$
$\mathbf{l}$	1.5	-0.9	0.6
$e^{\mathbf{l}_i}$	4.48168907	0.40656966	1.8221188
$\sum_i e^{\mathbf{l}_i}$			6.71037753
$\varphi_{sm}(\mathbf{l}_i)$	0.667874356	0.060588195	0.27153745

- Softmax returns a normalized set of positive values across the layer $\Longrightarrow$ the activation of each neuron in the layer can be interpreted as a probability
- With softmax layer, each neuron is trained to predict the probability of one of the levels of the categorical target feature.
- Final prediction: feature level whose neuron predicts the highest probability.

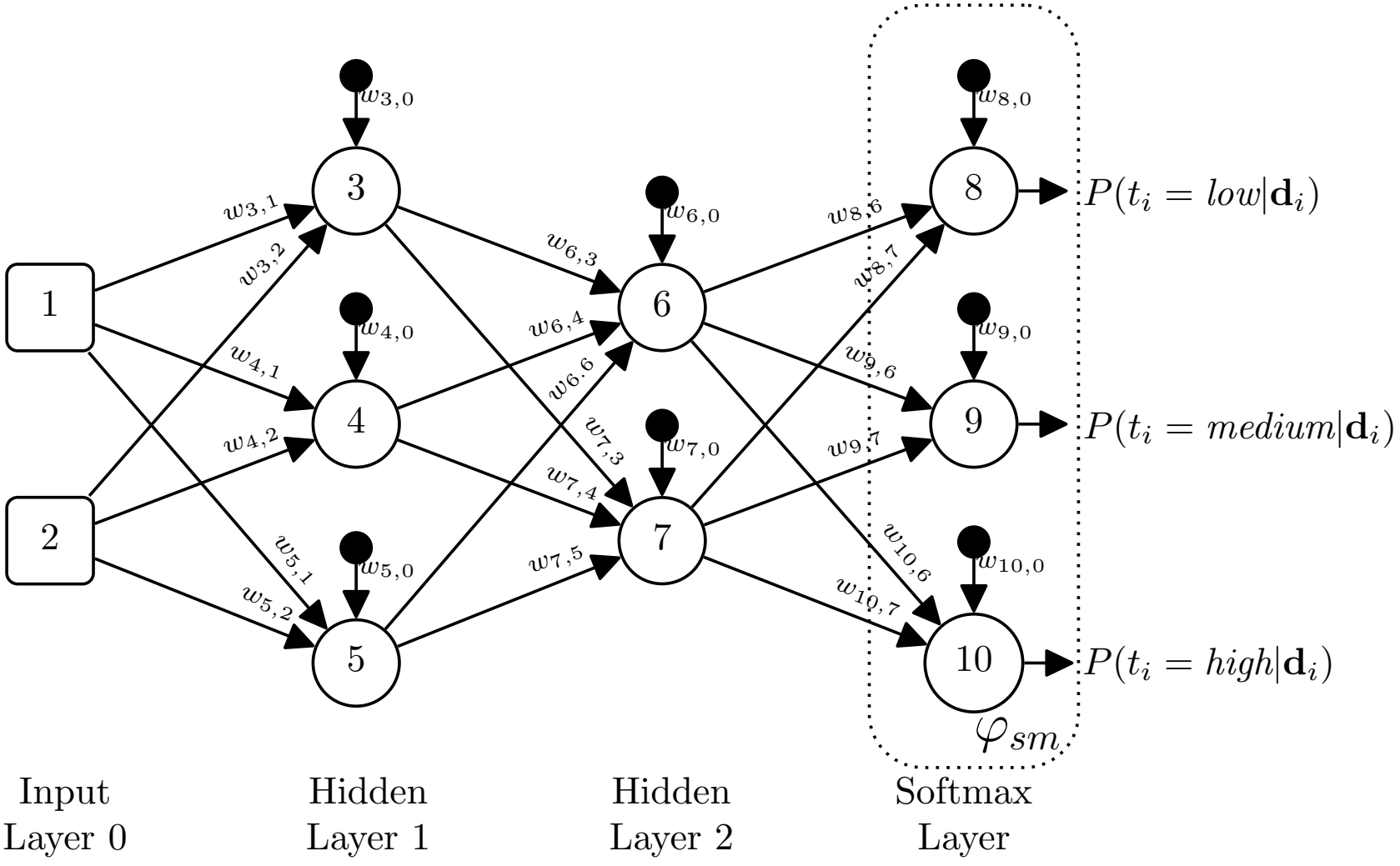


Figure 28: A schematic of a feedforward artificial neural network with a three-neuron softmax output layer.

- The cross-entropy error (or loss) function is generally used when the network output can be interpreted as a probability distribution over a set of exclusive categories.
- For a model predicting a distribution over a set of categories, the cross-entropy loss function is defined as:

$$L_{CE}\left(\mathbf{t}, \hat{\mathbf{P}}\right) = -\sum_{j} \mathbf{t}_j \ln\left(\hat{\mathbf{P}}_j\right) \tag{72}$$

where t is the target feature represented using one-hot encoding; $\hat{\mathbf{P}}$ is the distribution over the categories that the model has predicted.

- The cross-entropy error (or loss) function is generally used when the network output can be interpreted as a probability distribution over a set of exclusive categories.
- For a model predicting a distribution over a set of categories, the cross-entropy loss function is defined as:

$$L_{CE}\left(\mathbf{t}, \hat{\mathbf{P}}\right) = -\sum_{j} \mathbf{t}_j \ln\left(\hat{\mathbf{P}}_j\right) \tag{72}$$

where t is the target feature represented using one-hot encoding; $\hat{\mathbf{P}}$ is the distribution over the categories that the model has predicted.
Where the true distribution t is encoded as a one-hot vector, the cross-entropy loss function can be simplified to:

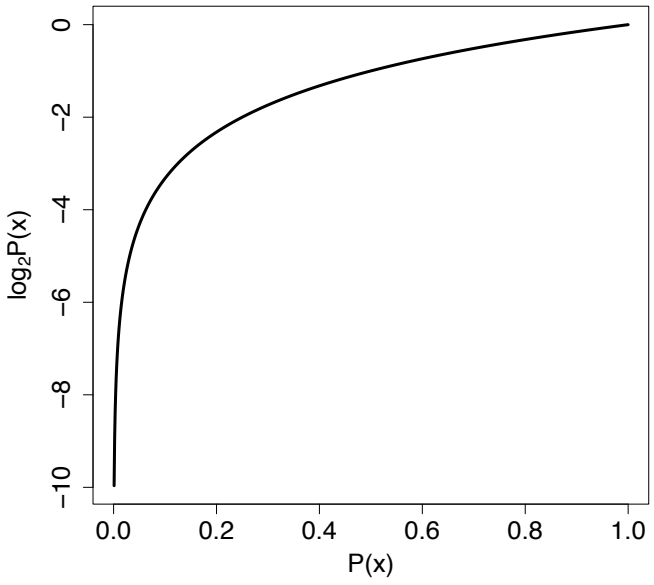
$$L_{CE}\left(\mathbf{t}, \hat{\mathbf{P}}\right) = -\ln\left(\hat{\mathbf{P}}_\star\right) \tag{73}$$

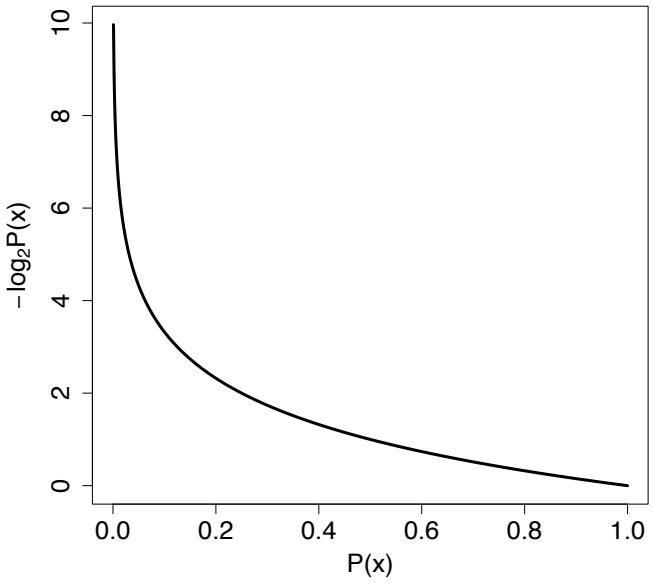
where $\hat{\mathbf{P}}_\star$: the predicted probability for the true category (i.e., the category encoded as a 1 in the one-hot encoded vector t).

Example: if the target distribution over a particular instance is $t[0, 1, 0]$ (i.e., the second category is the correct category), then the cross-entropy summation Equation 73[165] expands as follows:

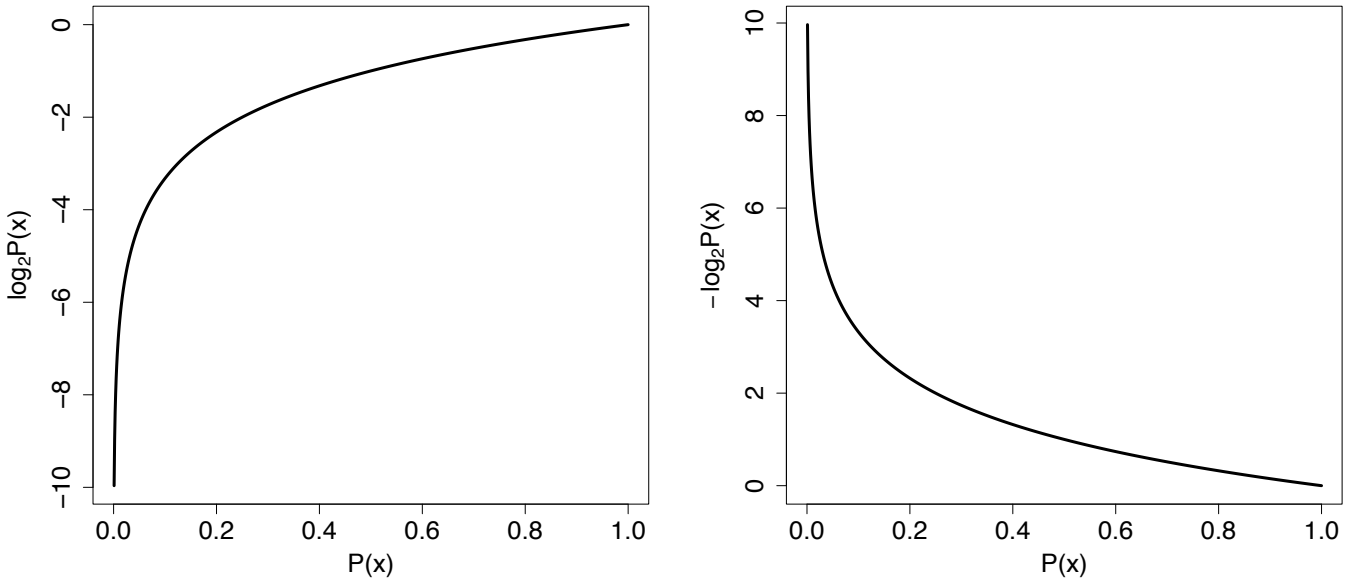
$$\begin{aligned} L_{CE}\left(\mathbf{t}, \hat{\mathbf{P}}\right) &= -\sum_{j} \mathbf{t}_j \ln\left(\hat{\mathbf{P}}_j\right) \\ &= -\left(\left(\mathbf{t}_0 \ln\left(\hat{\mathbf{P}}_0\right)\right) + \left(\mathbf{t}_1 \ln\left(\hat{\mathbf{P}}_1\right)\right) + \left(\mathbf{t}_2 \ln\left(\hat{\mathbf{P}}_2\right)\right)\right) \\ &= -\left(\left(0 \ln\left(\hat{\mathbf{P}}_0\right)\right) + \left(1 \ln\left(\hat{\mathbf{P}}_1\right)\right) + \left(0 \ln\left(\hat{\mathbf{P}}_2\right)\right)\right) \\ &= -1 \ln\left(\hat{\mathbf{P}}_1\right) \end{aligned} \tag{74}$$

Recall from Chapter 4 the graph illustrating how the value of a negative log to the base 2 of a probability changes across the range of probability values.





Recall from Chapter 4 the graph illustrating how the value of a negative log to the base 2 of a probability changes across the range of probability values.



This is exactly the behavior we desire from a loss function: small values for correct predictions and large values for incorrect predictions.

We can thus calculate a loss for the network's predictions over a set of exclusive categories.

Backpropagation: the following steps through the definition of the δ_k for a neuron in a softmax output layer when a cross-entropy loss function is used:

$$\boldsymbol{\delta}_k = \frac{\partial \mathcal{E}}{\partial z_k} \tag{75}$$

$$= \frac{\partial L_{CE}\left(\mathbf{t}, \hat{\mathbf{P}}\right)}{\partial \mathbf{l}_k} \tag{76}$$

$$= \frac{\partial - \ln\left(\hat{\mathbf{P}}_\star\right)}{\partial \mathbf{l}_k} \tag{77}$$

$$= \frac{\partial - \ln\left(\hat{\mathbf{P}}_\star\right)}{\partial\left(\hat{\mathbf{P}}_\star\right)} \times \frac{\partial\left(\hat{\mathbf{P}}_\star\right)}{\partial \mathbf{l}_k} \tag{78}$$

$$\frac{d \ln x}{dx} = \frac{1}{x} \tag{79}$$

$$\frac{\partial - \ln\left(\hat{\mathbf{P}}_{\star}\right)}{\partial\left(\hat{\mathbf{P}}_{\star}\right)} = -\frac{1}{\hat{\mathbf{P}}_{\star}} \tag{80}$$

$$\frac{\partial\left(\hat{\mathbf{P}}_{\star}\right)}{\partial \mathbf{l}_k} = \begin{cases} \hat{\mathbf{P}}_{\star}\left(1 - \hat{\mathbf{P}}_k\right) & \text{if } k = \star \\ -\hat{\mathbf{P}}_{\star}\hat{\mathbf{P}}_k & otherwise \end{cases} \tag{81}$$

$$\boldsymbol{\delta}_k = \frac{\partial - \ln\left(\hat{\mathbf{P}}_\star\right)}{\partial\left(\hat{\mathbf{P}}_\star\right)} \times \frac{\partial\left(\hat{\mathbf{P}}_\star\right)}{\partial \mathbf{l}_k} \tag{82}$$

$$= -\frac{1}{\hat{\mathbf{P}}_\star} \times \frac{\partial\left(\hat{\mathbf{P}}_\star\right)}{\partial \mathbf{l}_k} \tag{83}$$

$$= -\frac{1}{\hat{\mathbf{P}}_\star} \times \begin{cases} \hat{\mathbf{P}}_\star\left(1 - \hat{\mathbf{P}}_k\right) & \text{if } k = \star \\ -\hat{\mathbf{P}}_\star \hat{\mathbf{P}}_k & otherwise \end{cases} \tag{84}$$

$$= \begin{cases} -\left(1 - \hat{\mathbf{P}}_k\right) & \text{if } k = \star \\ \hat{\mathbf{P}}_k & otherwise \end{cases} \tag{85}$$

$$\boldsymbol{\delta}_{k=\star} = -\left(1 - \hat{\mathbf{P}}_k\right) \tag{86}$$

