Artificial Neural Networks and Deep Learning (Part II)

CS576 MACHINE LEARNING

Instructor: Dr. Jin S. Yoo
Department of Computer Science
Purdue University Fort Wayne

Reference

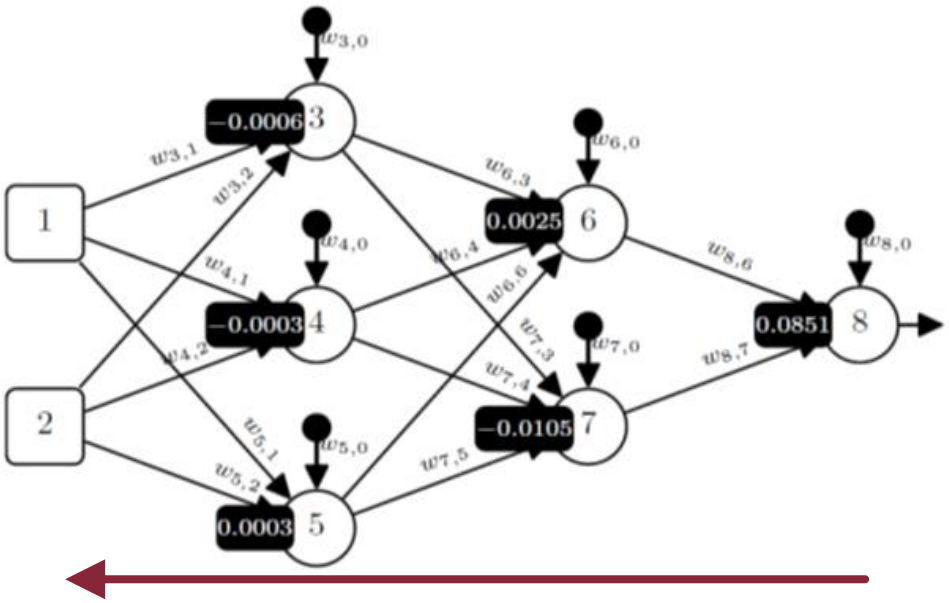
- Kelleher et al., Fundamentals of Machine Learning (2nd edition), Ch 8

Part II Outline

- Extensions and Variations
  - Vanishing Gradients and ReLUs
  - Weight Initialization and Unstable Gradients
  - Handling Categorical Target Feature
  - Early Stopping and Dropout: Preventing Overfitting

Challenge: Vanishing Gradient

- A significant challenge in training deep neural networks is maintaining a consist flow of error gradients (δs), as they are backpropagated through the layers during training.
- Error gradients are calculated by propagating the network error backward. These values tend to decrease as they progress from the output layer to the earlier layers in the network.
- This issue is commonly known as the **vanishing gradient** problem.



The error gradient δ values become smaller and smaller as it is backpropagated through the networks layers

Cause of Vanishing Gradient

- The error gradient at any point in the network is a result of the cumulative effect of all the factors involved in the backpropagation process.
- It's a chain of gradients, where each layer's gradient is influenced by the gradients of subsequent layers

Example: *i* feeds forward into *j*, and *j* into *k*

$$\delta_i = \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i}$$

$$= w_{j,i} \times \delta_j \times \frac{\partial a_i}{\partial z_i}$$

$$= w_{j,i} \times \frac{\partial \mathcal{E}}{\partial a_j} \times \frac{\partial a_j}{\partial z_j} \times \frac{\partial a_i}{\partial z_i}$$

$$= w_{j,i} \times w_{k,j} \times \delta_k \times \frac{\partial a_j}{\partial z_j} \times \frac{\partial a_i}{\partial z_i}$$