$$\boldsymbol{\delta}_{k\neq\star} = \hat{\mathbf{P}}_k \tag{87}$$

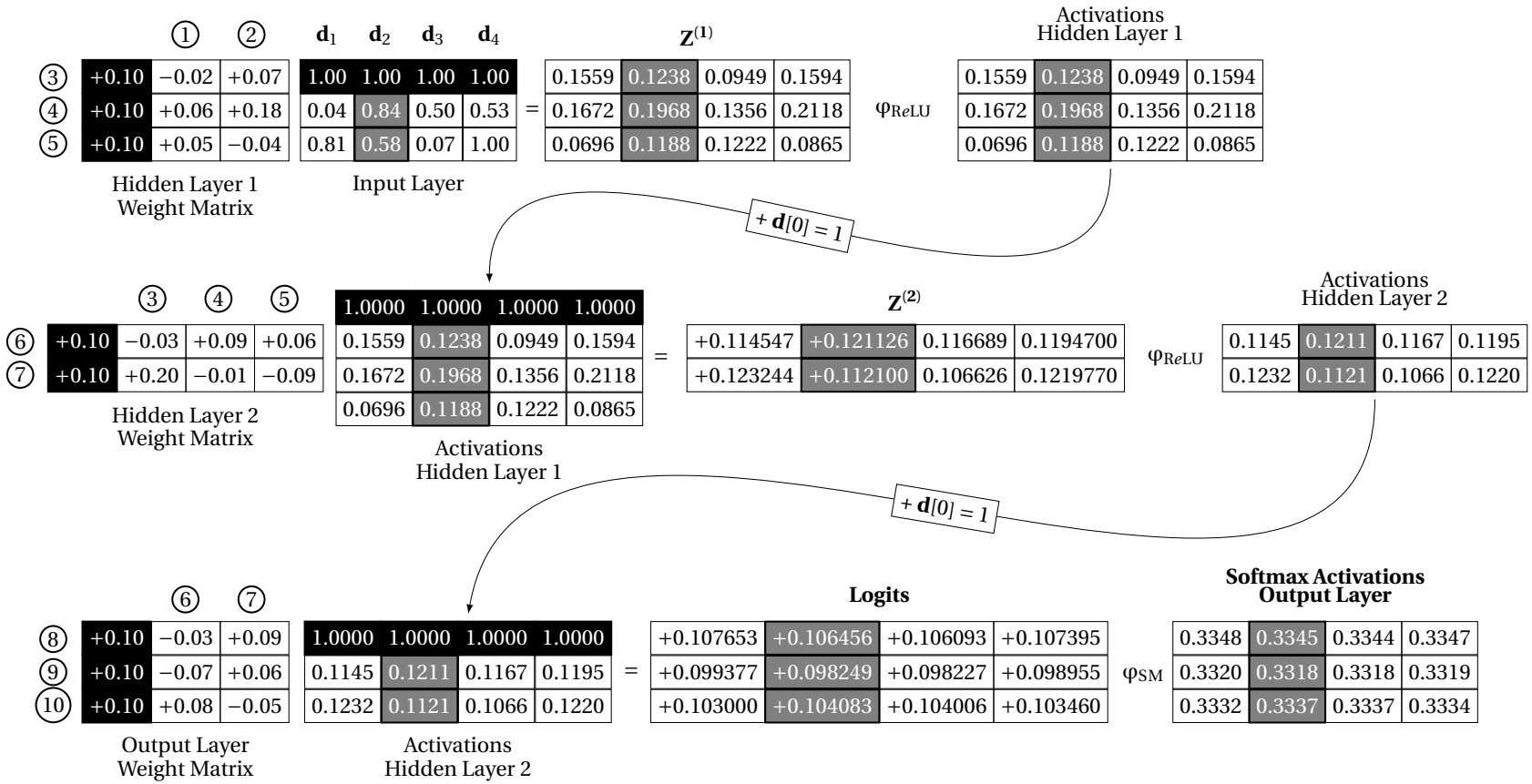


Figure 29: The forward pass of the mini-batch of examples listed in Table 14[160] through the network in Figure 28[164]. Neurons in hidden layers are ReLUs and output layer is a softmax layer.

Table 16: The calculation of the softmax activations for each of the neurons in the output layer for each example in the mini-batch, and the calculation of the δ for each neuron in the output layer for each example in the mini-batch.

	$\mathbf{d}_1$	$\mathbf{d}_2$	$\mathbf{d}_3$	$\mathbf{d}_4$
Per Neuron Per Example logits				
Neuron 8	0.107653	0.106456	0.106093	0.107395
Neuron 9	0.099377	0.098249	0.098227	0.098955
Neuron 10	0.103000	0.104083	0.1040060	0.103460
Per Neuron Per Example $e^{\mathbf{l}_i}$				
Neuron 8	1.113661238	1.112328983	1.111925281	1.11337395
Neuron 9	1.104482611	1.103237457	1.103213186	1.104016618
Neuron 10	1.108491409	1.109692556	1.109607113	1.109001432
$\sum_i e^{\mathbf{l}_i}$	3.326635258	3.325258996	3.324745579	3.326392
Per Neuron Per Example Softmax Activations				
Neuron 8	0.3348	0.3345	0.3344	0.3347
Neuron 9	0.3320	0.3318	0.3318	0.3319
Neuron 10	0.3332	0.3337	0.3337	0.3334
Per Neuron Target One-Hot Encodings				
Neuron 8	0	1	0	0
Neuron 9	0	0	1	1
Neuron 10	1	0	0	0
Per Neuron Per Example δs				
Neuron 8	0.3348	-0.6655	0.3344	0.3347
Neuron 9	0.3320	0.3318	-0.6682	-0.6681
Neuron 10	-0.6668	0.3337	0.3337	0.3334

$$
\begin{aligned}
\Delta w_{9,6} &= \sum_{j=1}^{4} \boldsymbol{\delta}_{9,j} \times a_{6,j} \\
&= (0.3320 \times 0.1145) + (0.3318 \times 0.1211) \\
&\quad + (-0.6682 \times 0.1167) + (-0.6681 \times 0.1195) \\
&= 0.038014 + 0.04018098 + -0.07797894 + -0.07983795 \\
&= -0.07962191 \qquad (88)
\end{aligned}
$$

$$\begin{aligned} w_{9,6} &= w_{9,6} - \alpha \times \Delta w_{9,6} \\ &= -0.07 - 0.01 \times -0.07962191 \\ &= -0.07 - (-0.000796219) \\ &= -0.069203781 \end{aligned} \tag{89}$$

Early Stopping and Dropout: Preventing Overfitting

- Deep learning models can have millions of parameters, and this complexity makes them **prone to overfitting**.
- Two of the most commonly used methods to avoid overfitting are **early stopping** and **dropout**.

- Deep learning models can have millions of parameters, and this complexity makes them **prone to overfitting**.
- Two of the most commonly used methods to avoid overfitting are **early stopping** and **dropout**.
- Early stopping main idea: identify the point during training when a model begins to overfit the training data (point when the error of the model on a validation dataset begins to increase).
- The early stopping algorithm uses the performance of the model on a validation dataset to determine when to stop training the model.
- To control early stopping, a **patience parameter** is commonly used: a predefined threshold (i.e., a hyper-parameter) specifying the number of times in a row the error on the validation set is allowed to get higher than the lowest recorded so far before we stop training.

Algorithm 1 The early stopping algorithm

Require: p the patience parameter
Require: $\mathcal{D}_\nu$ a validation set
bestValidationError $= \infty$
tmpValidationError $= 0$
$\theta =$ initial model parameters
$\theta^{best} = 0$
patienceCount = 0
while patienceCount $< p$ **do**
    $\theta =$ new model parameters after most recent weight update
    tmpValidationError = calculateValidationError(θ, $\mathcal{D}_\nu$)
    if bestValidationError $\geq$ tmpValidationError **then**
        bestValidationError = tmpValidationError
        $\theta^{best} = \theta$
        patienceCount = 0
    else
        patienceCount = patienceCount + 1
    end if
end while
return Best Model Parameters θ^{best}

- **Dropout:** For each loaded training example, a random set of neurons from the input and hidden layers are dropped (or deleted) from the network. (So smaller networks used.)
- Feed forward and backpropagation are performed as usual on this instance and the weights get updated.
- So the weights connected to neurons that are dropped for an example do not receive updates on that example.
- This process is repeated with each instance on a new reduced version of the network.
- Note that for a given example, a different set of neurons is dropped each time the example is presented to the network on different epochs.

- **Dropout:** For each loaded training example, a random set of neurons from the input and hidden layers are dropped (or deleted) from the network. (So smaller networks used.)
- Feed forward and backpropagation are performed as usual on this instance and the weights get updated.
- So the weights connected to neurons that are dropped for an example do not receive updates on that example.
- This process is repeated with each instance on a new reduced version of the network.
- Note that for a given example, a different set of neurons is dropped each time the example is presented to the network on different epochs.
- Once the model has been trained dropout is not used during inference; otherwise random noise would be introduced to the inference process.

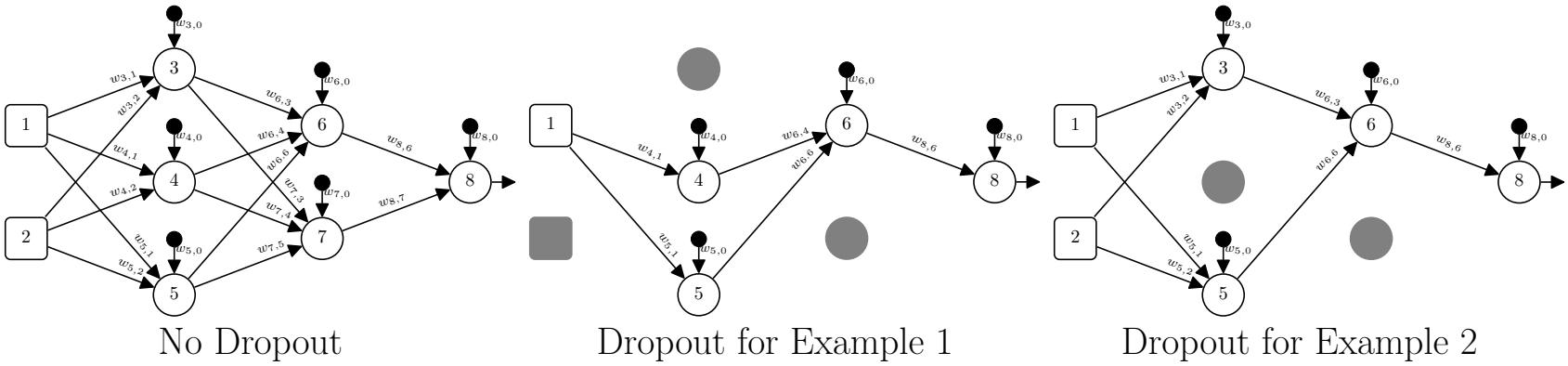


Figure 30: An illustration of how different small networks are generated for different training examples by applying dropout to the original large network. The gray nodes mark the neurons that have been dropped from the network for the training example.

- **inverted Dropout:** A neuron is dropped by multiplying its activation during the forward pass by zero.
- Also, during backpropagation no error gradients flow back through the dropped neurons; their δs are set to 0.
- Thus the weights on a dropped neuron won't receive any weight updates for that example.
- During the forward pass, for each input and hidden layer in the network, a vector $DropMask$ of 0 or 1 values is sampled from a **Bernoulli distribution** with probability ρ that a sampled value will be 1.
- Elementwise multiplication of the vector $a^{(l)}$ and $DropMask$ is performed yielding an updated activation vector $a^{(l)'}$.
- Each element in $a^{(l)'}$ is divided by the parameter ρ to scale up the non-zero activations so the weighted sum calculations in the next layer are of a similar magnitude to what they would have been if none of the activations had been set to 0.

Algorithm 2 Extensions to Backpropagation to Use Inverted Dropout

Require: ρ probability that a neuron in a layer will not be dropped

▷ Forward Pass

1: **for** each input or hidden layer l **do**
2: $\quad \mathbf{DropMask}^{(l)} = (m_1, \ldots, m_{size(l)}) \sim Bernoulli(\rho)$
3: $\quad \mathbf{a}^{(l)\prime} = \mathbf{a}^{(l)} \odot \mathbf{DropMax}^{(l)}$
4: $\quad \mathbf{a}^{(l)\prime\prime} = \frac{1}{\rho}\mathbf{a}^{(l)\prime}$
5: **end for**

▷ Backward Pass

6: **for** each layer l in backward pass **do**
7: $\quad \delta^{(l)} = \delta^{(l)} \odot \mathbf{DropMax}^{(l)}$
8: **end for**

Typical values for ρ are 0.8 for the input layer, and 0.5 for hidden layers,

- With dropout we are in effect training an ensemble of a very large number of smaller networks rather than training a single large model, and these smaller networks are less complex and so are less likely to overfit.
- The variation in the data presented to each subnetwork stops the model from memorizing the training data so it learns patterns that generalize over sets of features.
- More features contribute to the predictions made by the model $\implies$ the weight updates get spread out across more weights (more weights being involved in the prediction) $\implies$ the weight will in general remain smaller $\implies$ a model?s predictions are kept relatively stable with respect to small changes in the input.
- Models whose output changes drastically in response to small changes in the input are likely overfitting the data, because a small amount of noise in the input data can have a large effect on the outputs generated by the model.

- The approach of avoiding overfitting by modifying the learning algorithm in order to generate models that are stable with respect to changes in the input is generally known as regularization.
- Dropout can be understood as a regularization technique that improves the stability of the resulting model.
- Early stopping can also be understood as a regularization technique because it limits the number of updates to the weights in a model and by so doing keeps individual weights from getting too large.

Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) are primarily tailored to process grid like data, such as image data.
- The CNN architecture was originally applied to handwritten digit recognition.
- The MNIST dataset contains 60,000 training and 10,000 test images of handwritten digits from about 250 writers.
- Each image is labeled with the digit it contains, and the prediction task is to return the correct label for each image.
- Each image is grayscale, represented as a grid of 28 by 28 integers $\in [0, 255]$ where 0 indicates a white pixel, 255 indicates a black pixel, and numbers in between indicate shades of gray.

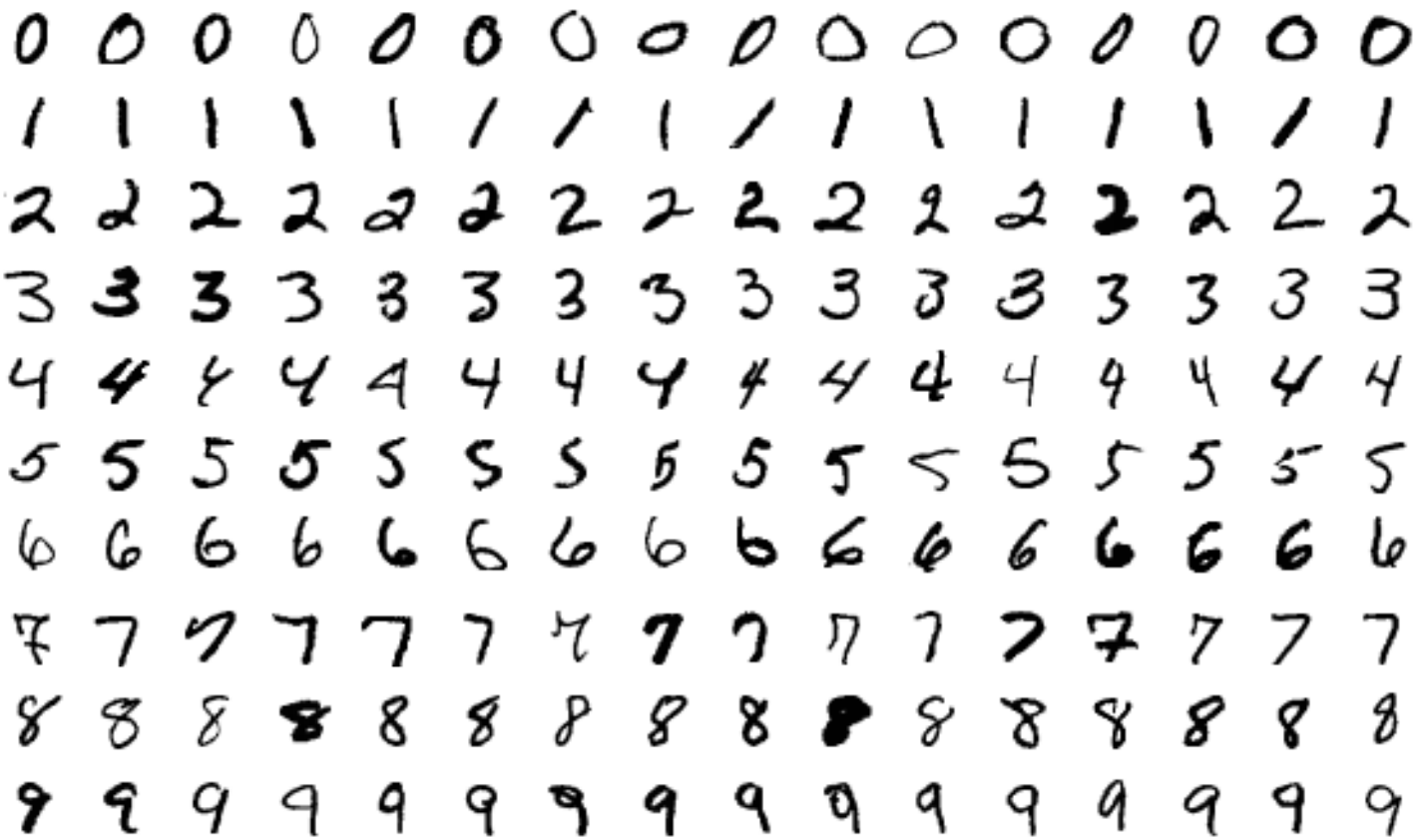


Figure 31: Samples of the handwritten digit images from the MNIST dataset. Image attribution: Josef Steppan, used here under the Creative Commons Attribution-Share Alike 4.0 International license https://creativecommons.org/licenses/by-sa/4.0) and was sourced via Wikimedia Commons

For the examples in this section, the following 6-by-6 matrix grayscale encoding of a 4 is used:

$$\begin{bmatrix} 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & \mathbf{255} & 000 & 000 & 000 & 000 \\ 000 & \mathbf{255} & 000 & \mathbf{255} & 000 & 000 \\ 000 & \mathbf{255} & \mathbf{255} & \mathbf{255} & \mathbf{255} & 000 \\ 000 & 000 & 000 & \mathbf{255} & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 \end{bmatrix} \tag{90}$$

CNNs have three distinctive characteristics:

1. local receptive fields;
2. weight sharing; and
3. sub-sampling (pooling).

- In the early 1960s, experiements on cats showed that each of the cats brain neurons (in each group) only inspected a local region of the visual field for a single feature, and these local regions became known as **local receptive fields**.
- Neural network research started to design networks in which neurons in one layer received input only from a localized subset of neurons in the preceding layer, i.e. each neuron had a local receptive field in the preceding layer.
- With local receptive fields, neurons can learn to extract low-level features in the input (such as a segment or an oriented edge in an image), and these features can be passed on to neurons in later layers that combine these low-level features into more complex features.

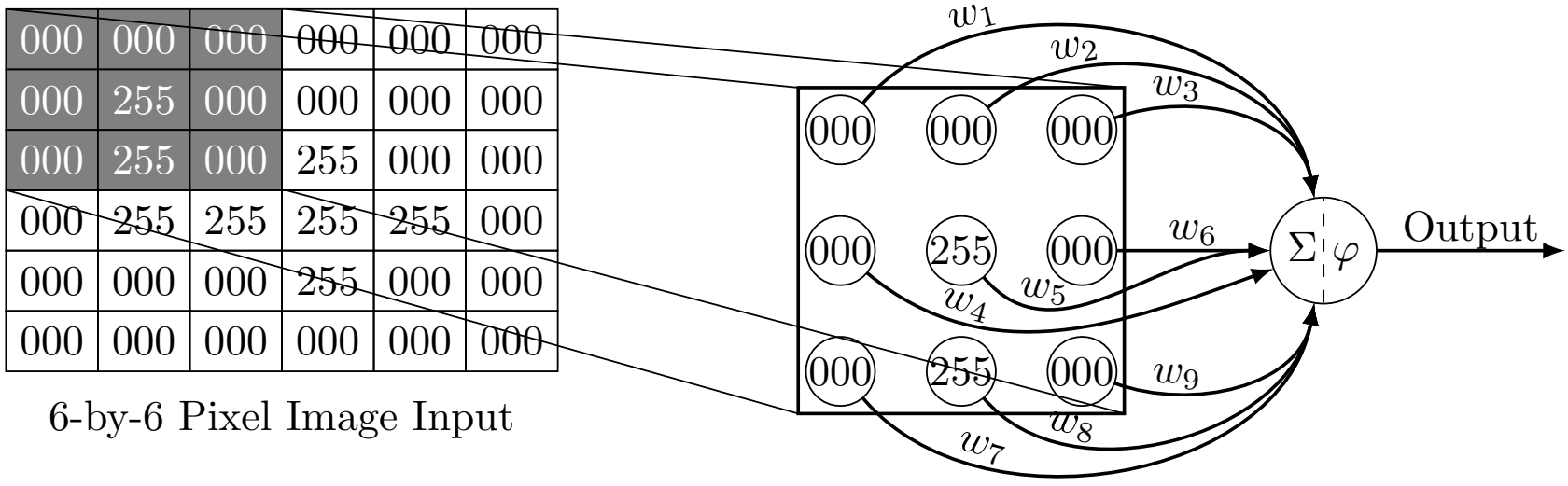


Figure 32: A 6-by-6 matrix representation of a grayscale image of a 4, and a neuron with a receptive field that covers the top-left corner of the image. This figure was inspired by Figure 2 of (**?**).

Just to simplify the coming explanations, bias will be omitted (though they obviously are used in CNNs).

To illustrate, let us assume that the neuron shown in Figure 32[194] uses the following set of weights:

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \tag{91}$$

$$\begin{aligned} a_i = rectifier((w_1 \times 000) + (w_2 \times 000) + (w_3 \times 000) \\ + (w_4 \times 000) + (w_5 \times 255) + (w_6 \times 000) \\ + (w_7 \times 000) + (w_8 \times 255) + (w_9 \times 000)) \\ = rectifier((0 \times 000) + (0 \times 000) + (0 \times 000) \\ + (1 \times 000) + (1 \times 255) + (1 \times 000) \\ + (0 \times 000) + (0 \times 255) + (0 \times 000)) \\ = 255 \end{aligned} \tag{92}$$

- With the previous set of weights, the neuron activation is solely dependent on the values along the middle row of inputs.
- Neurons using this set of weights are rudimentary detectors for horizontal edges since their maximum activation is when there is a horizontal line across the middle row of their inputs.
- This is the case for the next neuron's different local receptive field but with the same set of weights.

- The neuron's weights determine the type of visual feature to which it activates in response.
- These weight matrices are called **filters** because they filter the input by returning high activations for certain patterns of inputs and low activations for others.
- These filters are learned by the CNN just like for fully connected nets.
- Learning the filters means that the network is able to learn which visual patterns are useful to extract from the visual input in its prediction task.

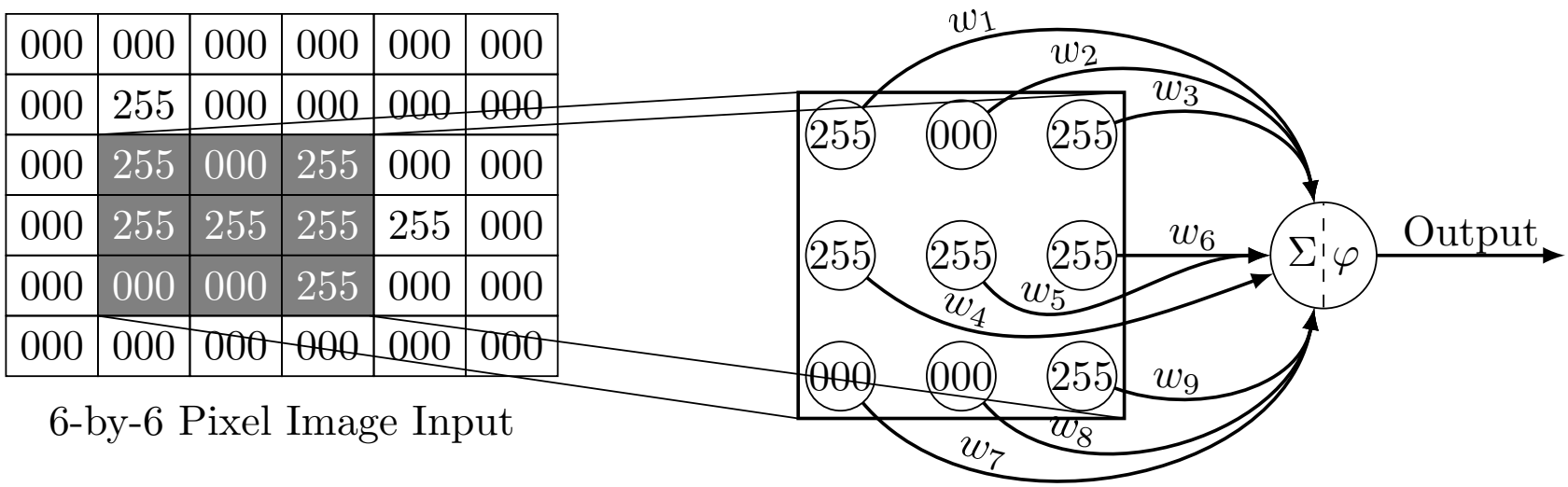


Figure 33: A 6-by-6 matrix representation of a grayscale image of a 4, and a neuron with a different receptive field from the neuron in Figure 32[194]. This figure was inspired by Figure 2 of (**?**).

The maximum grayscale values (255) across the middle row of the neurons receptive field has resulted in the neuron having a very large activation for this input.

Calculation of the neuron's activation:

$$\begin{aligned} a_i &= rectifier((w_1 \times 255) + (w_2 \times 000) + (w_3 \times 255) \\ &\quad + (w_4 \times 255) + (w_5 \times 255) + (w_6 \times 255) \\ &\quad + (w_7 \times 000) + (w_8 \times 000) + (w_9 \times 255)) \\ &= rectifier((0 \times 255) + (0 \times 000) + (0 \times 255) \\ &\quad + (1 \times 255) + (1 \times 255) + (1 \times 255) \\ &\quad + (0 \times 000) + (0 \times 000) + (0 \times 255)) \\ &= 765 \end{aligned} \tag{93}$$

Clearly, there are many different weight matrices that could be defined, each of which would cause a neuron to activate in response to a different visual pattern in its local receptive field.

Clearly, there are many different weight matrices that could be defined, each of which would cause a neuron to activate in response to a different visual pattern in its local receptive field.

Other sets of weights that the example neurons could use:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & +1 & -1 \\ -1 & +1 & -1 \\ -1 & +1 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ +1 & +1 & +1 \\ -1 & -1 & -1 \end{bmatrix} \tag{94}$$

- When a neuron applies a filter to its local receptive field, it acts as a local visual feature detector presented to it by the input values.
- The neuron will have a high activation if the feature occurs in its local receptive field.
- But an image processing system should be able to detect a visual feature in an image wherever it occurs in the image.
- If so, the model is said to be **equivariant** to the translation of features.
- CNNs achieve translation equivariant feature detection through weight sharing.
- Neurons are organized into groups such that all the neurons in a group apply the same filter to their inputs.

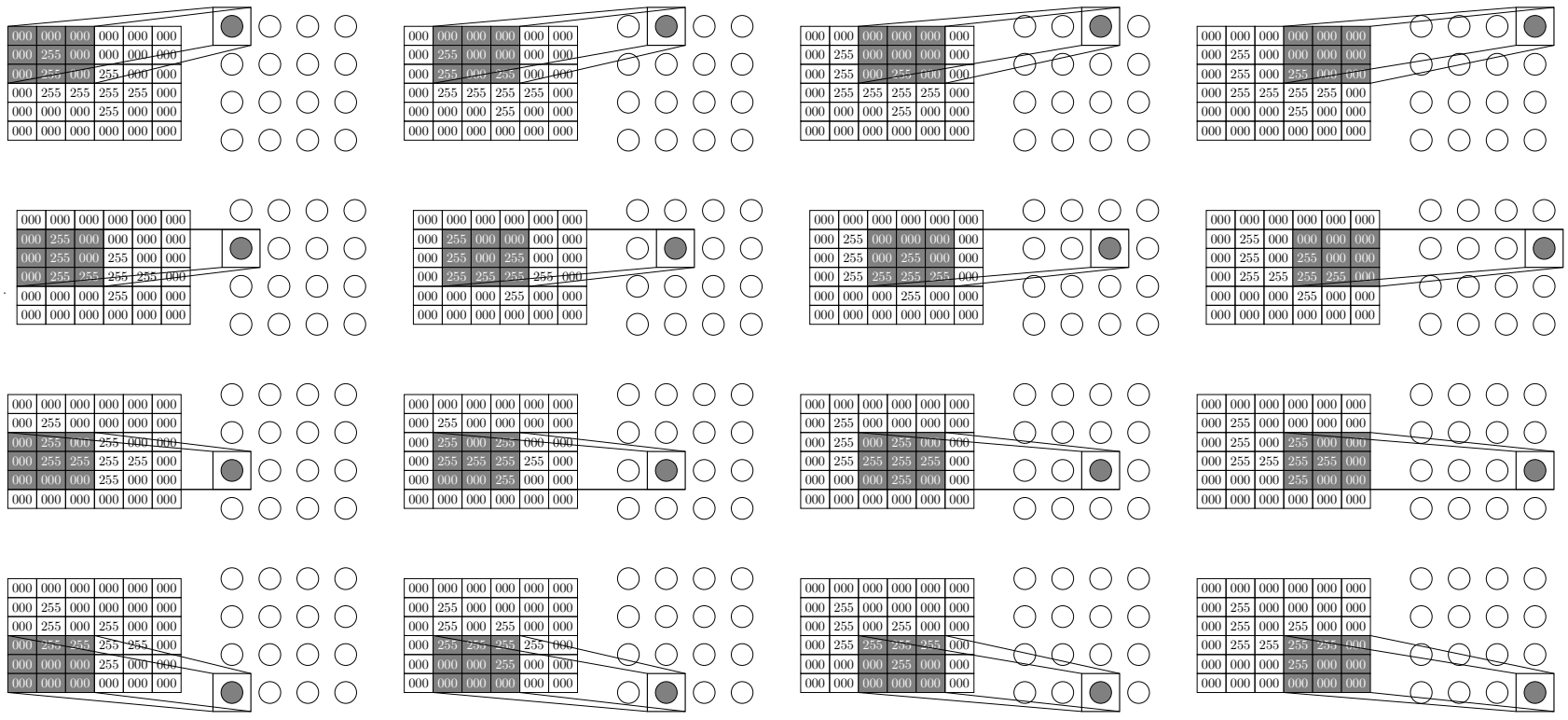


Figure 34: Illustration of the organization of a set of neurons that share weights (use the same filter) and their local receptive fields such that together the receptive fields cover the entirety of the input image.

- When two or more neurons share a filter, each weight in the filter is used multiple times during the forward pass of the training algorithm to process a given input (once by each neuron that uses the filter).
- During the backward pass, a separate weight update is calculated for each neuron that uses the weight, and then the final weight update that is applied is the sum of these separate weight updates.
- This is similar to the weight updates in batch training except that here for each training example we sum over the weight updates for each neuron that uses the weight (as opposed to weight updates for different training examples).

- When two or more neurons share a filter, each weight in the filter is used multiple times during the forward pass of the training algorithm to process a given input (once by each neuron that uses the filter).
- During the backward pass, a separate weight update is calculated for each neuron that uses the weight, and then the final weight update that is applied is the sum of these separate weight updates.
- This is similar to the weight updates in batch training except that here for each training example we sum over the weight updates for each neuron that uses the weight (as opposed to weight updates for different training examples).
- For m different neurons:

$$\Delta w_{i,*} = \sum_{i=1}^{m} \boldsymbol{\delta}_i \times a_*$$

$$w_{i,*} \leftarrow w_{i,*} - \alpha \times \Delta w_{i,*} \tag{95}$$

- In mathematics, the process of passing a function over a sequence of values is known as **convolving a function**.
- By analogy, a set of neurons that share a filter (thus each implementing the same function) and are organized such that together their receptive fields cover the input, are convolving a function over the input.
- As such, if the relevant visual feature (where relevance is defined by the filter used by a set of neurons) occurs anywhere in the input, then at least one of the neurons in the set will have a high activation.
- The activations of the neurons in a set provide a map of where in the input the relevant visual feature occurred, and for this reason the set of activations for a set of neurons that share a filter is called a **feature map**.

This slide and the following illustrate how the feature map generated by the neurons change if the filter used by the neurons to process the example input is changed.

$$\underbrace{\begin{bmatrix} 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & \mathbf{255} & 000 & 000 & 000 & 000 \\ 000 & \mathbf{255} & 000 & \mathbf{255} & 000 & 000 \\ 000 & \mathbf{255} & \mathbf{255} & \mathbf{255} & \mathbf{255} & 000 \\ 000 & 000 & 000 & \mathbf{255} & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 \end{bmatrix}}_{\text{Input Image}} \rightarrow \underbrace{\begin{bmatrix} -1 & +1 & -1 \\ -1 & +1 & -1 \\ -1 & +1 & -1 \end{bmatrix}}_{\text{Convolved Filter}} \rightarrow \underbrace{\begin{bmatrix} 510 & 0 & 255 & 0 \\ 510 & 0 & 0 & 0 \\ 255 & 0 & 255 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{Feature Map}}$$

(96)

$$\underbrace{\begin{bmatrix} 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & \mathbf{255} & 000 & 000 & 000 & 000 \\ 000 & \mathbf{255} & 000 & \mathbf{255} & 000 & 000 \\ 000 & \mathbf{255} & \mathbf{255} & \mathbf{255} & \mathbf{255} & 000 \\ 000 & 000 & 000 & \mathbf{255} & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 \end{bmatrix}}_{\text{Input Image}} \rightarrow \underbrace{\begin{bmatrix} -1 & -1 & -1 \\ +1 & +1 & +1 \\ -1 & -1 & -1 \end{bmatrix}}_{\text{Convolved Filter}} \rightarrow \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 255 & 0 & 255 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{Feature Map}} \tag{97}$$

Filter hyperparameters: Dimension, stride, and padding

- The dimensionality of a feature map generated by applying a filter to an input is determined by the number of neurons used to process the input (each element in a feature map corresponds to the output of one neuron).
- Three hyper-parameters affect the number of neurons required to entirely cover the input, and hence the dimensionality of the resulting feature map: **filter dimensions**, **stride**, and **padding**.

- The illustration in Figure 34[203] assumes that the neurons are using a two-dimensional 3-by-3 (height by width) filter.
- Larger and smaller filters are possible. With larger filters the number of neurons required to cover the input gets smaller and vice versa.
- Example: if the dimension of the filter is 2-by-2, then the set of neurons needs to be a 5-by-5 layer in order to cover the input, yielding a 5-by-5 feature map.
- Choosing a good filter size for a given dataset often involves a trial-and-error process.
- So far all filter examples have been two-dimensional because the focus on processing a grayscale image that is a 2D input. It is possible to use 1D filters, or with three or more dimensions.

- **The stride parameter** specifies the distance between the center of the local receptive field of one neuron and the center of the local receptive fields of its horizontal or vertical neighbor in the set of neurons sharing the filter.
- In Figure 34[203], a horizontal and vertical stride of 1. This means that there is a relatively large overlap in the receptive fields between a neuron and its neighbors.
- Other strides are possible (possibly different horizontal and vertical strides).
- Example: if we used a horizontal and vertical stride of 3 in Figure 34[203], then there would be no overlap between the receptive fields of different neurons $\Longrightarrow$ reduced number of neurons required to cover the input.
- Finding the appropriate stride for a given dataset involves trial-and-error experimentation.

- Two related phenomena may be undesirable consequences of how the receptive fields for the neurons have been defined.
  1. If we use a 4-by-4 layer of neurons, to cover a 6-by-6 input matrix, the dimensionality of the resulting feature map is also 4-by-4. In some cases we may wish to avoid this reduction in dimensionality.
  2. There is a difference in the number of times that each pixel in the image is used as an input to a neuron in the grid. E.g. only one of the receptive fields covers the top-left pixel in the image, whereas nine receptive fields cover the pixel at coordinate (3,3).
- Both phenomena are a consequence of applying the filter only to valid pixels in the image, i.e. not considering any imaginary pixels (**padding**) around its border.

- Two related phenomena may be undesirable consequences of how the receptive fields for the neurons have been defined.
  1. If we use a 4-by-4 layer of neurons, to cover a 6-by-6 input matrix, the dimensionality of the resulting feature map is also 4-by-4. In some cases we may wish to avoid this reduction in dimensionality.
  2. There is a difference in the number of times that each pixel in the image is used as an input to a neuron in the grid. E.g. only one of the receptive fields covers the top-left pixel in the image, whereas nine receptive fields cover the pixel at coordinate (3,3).
- Both phenomena are a consequence of applying the filter only to valid pixels in the image, i.e. not considering any imaginary pixels (**padding**) around its border.
- Using padding or not is task dependent and is often based on trial and error.
- When padding is applied, it is generally added to all edges as equally as is possible.

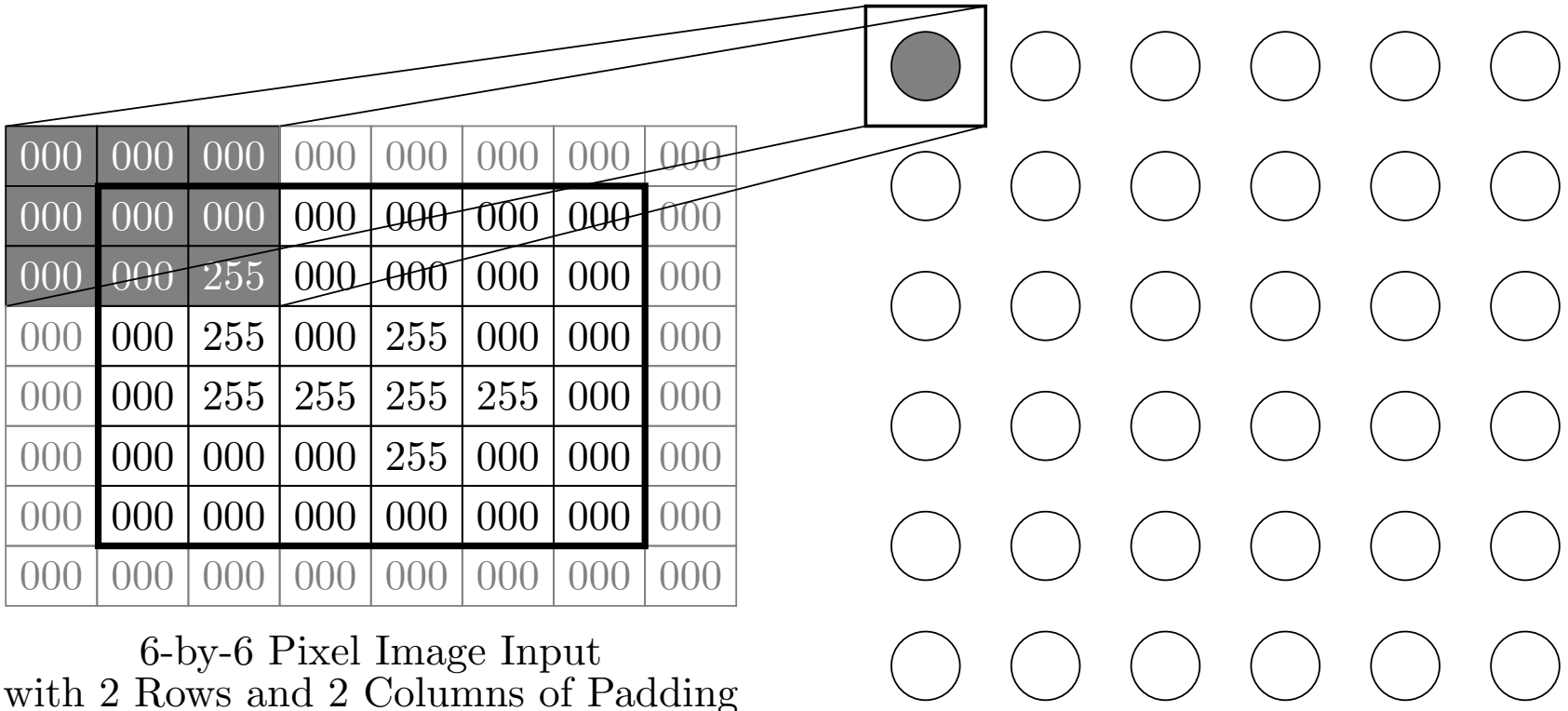


Figure 35: A grayscale image of a 4 after padding has been applied to the original 6-by-6 matrix representation, and the local receptive field of a neuron that includes both valid and padded pixels.

- A popular combination is to use a stride length of 1 and to pad the image with imaginary pixels.
- This combination ensures that the output from a layer of neurons applying a filter across an image has the same dimensions as the input image.
- Maintaining the dimensionality between input and output becomes important in CNNs when using multiple layers of neurons, the output for one layer being interpreted as the image input to the next layer.

$$\underbrace{\begin{bmatrix} 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & \mathbf{255} & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & \mathbf{255} & 000 & \mathbf{255} & 000 & 000 & 000 \\ 000 & 000 & \mathbf{255} & \mathbf{255} & \mathbf{255} & \mathbf{255} & 000 & 000 \\ 000 & 000 & 000 & 000 & \mathbf{255} & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \end{bmatrix}}_{\text{Input Image}} \rightarrow \underbrace{\begin{bmatrix} -1 & +1 & -1 \\ -1 & +1 & -1 \\ -1 & +1 & -1 \end{bmatrix}}_{\text{Convolved Filter}} \rightarrow \underbrace{\begin{bmatrix} 0 & 255 & 0 & 0 & 0 & 0 \\ 0 & 510 & 0 & 255 & 0 & 0 \\ 0 & 510 & 0 & 0 & 0 & 0 \\ 0 & 255 & 0 & 255 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 255 & 0 & 0 \end{bmatrix}}_{\text{Feature Map}}$$

(98)

$$\underbrace{\begin{bmatrix} 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & \mathbf{255} & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & \mathbf{255} & 000 & \mathbf{255} & 000 & 000 & 000 \\ 000 & 000 & \mathbf{255} & \mathbf{255} & \mathbf{255} & \mathbf{255} & 000 & 000 \\ 000 & 000 & 000 & 000 & \mathbf{255} & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \end{bmatrix}}_{\text{Input Image}} \rightarrow \underbrace{\begin{bmatrix} -1 & -1 & -1 \\ +1 & +1 & +1 \\ -1 & -1 & -1 \end{bmatrix}}_{\text{Convolved Filter}} \rightarrow \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 255 & 0 & 255 & 0 & 255 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{Feature Map}}$$

(99)

- The precise location of a visual feature in an image may not be relevant for an image-processing task.
- E.g., knowing that there is an eye in the top-left region of an image is useful for face recognition, but the extra precision of knowing that it is centered at pixel (998,742) may not be useful.
- In training an image processing model we typically want the model to generalize over the precise locations of features in training images in any location.
- The most straightforward way to make a model abstract away from the precise location of visual features is to sub-sample the feature maps.
- Typically, the local receptive fields of neurons in a sub-sampling layer do not overlap.

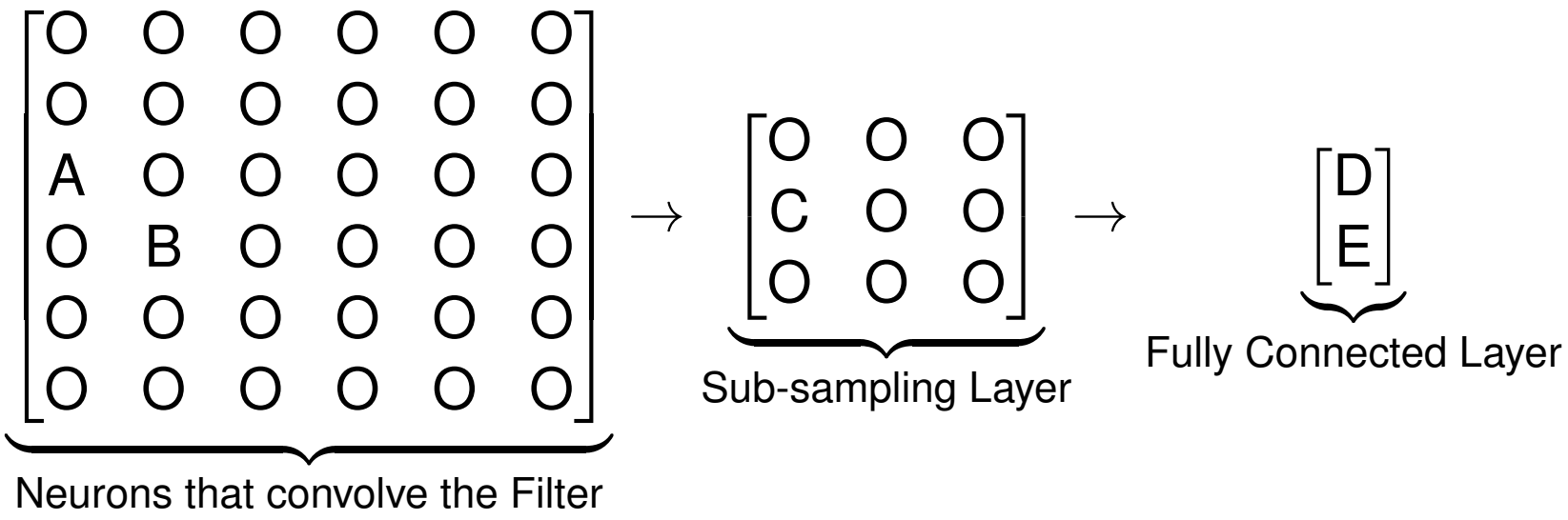
- Consequently, there are fewer output activations from a sub-sampling layer than there are inputs: one output per local receptive field and multiple inputs per field.
- The amount of sub-sampling applied is dependent on the dimensions of the receptive fields of the neurons; for example, using non-overlapping 2-by-2 receptive fields, the output from a sub-sampling layer will have half the number of rows and columns as the feature map input to the layer.
- In early CNNs, the activation of sub-sampling neurons was often the average of the values in the feature map covered by the local receptive field of the neuron.
- Many modern CNNs use a max function that returns the maximum value in the region of the feature map covered by the local receptive field (**max pooling**).

$$\underbrace{\begin{bmatrix} 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & \mathbf{255} & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & \mathbf{255} & 000 & \mathbf{255} & 000 & 000 & 000 \\ 000 & 000 & \mathbf{255} & \mathbf{255} & \mathbf{255} & \mathbf{255} & 000 & 000 \\ 000 & 000 & 000 & 000 & \mathbf{255} & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \end{bmatrix}}_{\text{Input Image}} \rightarrow \underbrace{\begin{bmatrix} -1 & -1 & -1 \\ +1 & +1 & +1 \\ -1 & -1 & -1 \end{bmatrix}}_{\text{Convolved Filter}} \rightarrow$$

$$\rightarrow \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 255 & 0 & 255 & 0 & 255 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{Feature Map}} \rightarrow \text{max pooling} \rightarrow \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 255 & 255 & 255 \\ 0 & 0 & 0 \end{bmatrix}}_{\text{Output from Sub-sampling}}$$

(100)

- Using a max function requires that the backward pass in the backpropagation algorithm be slightly modified.
- This is because a max function allows through only the maximum value from its inputs, and so the non-max values did not affect the output (and hence the error) of the network.
- As a result, there is no error gradient with respect to the non-max values that the max function received.
- Furthermore, the gradient for the max function $\frac{\partial a}{\partial z}$ for the max value is 1, because the output of the activation function will be linear for small changes in the input value that achieved the max (i.e., it will change by the same amount as that input value is changed).
- $\implies$ in backpropagating through a max function, the entire error gradient is backpropagated to the neuron that propagated forward the max value, and the other neurons receive an error gradient of zero.


O O O O O O
O O O O O O
A O O O O O
O B O O O O
O O O O O O
O O O O O O
Neurons that convolve the Filter
O O O
C O O
O O O
Sub-sampling Layer
D
E
Fully Connected Layer

(101)

The backpropagation process through the CNN assumes that the δs for the neurons D and E have already been calculated. So the δ for Neuron C can now be calculated.

$$\begin{aligned}\delta_C &= \frac{\partial \mathcal{E}}{\partial a_C} \times \frac{\partial a_C}{\partial z_C} \\ &= ((\delta_D \times w_{D,C}) + (\delta_E \times w_{E,C})) \times 1\end{aligned} \tag{102}$$

The backpropagation process through the CNN assumes that the δs for the neurons D and E have already been calculated. So the δ for Neuron C can now be calculated.

$$\begin{aligned}\delta_C &= \frac{\partial \mathcal{E}}{\partial a_C} \times \frac{\partial a_C}{\partial z_C} \\ &= ((\delta_D \times w_{D,C}) + (\delta_E \times w_{E,C})) \times 1\end{aligned} \tag{102}$$

We calculate the δs for the other neurons in the sub-sampling layer in a similar fashion.

Once we have calculated a δ for each of the neurons in the sub-sampling layer, we can backpropagate these δs to the layer of neurons that convolve the filter.
Recall that the receptive fields of the neurons in the sub-sampling layer do not overlap. So, each of the neurons in the first layer connects only to a single neuron in the sub-sampling layer.

$$\begin{aligned}\delta_B &= \frac{\partial \mathcal{E}}{\partial a_B} \times \frac{\partial a_B}{\partial z_B} \\ &= (\delta_D \times w_{C,B}) \times \frac{\partial a_B}{\partial z_B} \\ &= (\delta_D \times 1) \times 1\end{aligned} \tag{103}$$

- The δs for the other neurons in the first layer will either be 0 (if, like Neuron A, they did not produce the maximum value in the local receptive field of the sub-sampling neuron to which they connect) or can be calculated in a similar way to Neuron B.
- Once the δ for each of the neurons in this layer has been calculated, the weight updates for each weight in the filter can be calculated by

- So far all example filters were 2-D because the MNIST dataset we use involves grayscale images $\implies$ only a 2-D filter required because all pixel information for an image can be represented in a 2-D matrix indexing over the height and width of the grayscale image.
- Color images typically encode three types of information for each pixel: red, green and blue information (other colors generated by combination of these three primary colors).
- Encoding **RGB** information is normally done using a separate 2-D matrix for each color, with the dimensions of each being equal to the image resolution. The term channel is used to describe the number of matrices used to encode the information in an image.
- An RGB image has three **channels**, and a grayscale image will have one channel.
- To process RGB images, use 3-D filters: height by width by channel.

The third dimension of a filter is referred to as the **depth** of the filter.
Illustration of the structure of a 3-D filter. (The bias term w_0 is included.)

$$\left[\underbrace{w_0}_{\text{bias}} \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} \begin{bmatrix} w_5 & w_6 \\ w_7 & w_8 \end{bmatrix} \begin{bmatrix} w_9 & w_{10} \\ w_{11} & w_{12} \end{bmatrix} \right] \tag{104}$$

- Adding depth to a filter does not involve a major change in the way a neuron applies a filter to its local receptive field.
- E.g., if a neuron is applying a 2-by-2-by-3 filter, then its local receptive field will have the same dimensions.
- For color images, we can distinguish the different layers of depth by the color channels.
- In this context, the neuron will apply a different 2-by-2 filter to each color channel (values of the pixels in the receptive field).
- Then the results of these three calculations are summed together along with the bias, to generate a single scalar value that is pushed through the activation function and then stored in the feature map

The following lists a 2-by-2-by-3 filter:

$$\left[\underbrace{w_0 = 0.5}_{\text{bias}} \underbrace{\begin{bmatrix} w_1 = 1 & w_2 = 1 \\ w_3 = 0 & w_4 = 0 \end{bmatrix}}_{\text{Red Channel}} \underbrace{\begin{bmatrix} w_5 = 0 & w_6 = 1 \\ w_7 = 0 & w_8 = 1 \end{bmatrix}}_{\text{Green Channel}} \underbrace{\begin{bmatrix} w_9 = 1 & w_{10} = 0 \\ w_{11} = 0 & w_{12} = 1 \end{bmatrix}}_{\text{Blue Channel}} \right] \tag{105}$$

The previous filter is used to process the following 3-by-3 RGB image:

$$\left[\underbrace{\begin{bmatrix}1&1&1\\0&0&0\\0&0&0\end{bmatrix}}_{\text{Red Channel}}\underbrace{\begin{bmatrix}0&0&2\\0&0&2\\0&0&2\end{bmatrix}}_{\text{Green Channel}}\underbrace{\begin{bmatrix}3&0&0\\0&3&0\\0&0&3\end{bmatrix}}_{\text{Blue Channel}}\right] \tag{106}$$

Assuming a stride length of 1 and no padding on the input, we would require a 2-by-2 layer of neurons to convolve the filter over the previous image.
The top-left neuron in this layer would have a local receptive field covering the 2-by-2 square in the top-left of each of the channels.
The following lists the values from the image that are inside this neuron's local receptive field:

$$\Big[\underbrace{\begin{bmatrix}1 & 1\\ 0 & 0\end{bmatrix}}_{\text{Red Channel}}\ \underbrace{\begin{bmatrix}0 & 0\\ 0 & 0\end{bmatrix}}_{\text{Green Channel}}\ \underbrace{\begin{bmatrix}3 & 0\\ 0 & 3\end{bmatrix}}_{\text{Blue Channel}}\Big] \tag{107}$$

The output activation for this neuron would be calculated as follows:

$$
\begin{aligned}
z &= ((w_0 \times 1) \\
&\quad + (w_1 \times 1) + (w_2 \times 1) + (w_3 \times 0) + (w_4 \times 0) \\
&\quad + (w_5 \times 0) + (w_6 \times 0) + (w_7 \times 0) + (w_8 \times 0) \\
&\quad + (w_9 \times 3) + (w_{10} \times 0) + (w_{11} \times 0) + (w_{12} \times 3)) \\
&= 0.5 + 1 + 1 + 0 + 0 + 0 + 0 + 0 + 0 + 3 + 0 + 0 + 3 \\
&= 8.5 \\
a &= rectifier(z) \\
&= rectifier(8.5) \\
&= 8.5
\end{aligned}
\tag{108}
$$

The activations for the other three neurons using this filter would be calculated in a similar way, resulting in the following feature map being generated by this 2-by-2 layer of neurons:

$$\begin{bmatrix} 8.5 & 6.5 \\ 0.5 & 10.5 \end{bmatrix} \tag{109}$$

- Adding depth to filters (1) enables CNNs to process multi-dimensional input and (2) enables the CNN to apply multiple filters in parallel to the same input and for later layers in the network to integrate information from across these layers.
- So CNNs can learn to identify, extract, and use multiple different features in the input.
- The sequence of convolving a filter over an input, then applying a non-linear activation function, and finally sub-sampling the resulting feature maps is relatively standard in most modern CNNs, and often this sequence of operations is taken as defining a convolutional layer.

- As previously discussed, a CNN may have multiple convolutional layers in sequence because the outputs from a sub-sampling layer may be passed as an input to another filter convolution, and so this sequence of operations may be repeated multiple times.
- Padding may be applied to retain dimensionality, and in some cases the non-linearity activation or sub-sampling may be dropped in some convolutional layers.
- Generally, the later layers of a CNNs will include one or more fully connected layers with a softmax output layer if the model is used for classification.

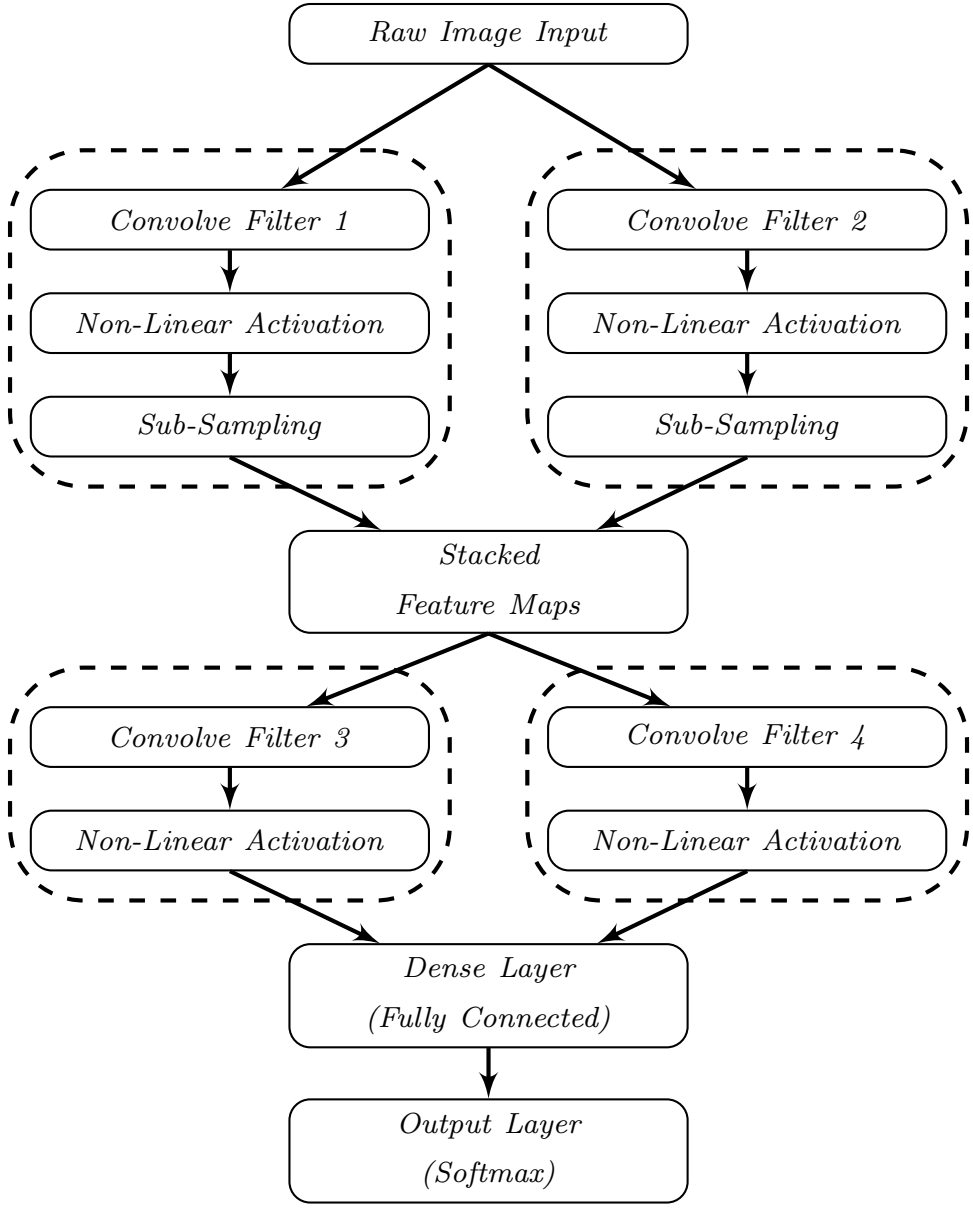


Figure 36: Schematic of the typical sequences of layers found in a convolutional neural network.

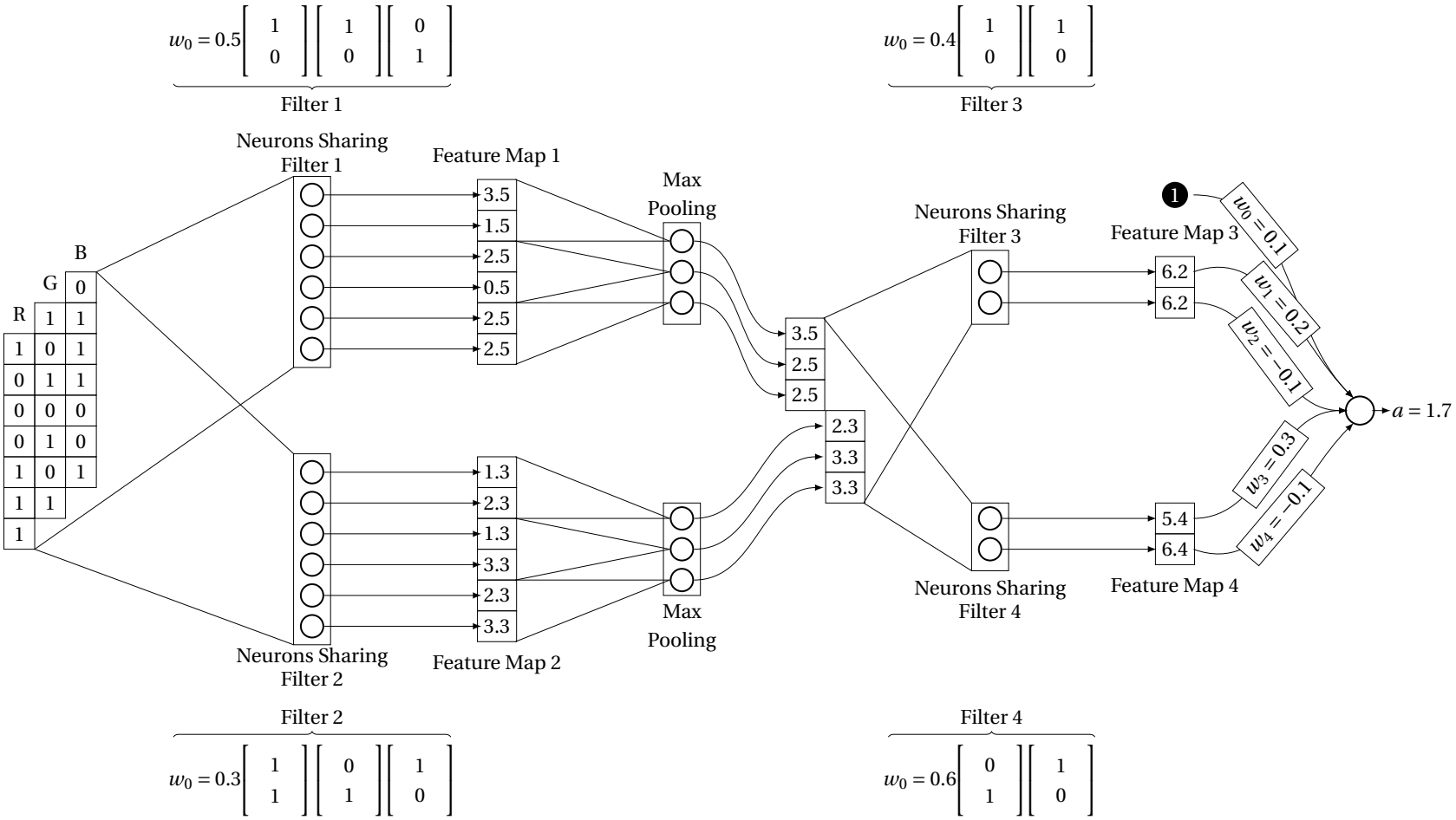


Figure 37: Worked example illustrating the dataflow through a multilayer, multifilter CNN.

Recurrent Neural Networks

- The convolutional neural networks are ideally suited to processing data that have a fixed-size grid-like structure and where the basic features have a local extent, such as images.
- In many domains the data has a sequential varying-length structure and interactions between data points may span long distances in the sequence.
- Natural language is an example of this type of data: it is sequential, one word follows the other, each sentence may have a different number of words, and it contains long-distance dependencies between elements.
- Example, in English, the subject and verb of a sentence should agree.

- Processing such data requires a model that can remember relevant information from earlier in the sequence.
- **Recurrent neural networks (RNN)** are designed to process this type of data.
- An RNN works in discrete time. In processing a sequence, it takes one input from the sequence at each time point.
- Its main characteristic is that it contains feedback connections, and so, unlike a feedforward network, which is a directed acyclic graph, an RNN is a **directed cyclic graph**.
- With these cycles, or recurrent links, the output from a neuron at one time point may be fed back into the same neuron at another time point.
- As such the network has a memory over past activations.

- A simple RNN architecture is a feedforward architecture with one hidden layer that has been extended with a memory buffer that is used to store the activations from the hidden layer for one time-step.
- On each time-step, the information stored in the memory buffer is concatenated with the next input to each neuron.
- In the sequel, x denotes an input; h denotes a hidden layer; and y denotes the output from the network. t will be a subscript to denote a point in time.

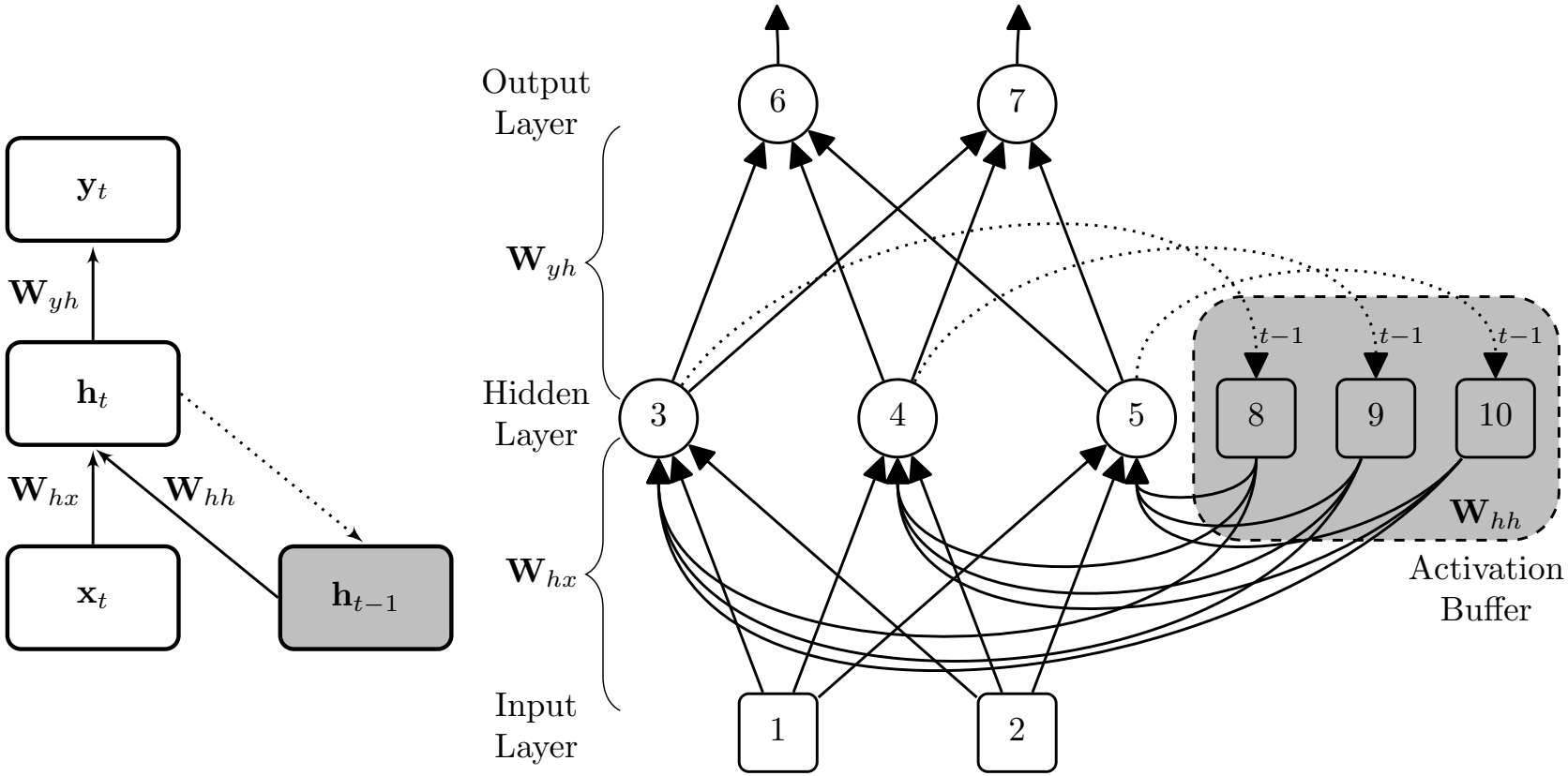


Figure 38: Schematic of the simple recurrent neural architecture.

- There are no weights on the connections between the output of the hidden layer and the memory buffer because the transfer of hidden neuron activations to the memory buffer is a simple copy operation (dashed arrow).
- There are weights on the connections from the memory buffer to each of the neurons because the information read from the memory buffer is processed by the hidden neurons as for each of the inputs.
- The three weight matrices are:
  1. W_{hx} containing the weights for the connections between the input layer (x) and the hidden layer (h);
  2. W_{yh} containing the weights for the connections between the hidden layer (h) and the output layer (y); and
  3. W_{hh} containing the weights for the connections between the memory buffer and the hidden layer.

The forward propagation of the activations through a simple RNN is defined as follows (the subscript t denoting the time-step of the system; and there is one input per time-step?although this input may be a vector of values):

$$\mathbf{h}_t = \varphi\left((\mathbf{W}_{hh} \cdot \mathbf{h}_{t-1}) + (\mathbf{W}_{hx} \cdot \mathbf{x}_t) + \mathbf{w}_0\right) \tag{110}$$

$$\mathbf{y}_t = \varphi\left(\mathbf{W}_{yh} \cdot \mathbf{h}_t\right) \tag{111}$$

- Training an RNN using backpropagation is quite similar to training a normal feedforward network.
- For example, **if the network** shown on the right of the previous figure **was applied only to a single input**, one would calculate the δs for the neurons in the output layer and then backpropagate them to the hidden layer neurons.
- The main novelty now is that the neurons in the hidden layer of a simple RNN have two weight matrices associated with them: W_{hx} and W_{hh}.
- For the update of the weights on the connections into a hidden layer neuron, this distinction is irrelevant and **the same δ is used to update all the weights on the connections into a neuron**.

- **When an RNN is applied to a sequence of inputs**, things become a little more complex.
- The variant of backpropagation used to train an RNN is called **backpropagation through time**.
- An RNN is deliberately designed so that when the network processes an input at time t in a sequence, the output generated is dependent not only on input t but also on the previous inputs in the sequence.
- As such, the error for the output at time t is also dependent on the states of the network for all the previous inputs in the sequence ($t-1$, $t-2$, and so on) back to the start of the sequence.

- **When an RNN is applied to a sequence of inputs**, things become a little more complex.
- The variant of backpropagation used to train an RNN is called **backpropagation through time**.
- An RNN is deliberately designed so that when the network processes an input at time t in a sequence, the output generated is dependent not only on input t but also on the previous inputs in the sequence.
- As such, the error for the output at time t is also dependent on the states of the network for all the previous inputs in the sequence ($t-1$, $t-2$, and so on) back to the start of the sequence.
- Because of the RNN design, we must backpropagate the error at time t to all the parameters that contributed to the error, i.e. through the previous states of the network.
- Hence the name backpropagation through time: as we backpropagate through the previous states of the network, we are in effect going back through time.

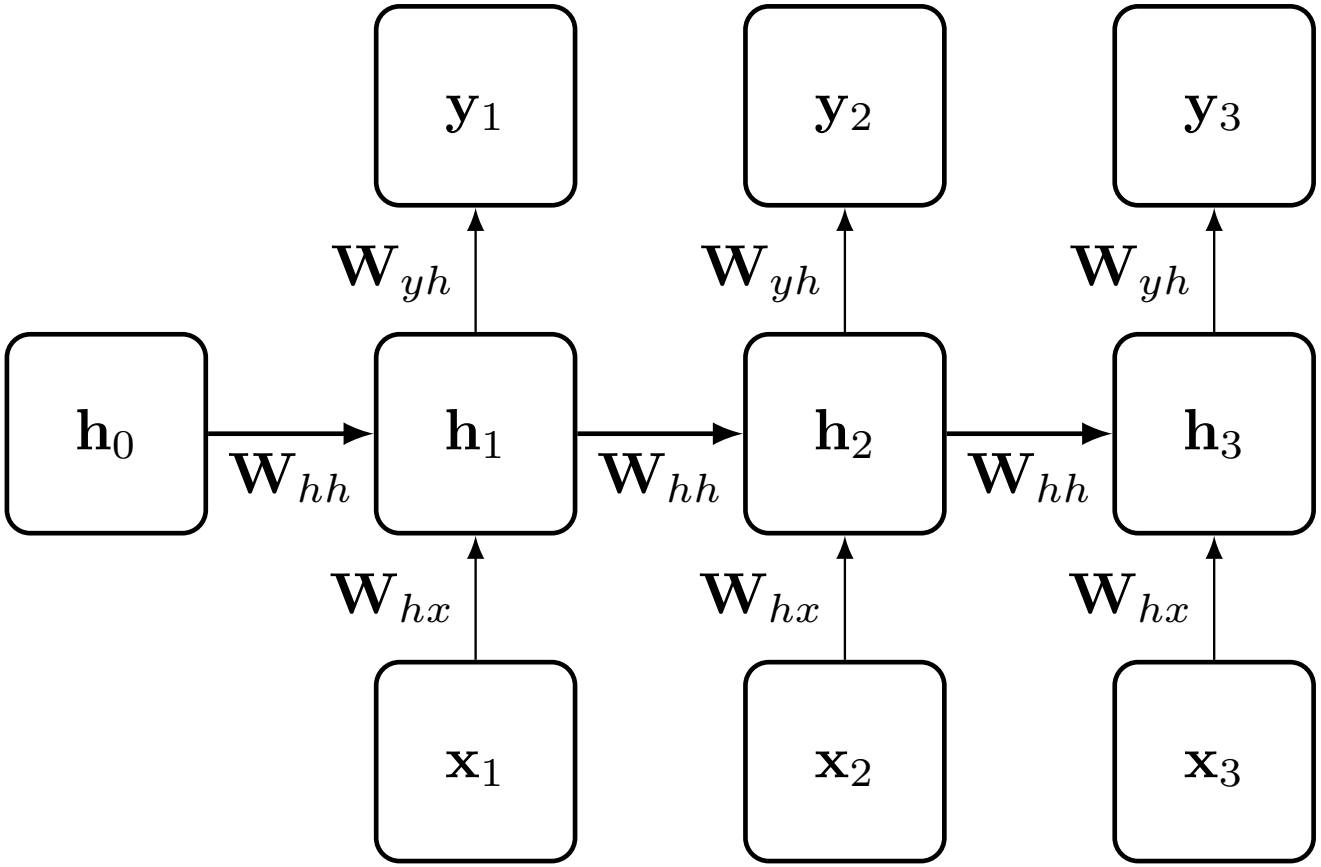


Figure 39: A simple RNN model unrolled through time (in this instance, three time-steps).

- Notice that unlike the layers, the weight matrices do not have a time subscript on them because the network uses the same weights to process Input 1 as it does to process Input 3.
- Indeed, though the network is unrolled through three time-steps, it still has only 3 weight matrices (not 9).
- In a sense, the unrolled recurrent neural network is similar to a CNN in that weights are shared between different neurons: the neurons in the hidden layer at time $t = 1$ use exactly the same weights as the neurons in the hidden layer at time $t = 2$ and $t = 3$, and so on.

- Notice that unlike the layers, the weight matrices do not have a time subscript on them because the network uses the same weights to process Input 1 as it does to process Input 3.
- Indeed, though the network is unrolled through three time-steps, it still has only 3 weight matrices (not 9).
- In a sense, the unrolled recurrent neural network is similar to a CNN in that weights are shared between different neurons: the neurons in the hidden layer at time $t = 1$ use exactly the same weights as the neurons in the hidden layer at time $t = 2$ and $t = 3$, and so on.
- The standard process for updating a shared weight is:
  (a) to calculate the weight update for the weight at each location in the network where it is used;
  (b) to sum these weight updates together; and
  (c) finally to update the weight once using this summed weight update.

- To train a recurrent neural network using backpropagation through time, we first do a forward pass by presenting each input in the sequence in turn and unrolling the network through time.
- The unrolling of the network through time during the forward pass means that some neurons will occur multiple times in the unrolled network, and **we store the weighted sum z value and activation value a of each neuron at each time-step**.
- The slightly complicating factor here is that in an unrolled RNN a neuron may occur multiple times (for example, neurons in the hidden layer will occur once for each time-step), and so for each neuron we have a time-stamped sequence of z and a values.

- Once the forward pass is complete, we calculate an error term for each of the outputs of the network.
- The total error of the network is then the sum of these individual errors.
- For the network in the previous figure we would calculate three errors: one for y_1, one for y_2, and one for y_3.
- Each of these errors is then backpropagated through the unrolled network.

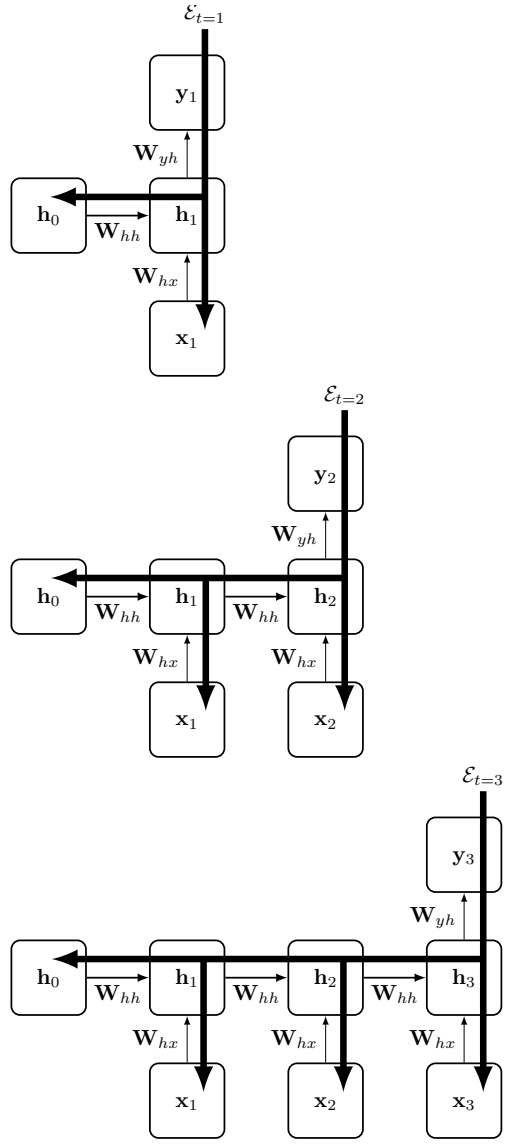


Figure 40: An illustration of the different iterations of backpropagation during backpropagation through time.

- Note that, W_{yh} was involved once in the generation of y_3 whereas W_{hh} and W_{hx} were involved three times.
- That a weight matrix may be involved multiple times in the generation of an output means that backpropagating the error for an output can result in multiple error gradients being calculated for a weight: one error gradient for each time the weight was involved in generating the output.
- In the previous Figure, we see that when $\mathcal{E}_{t=3}$ is backpropagated through the network, a single error gradient is calculated for each weight in W_{yh} because this matrix occurs only once in the unrolled network, but three separate error gradients are calculated for each weight in W_{hh} and W_{hx}.
- This is why during the forward pass a separate weighted sum z and activation a were stored for each occurrence of a neuron in the unrolled network.

- As we backpropagate through each time-step in the unrolled network, the corresponding z and a for a neuron are used at that time-step to backpropagate the error gradient for that neuron.
- In backpropagating the error $\mathcal{E}_{t=2}$ we will calculate a single error gradient for each weight in W_{yh} and two error gradients for each weight in W_{hh} and W_{hx};
- when we backpropagate the error for y_1 we calculate one error gradient for each weight in each of the three weight matrices.
- Once we have calculated all these error gradients for a sequence, we then update each weight by summing all the error gradients for that weight and then using the summed error gradient to update the weight.
- Then the weight is updated once using this summed gradient.

- it may be that an RNN outputs only a single value once it has processed the whole sequence; e.g. when training a network to process a sentence and then return a label as +ve or -ve sentiment expressed in the sentence.
- In this case we would calculate an error (or loss) only at the output at the end of the sequence.
- This error is then backpropagated through the unrolled network, in the same way that $\mathcal{E}_{t=3}$ was backpropagated, resulting in a single error gradient for each weight in W_{yh} and three separate error gradients for each weight in W_{hh} and W_{hx}.
- The error gradients for a weight are then summed and the weight updated once.

- For long sequences of inputs it is costly to keep track of all the error gradients for all the different weights in the unrolled network.
- A common practice when training a RNN to process a long sequence is to break the sequence up into subsequences.
- A typical size for a subsequence might be 20 inputs.
- The forward and backward pass is then carried out on each subsequence in turn: in the forward pass the network is unrolled over a subsequence, and in the backward pass the error gradients are backpropagated only through this truncated unrolled network.
- Information can flow forward from one subsequence to the next by using the hidden state of the network at the end of processing one subsequence to initialize the activation buffer at the start of the next subsequence.

Algorithm 3 The Backpropagation Through Time Algorithm

Require: h_0 initialized hidden state
Require: $\mathbf{x}$ a sequence of inputs
Require: $\mathbf{y}$ a sequence of target outputs
Require: n length of the input sequence
Require: Initialized weight matrices (with associated biases)
Require: $\Delta\mathbf{w}$ a data structure to accumulate the summed weight updates for each weight across time-steps

for $t = 1$ to n **do**
    $Inputs = [x_0, \ldots, x_t]$
    $h_{tmp} = h_0$
    for $i = 0$ to t **do** ▷ Unroll the network through t steps
        $h_{tmp} = ForwardPropagate(Inputs[i], h_{tmp})$
    end for
    $\hat{y}_t = OutputLayer(h_{tmp})$ ▷ Generate the output for time-step t
    $\mathcal{E}_t = \mathbf{y}[t] - \hat{y}_t$ ▷ Calculate the error at time-step t
    $Backpropagate(\mathcal{E}_t)$ ▷ Backpropagate $\mathcal{E}_t$ through t steps
    For each weight, sum the weight updates across the unrolled network and update $\Delta\mathbf{w}$
end for
Update the network weights using $\Delta\mathbf{w}$

- Using **dropout** during the training of a RNN can be problematic because dropping different neurons from the network at different time-steps across a sequence can stop the network from propagating important information forward through the sequence.
- The standard technique for applying dropout to a RNN during training is known as **variational RNN**.
- In variational RNN, the dropout mask is selected once per sequence (rather than at each input), and so the same neurons are dropped across all time-steps in the sequence.

- RNNs are particularly susceptible to the exploding gradients and vanishing gradients problems.
- Indeed, during the backward pass, error gradients will be multiplied by the W_{hh} matrix multiple times, once for each time-step through which we backpropagate.
- These multiplications of the error gradient can rapidly scale up the size of the gradient if a weight in W_{hh} is > 1 (causing our weight updates to become too large and our training to become unstable) or cause the gradient to vanish if the weight is very small.
- The vanishing and exploding gradient problems limit the ability of these networks to learn these dependencies.

- **Long short-term memory (LSTM)** networks are specifically designed to improve the ability of a recurrent network to model dependencies over long distances in a sequence.
- They remove the repeated multiplication by the W_{hh} matrix during backpropagation.
- The fundamental element in an LSTM network is the cell.
- In the following figure, the cell is depicted by the line extending from c_{t-1} to c_t across the top of the diagram.

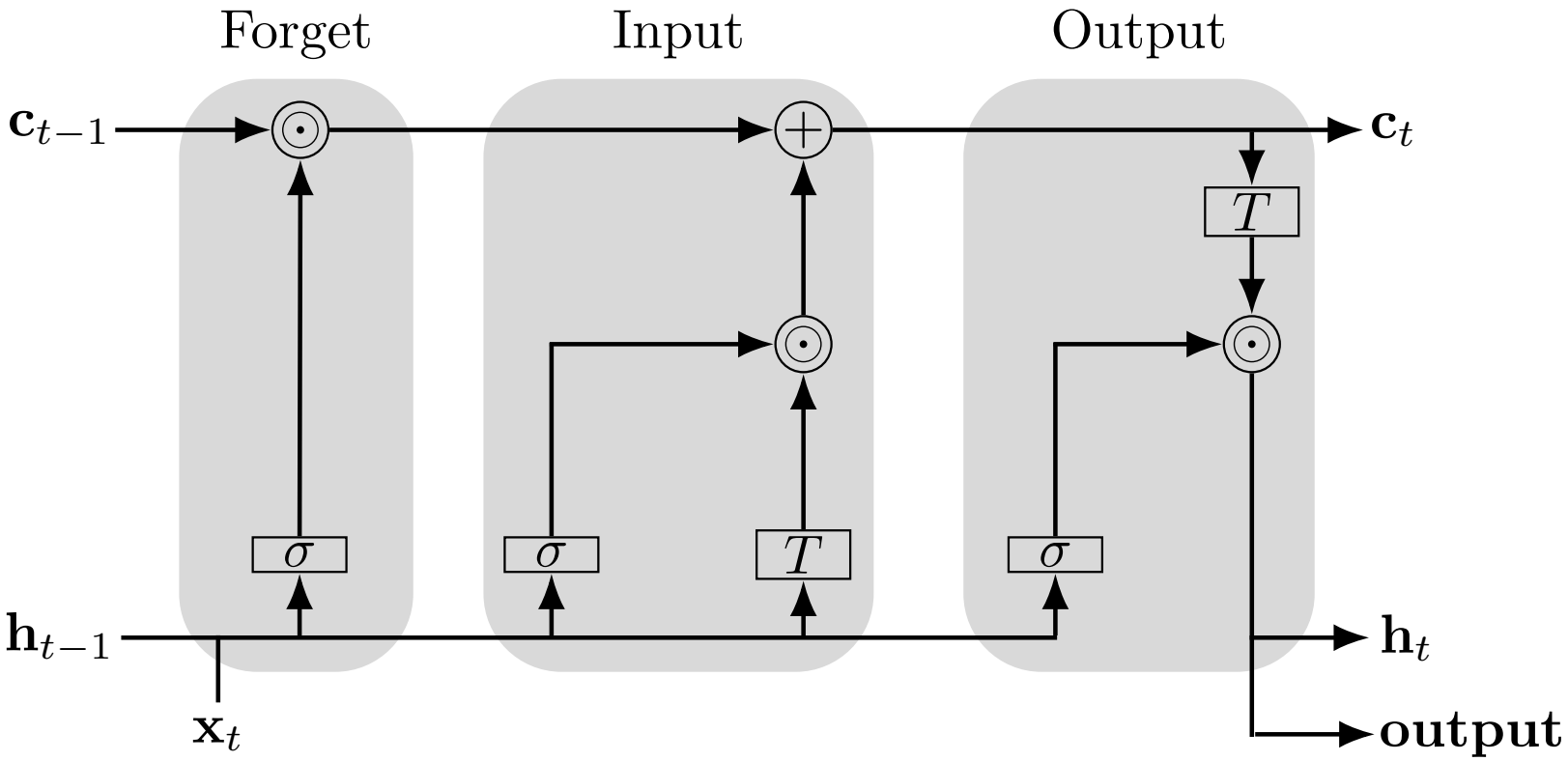


Figure 41: A schematic of the internal structure of a long short-term memory unit. This figure is based on Figure 5.4 of (**?**), which in turn was inspired by an image by Christopher Olah (available at: http://colah.github.io/posts/2015-08-Understanding-LSTMs/).

- In the LSTM Cell, the activations can take values in the range [-1,+1].
- The propagation of activations along the cell is controlled by three gates: the **forget gate**, the **input gate**, and the **output gate**.
- The forget gate removes information from the cell; the input gate adds information to the cell; and the output gate decides which information should be output by the network at the current time-step.

- In the LSTM Cell, the activations can take values in the range [-1,+1].
- The propagation of activations along the cell is controlled by three gates: the **forget gate**, the **input gate**, and the **output gate**.
- The forget gate removes information from the cell; the input gate adds information to the cell; and the output gate decides which information should be output by the network at the current time-step.
- The input to all three of these gates is the vector of hidden state activations propagated forward from the previous time-step h_{t-1} concatenated with the current input vector x_t

- The forget gate is the first gate to process the inputs to the time step.
- It works by passing hx_t through a layer of neurons that use sigmoid activation functions.
- There is one activation in the output of the sigmoid layer for each activation in the cell state.
- The sigmoid activation function outputs values in the range 0 to 1 so that the multiplication of the cell state by the sigmoid layer activations has the effect of pushing all the cell state activations that have a corresponding sigmoid activation near 0 to 0 (i.e., these activations are **forgotten**, negligible contribution) and all the cell state activations that have a corresponding sigmoid activation near 1 to be maintained (i.e., **remembered**) and propagated forward.

The following two equations respectively define the calculation of the forget mask for time-step t and the filtering of the cell state by the forget mask.

$$\mathbf{f}_t = \varphi_{sigmoid}(\mathbf{W}^{(f)} \cdot \mathbf{hx}_t) \tag{112}$$

$$\mathbf{c}_{\ddagger} = \mathbf{c}_{t-1} \odot \mathbf{f}_t \tag{113}$$

Example: Consider the following LSTM unit with the inputs and $W^{(f)}$ matrix (the zeros being the bias terms):

$$\mathbf{c}_{t-1} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \mathbf{h}_{t-1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \mathbf{x}_t = \begin{bmatrix} 4 \end{bmatrix}$$

$$\mathbf{W}^{(f)} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{114}$$

Given this context, the processing of the forget gate would be as follows (note that in this calculation hx_t is augmented with a bias input 1):

$$\underbrace{\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{W}^{(f)}} \times \underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \\ 4 \end{bmatrix}}_{\mathbf{hx}_t} = \underbrace{\begin{bmatrix} 6 \\ -6 \\ 0 \end{bmatrix}}_{\mathbf{z}_t^{(f)}} \rightarrow \varphi_{sigmoid} \rightarrow \underbrace{\begin{bmatrix} 0.997527377 \\ 0.002472623 \\ 0.500000000 \end{bmatrix}}_{\mathbf{f}_t}$$

$$\underbrace{\begin{bmatrix} 0.997527377 \\ 0.002472623 \\ 0.500000000 \end{bmatrix}}_{\mathbf{f}_t} \odot \underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}_{\mathbf{c}_{t-1}} = \underbrace{\begin{bmatrix} 0.997527377 \\ 0.002472623 \\ 0.500000000 \end{bmatrix}}_{\mathbf{c}\ddagger}$$

(115)

The calculations in the input gate are defined by the following equations:

$$\mathbf{i}\dagger_t = \varphi_{sigmoid}(\mathbf{W}^{(i\dagger)} \cdot \mathbf{hx}_t) \tag{116}$$

$$\mathbf{i}\ddagger_t = \varphi_{tanh}(\mathbf{W}^{(i\ddagger)} \cdot \mathbf{hx}_t) \tag{117}$$

$$\mathbf{i}_t = \mathbf{i}\dagger_t \odot \mathbf{i}\ddagger_t \tag{118}$$

$$\mathbf{c}_t = \mathbf{c}\ddagger + \mathbf{i}_t \tag{119}$$

The output gate uses a three-step process described mathematically in the following equations:

$$\mathbf{o}\dagger_t = \varphi_{sigmoid}(\mathbf{W}^{(o\dagger)} \cdot \mathbf{hx}_t) \tag{120}$$

$$\mathbf{o}\ddagger_t = \varphi_{tanh}(\mathbf{c}_t) \tag{121}$$

$$\mathbf{o}_t = \mathbf{o}\dagger_t \odot \mathbf{o}\ddagger_t \tag{122}$$

$$\mathbf{h}_{t+1} = \mathbf{o}_t \tag{123}$$

Bringing all the LSTM equations together specifies the sequence of calculations that occur in the forward pass of an LSTM.

$$
\begin{aligned}
\mathbf{f}_t &= \varphi_{sigmoid}(\mathbf{W}^{(f)} \cdot \mathbf{hx}_t) \\
\mathbf{c}\ddagger &= \mathbf{c}_{t-1} \odot \mathbf{f}_t \\
\mathbf{i}\dagger_t &= \varphi_{sigmoid}(\mathbf{W}^{(i\dagger)} \cdot \mathbf{hx}_t) \\
\mathbf{i}\ddagger_t &= \varphi_{tanh}(\mathbf{W}^{(i\ddagger)} \cdot \mathbf{hx}_t) \\
\mathbf{i}_t &= \mathbf{i}\dagger_t \odot \mathbf{i}\ddagger_t \\
\mathbf{c}_t &= \mathbf{c}\ddagger + \mathbf{i}_t \\
\mathbf{o}\dagger_t &= \varphi_{sigmoid}(\mathbf{W}^{(o\dagger)} \cdot \mathbf{hx}_t) \\
\mathbf{o}\ddagger_t &= \varphi_{tanh}(\mathbf{c}_t) \\
\mathbf{o}_t &= \mathbf{o}\dagger_t \odot \mathbf{o}\ddagger_t \\
\mathbf{h}_{t+1} &= \mathbf{o}_t
\end{aligned}
$$

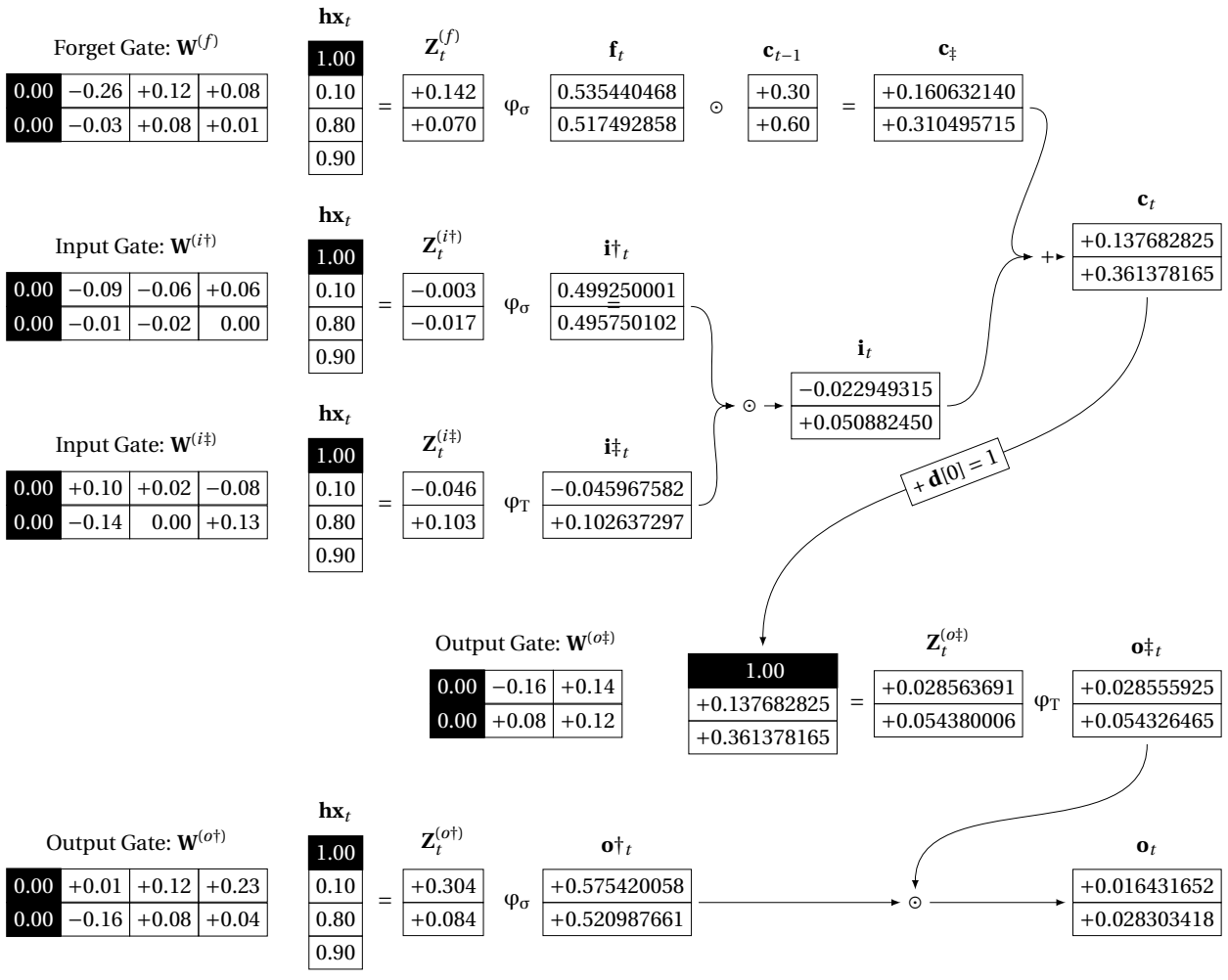


Figure 42: The flow of activations through a long short-term memory unit during forward propagation when $\mathbf{c}_{t-1} = [0.3, 0.6]$, $\mathbf{h}_t = [0.1, 0.8]$, and $\mathbf{x}_t = [0.9]$.

The backpropagation process within an LSTM begins with three vectors of error gradients:

1. $\frac{\partial \mathcal{E}_t}{\partial \mathbf{o}_t}$: the rate of change of the error of the network at time-step t with respect to changes in the activation vector o_t that was propagated to the output layer during the forward pass;
2. $\frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{h}_t}$: the rate of change of the error of the network at time-step $t+1$ with respect to changes in the activation vector h_t that was propagated forward to the next time-step during the forward pass; and
3. $\frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{c}_t}$: the rate of change of the error of the network at time-step $t+1$ with respect to changes in the cell state ct that was propagated forward to the next time-step during the forward pass.

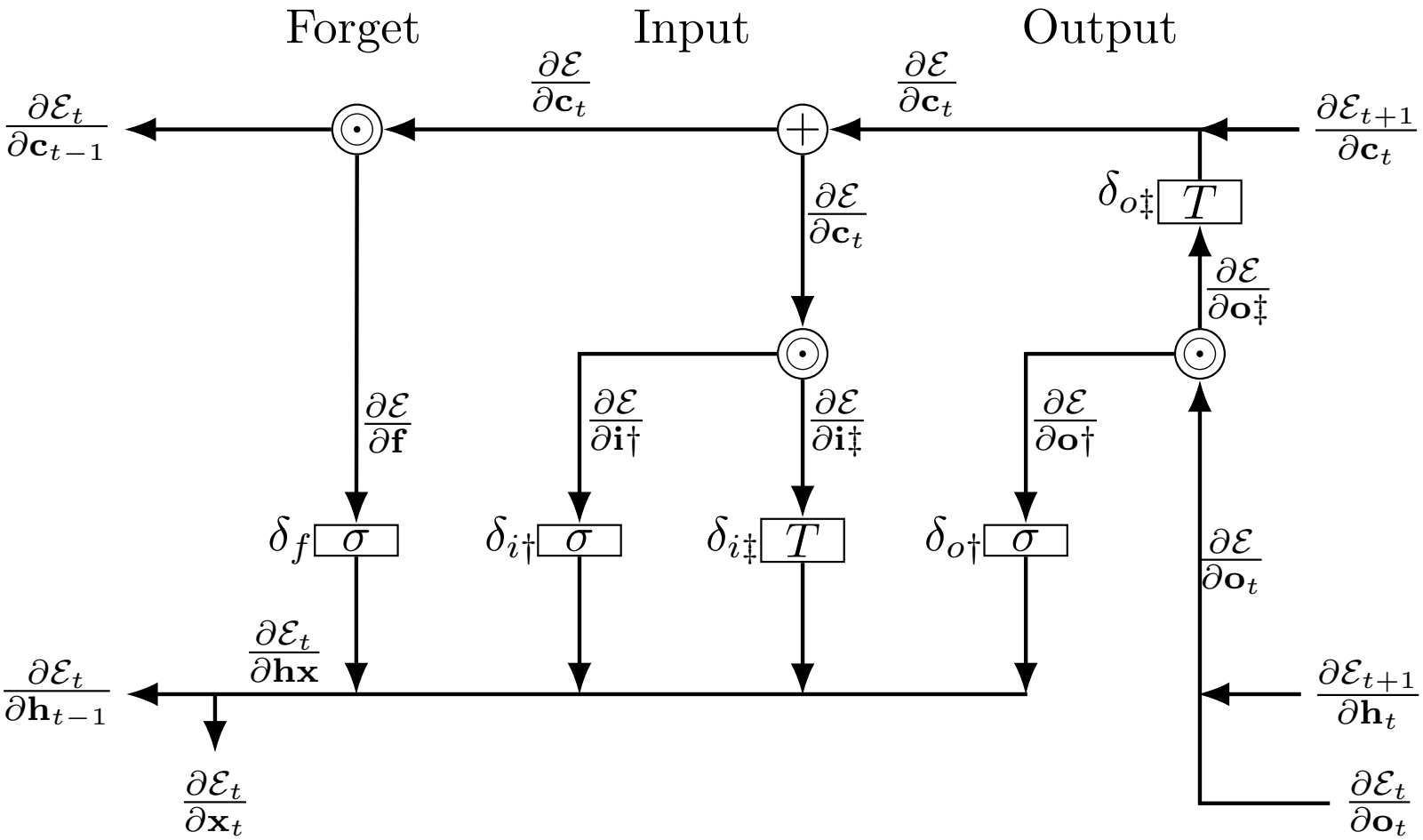


Figure 43: The flow of error gradients through a long short-term memory unit during backpropagation.

- During the forward pass of an LSTM unit, three operations on activation vectors are novel: **forks** in computational flow, **elementwise products** of vectors, and **elementwise addition** of vectors.
- A **fork** occurs twice in an LSTM:
  1. the cell state c_t vector flows forward into the next time-step and is also passed through a $tanh$ layer as part of the output layer; and (2)
  2. the vector of output activations o_t flows forward to both the output layer and the next time-step (as the propagated hidden state h_t).
- Forks in the forward computation flow are handled in backpropagation by summing the derivatives that are flowing back along each of the fork branches.

$$\frac{\partial \mathcal{E}}{\partial \mathbf{o}_t} = \frac{\partial \mathcal{E}_t}{\partial \mathbf{o}_t} + \frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{h}_t} \tag{124}$$

- For an **elementwise product** , we wish to calculate during backpropagation the rate of change of the error with respect to changes in each of the inputs to the product.
- I.e. we will generate two sets of error gradients when we backpropagate through an elementwise vector product, one for each branch of data that flows into the product.
- In backpropagation, the error gradient with respect to an input to a product of two terms is the error gradient with respect to the result of the product multiplied by the other input to the product.

$$\frac{\partial \mathcal{E}}{\partial \mathbf{o}\ddagger} = \frac{\partial \mathcal{E}}{\partial \mathbf{o}_t} \odot \mathbf{o}\dagger \tag{125}$$

$$\frac{\partial \mathcal{E}}{\partial \mathbf{o}\dagger} = \frac{\partial \mathcal{E}}{\partial \mathbf{o}_t} \odot \mathbf{o}\ddagger \tag{126}$$

Elementwise addition of two activation vectors:

- The term $\frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{c}_t}$ describes the vector of error gradients that are backpropagated through this operation.
- The same error gradient vector flows back along both paths that feed into the elementwise summation in the forward path.
- As such, both of the paths emerging from the elementwise summation are labeled with the same term as the input arrow.

To calculate the error gradients we must backpropagate $\frac{\partial \mathcal{E}}{\partial \mathbf{o}^{\ddagger}}$ through a $tanh$ layer and then merge the resulting gradients with the error gradients from the next time-step with respect to the current cell state:

$$\delta_{o^{\ddagger}} = \frac{\partial \mathcal{E}}{\partial \mathbf{o}^{\ddagger}} \odot \frac{\partial \mathbf{o}^{\ddagger}_t}{\partial \mathbf{c}_{t^{\ddagger}}} = \frac{\partial \mathcal{E}}{\partial \mathbf{o}^{\ddagger}} \odot \underbrace{\left(\mathbf{1} - tanh^2(\mathbf{c}^{\ddagger}_t)\right)}_{\text{Derivate of tanh, i.e.: } \frac{\partial a}{\partial z}} \tag{127}$$

$$\frac{\partial \mathcal{E}}{\partial \mathbf{c}_t} = \delta_{o^{\ddagger}} + \frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{c}_t} \tag{128}$$

For the input gate, the gradients for each of the inputs to the elementwise product can now be calculated as follows:

$$\frac{\partial \mathcal{E}}{\partial \mathbf{i}\ddagger} = \frac{\partial \mathcal{E}}{\partial \mathbf{c}_t} \odot \mathbf{i}\dagger \tag{129}$$

$$\frac{\partial \mathcal{E}}{\partial \mathbf{i}\dagger} = \frac{\partial \mathcal{E}}{\partial \mathbf{c}_t} \odot \mathbf{i}\ddagger \tag{130}$$

Similarly, the gradients for the inputs to the elementwise product in the forget gate are calculated:

$$\frac{\partial \mathcal{E}}{\partial \mathbf{c}_{t-1}} = \frac{\partial \mathcal{E}}{\partial \mathbf{c}_t} \odot \mathbf{f}_t \tag{131}$$

$$\frac{\partial \mathcal{E}}{\partial \mathbf{f}} = \frac{\partial \mathcal{E}}{\partial \mathbf{c}_t} \odot \mathbf{c}_{t-1} \tag{132}$$

In order to calculate the δs for the neurons in each of the sigmoid and $tanh$ layers in the LSTM, we must multiply these error gradients by the derivative of the activation function for the layer with respect to the inputs to the activation function (i.e., $\frac{\partial a}{\partial z}$).

$$\delta_f = \frac{\partial \mathcal{E}}{\partial \mathbf{f}} \odot \frac{\partial \mathbf{f}_t}{\partial \mathbf{z}_f} = \frac{\partial \mathcal{E}}{\partial \mathbf{f}} \odot (\mathbf{f}_t \odot (\mathbf{1} - \mathbf{f}_t)) \tag{133}$$

$$\delta_{i\dagger} = \frac{\partial \mathcal{E}}{\partial \mathbf{i}\dagger} \odot \frac{\partial \mathbf{i}\dagger_t}{\partial \mathbf{z}_{i\dagger}} = \frac{\partial \mathcal{E}}{\partial \mathbf{i}\dagger} \odot (\mathbf{i}\dagger_t \odot (\mathbf{1} - \mathbf{i}\dagger_t)) \tag{134}$$

$$\delta_{i\ddagger} = \frac{\partial \mathcal{E}}{\partial \mathbf{i}\ddagger} \odot \frac{\partial \mathbf{i}\ddagger_t}{\partial \mathbf{z}_{i\ddagger}} = \frac{\partial \mathcal{E}}{\partial \mathbf{i}\ddagger} \odot \left(\mathbf{1} - tanh^2(\mathbf{i}\ddagger_t)\right) \tag{135}$$

$$\delta_{o\dagger} = \frac{\partial \mathcal{E}}{\partial \mathbf{o}\dagger} \odot \frac{\partial \mathbf{o}\dagger_t}{\partial \mathbf{z}_{o\dagger}} = \frac{\partial \mathcal{E}}{\partial \mathbf{o}\dagger} \odot (\mathbf{o}\dagger_t \odot (\mathbf{1} - \mathbf{o}\dagger_t)) \tag{136}$$

Once the δs calculated for the neurons in these layers, two further sets of calculations remain to complete the backpropagation through the LSTM.

1. calculate the updates for each weight in each of these layers; and
2. calculate the vector of gradients $\frac{\partial \mathcal{E}_t}{\partial \mathbf{h}_{t-1}}$ that are backpropagated to the previous time-step.

Then each time-step, we calculate the update for each weight by multiplying the δ for the neuron that uses the weight by the input value that weight was applied to, and then we sum these weight updates across the time-steps. The weight is then updated using the summed weight update. E.g.:

$$\Delta \mathbf{W}^{(f)} = \delta_f \cdot \mathbf{h}\mathbf{x}^\intercal \tag{137}$$

With the same example used in the forward pass, let us assume that the following error gradients are already calculated:

$$\frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{c}_t} = \begin{bmatrix} 0.35 \\ 0.50 \end{bmatrix} \quad \frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{h}_t} = \begin{bmatrix} 0.75 \\ 0.25 \end{bmatrix} \quad \frac{\partial \mathcal{E}_t}{\partial \mathbf{o}_t} = \begin{bmatrix} 0.15 \\ 0.60 \end{bmatrix}$$

We proceed with the following calculations (next slides) till we calculate the weight update for the weights in $W^{(f)}$:

$$\frac{\partial \mathcal{E}}{\partial \mathbf{o}_t} = \underbrace{\begin{bmatrix} 0.75 \\ 0.25 \end{bmatrix}}_{\partial \mathcal{E}_{t+1}/\partial \mathbf{h}_t} + \underbrace{\begin{bmatrix} 0.15 \\ 0.60 \end{bmatrix}}_{\partial \mathcal{E}_t/\partial \mathbf{o}_t} = \begin{bmatrix} 0.9 \\ 0.85 \end{bmatrix} \tag{138}$$

$$\frac{\partial \mathcal{E}}{\partial \mathbf{o}\ddagger} = \underbrace{\begin{bmatrix} 0.9 \\ 0.85 \end{bmatrix}}_{\partial \mathcal{E}/\partial \mathbf{o}_t} \odot \underbrace{\begin{bmatrix} 0.575420058 \\ 0.52098661 \end{bmatrix}}_{\mathbf{o}\dagger} = \begin{bmatrix} 0.517878052 \\ 0.442839512 \end{bmatrix} \tag{139}$$

$$\delta_{o\ddagger} = \underbrace{\begin{bmatrix} 0.517878052 \\ 0.442839512 \end{bmatrix}}_{\partial \mathcal{E}/\partial \mathbf{o}\ddagger} \odot \underbrace{\begin{bmatrix} 0.999184559 \\ 0.997048635 \end{bmatrix}}_{1-tanh(\mathbf{c}_t)} = \begin{bmatrix} 0.517455753 \\ 0.441532531 \end{bmatrix} \tag{140}$$

$$\frac{\partial \mathcal{E}}{\partial \mathbf{c}_t} = \underbrace{\begin{bmatrix} 0.517455753 \\ 0.441532531 \end{bmatrix}}_{\delta_{o\ddagger}} + \underbrace{\begin{bmatrix} 0.35 \\ 0.50 \end{bmatrix}}_{\partial \mathcal{E}_{t+1}/\partial \mathbf{c}_t} = \begin{bmatrix} 0.867455753 \\ 0.941532531 \end{bmatrix} \tag{141}$$

$$\frac{\partial \mathcal{E}}{\partial \mathbf{f}} = \underbrace{\begin{bmatrix} 0.867455753 \\ 0.941532531 \end{bmatrix}}_{\partial \mathcal{E}/\partial \mathbf{c}_t} \odot \underbrace{\begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix}}_{\mathbf{c}_{t-1}} = \begin{bmatrix} 0.260236726 \\ 0.564919518 \end{bmatrix} \tag{142}$$

$$\delta_f = \underbrace{\begin{bmatrix} 0.260236726 \\ 0.564919518 \end{bmatrix}}_{\partial \mathcal{E}/\partial \mathbf{f}} \odot \underbrace{\begin{bmatrix} 0.248743973 \\ 0.249694 \end{bmatrix}}_{\mathbf{f}_t \odot (1 - \mathbf{f}_t)} = \begin{bmatrix} 0.064732317 \\ 0.141057014 \end{bmatrix} \tag{143}$$

$$\begin{aligned} \Delta \mathbf{W}^{(f)} &= \underbrace{\begin{bmatrix} 0.064732317 \\ 0.141057014 \end{bmatrix}}_{\delta_f} \cdot \underbrace{\begin{bmatrix} 1.00 & 0.10 & 0.80 & 0.90 \end{bmatrix}}_{\mathbf{h}\mathbf{x}^\intercal} \\ &= \begin{bmatrix} 0.064732317 & 0.006473232 & 0.051785854 & 0.058259085 \\ 0.141057014 & 0.014105701 & 0.112845611 & 0.126951313 \end{bmatrix} \end{aligned} \tag{144}$$

$$\frac{\partial \mathcal{E}}{\partial \mathbf{hx}} = \left(\mathbf{W}^{(f)\intercal} \cdot \delta_f\right) + \left(\mathbf{W}^{(i\dagger)\intercal} \cdot \delta_{i\dagger}\right) + \left(\mathbf{W}^{(i\ddagger)\intercal} \cdot \delta_{i\ddagger}\right) + \left(\mathbf{W}^{(o\dagger)\intercal} \cdot \delta_{o\dagger}\right) \tag{145}$$

$$
\begin{aligned}
\frac{\partial \mathcal{E}}{\partial \mathbf{c}_{t-1}} &= \mathbf{f}_t \odot \frac{\partial \mathcal{E}}{\partial \mathbf{c}_t} \\
&= \mathbf{f}_t \odot \left(\frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{c}_t} + \delta_{o\ddagger} \right) \\
&= \mathbf{f}_t \odot \left(\frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{c}_t} + \left(\left(\mathbf{1} - tanh^2(\mathbf{c}\ddagger_t) \right) \odot \frac{\partial \mathcal{E}}{\partial \mathbf{o}\ddagger} \right) \right) \\
&= \mathbf{f}_t \odot \left(\frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{c}_t} + \left(\left(\mathbf{1} - tanh^2(\mathbf{c}\ddagger_t) \right) \odot \left(\mathbf{o}\dagger \odot \frac{\partial \mathcal{E}}{\partial \mathbf{o}_t} \right) \right) \right) \\
&= \mathbf{f}_t \odot \left(\frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{c}_t} + \left(\left(\mathbf{1} - tanh^2(\mathbf{c}\ddagger_t) \right) \odot \left(\mathbf{o}\dagger \odot \left(\frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{h}_t} + \frac{\partial \mathcal{E}_t}{\partial \mathbf{o}_t} \right) \right) \right) \right)
\end{aligned}
\tag{146}
$$

Summary

- Deep neural networks to learn and represent complex mappings from inputs to outputs
- The standard algorithm for training a deep neural network combines the backpropagation algorithm
- Unstable gradients (either vanishing or exploding gradients) can make training a deep network with backpropagation and gradient descent difficult
  - Activation Functions (ReLUs)
  - Weight Initialization
- Dropout is a very simple and effective method that helps to stop overfitting.
- We can tailor the structure of a network toward the characteristics of the data
  - Convolutional Neural Networks
  - Recurrent Neural Networks

Further Reading

- Other texts on neural networks and deep learning: (**????**)
- Programming focused introductions to deep learning: (**??**)
- Introduction to neural networks for natural language processing: (**?**)
- Computer architecture perspective on deep learning: (**?**)
- Recent developments in the field: **batch normalization** (**?**), adaptively learning algorithms such as **Adam** (**?**), **Generative Adversarial Networks** (**?**), and attention-based architectures such as the **Transformer** (**?**).

Slide Acknowledgment

The slides used in this course are based on the official textbook materials, with modifications made where necessary to suit the course requirements and enhance the learning experience.