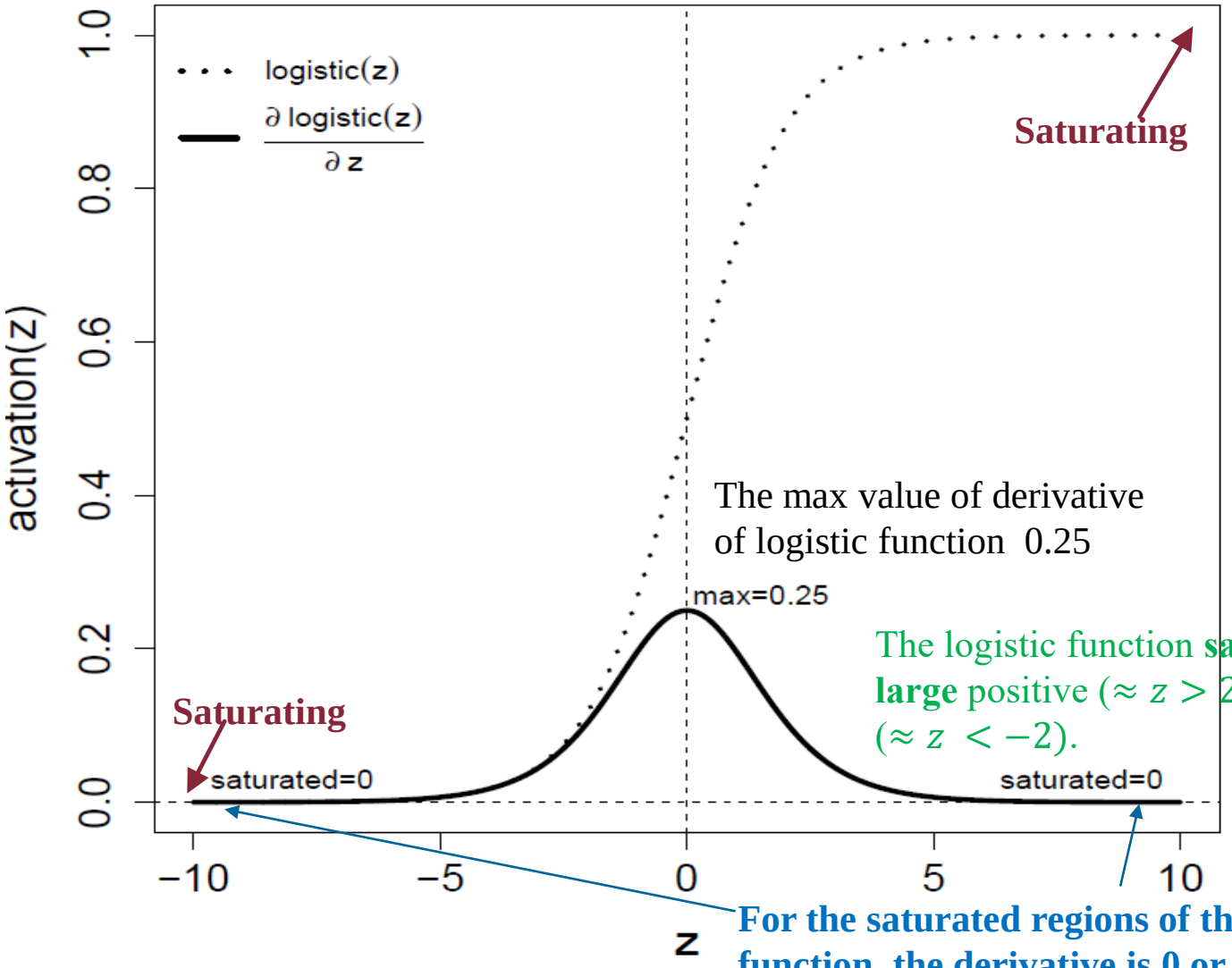
$$= w_{j,i} \times w_{k,j} \times \frac{\partial \mathcal{E}}{\partial a_k} \times \frac{\partial a_k}{\partial z_k} \times \frac{\partial a_j}{\partial z_j} \times \frac{\partial a_i}{\partial z_i}$$

- To calculate δ, a sequence of products is computed by multiplying weights, error gradients, and derivatives of the activation function iteratively.
- This process may result in a reduction in the error gradient value.
  - E.g., The derivative value of a logistic activation function is less than 1 (a maximum of 0.25).

Saturated Region of Activation Function

- In the context of activation functions in neural networks, a "saturated region" refers to a specific range of inputs for which the activation function produces nearly constant or very small outputs.
- For the saturated regions of the activation function, the derivative (error gradient) is either 0 or close to 0.
  - E.g., The logistic function saturates when z is significantly negative ($\approx z < -2$) or significantly positive ($\approx z > 2$). In these regions, the error gradient for the z values is roughly 0.
- The diminishing of the error gradient is especially noticeable for neurons with z values situated in the saturated region of the activation function.

Logistic Activation Function Saturation



Vanishing Gradient Problem

- Additionally, the weight updates are determined by the value of δ, as indicated by the weight update rule:
  $w_{i,k} \leftarrow w_{i,k} - \alpha \times \delta_i \times a_k$
- As the δ values decrease while the network error propagate backward, the weights in the network's earlier layers receive relatively smaller updates compared to those in the later layers.
- This phenomenon can result in extended training times for deep networks.

How to Avoid the Vanishing Gradient Issue

- To address the vanishing gradient problem, an alternative activation function can be employed, such as the Rectified Linear Unit (ReUL), defined as:

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

  - The derivative of the rectifier function is:

$$\frac{\partial}{\partial z} rectifier(z) = \begin{cases} 1 & if\ z > 0 \\ 0 & otherwise \end{cases}$$

    - Thanks to the derivative being equal to 1 when z is greater than 0 during the backward propagation phase, gradients can effectively propagate through deeper layers of the network.

Example

- A training dataset

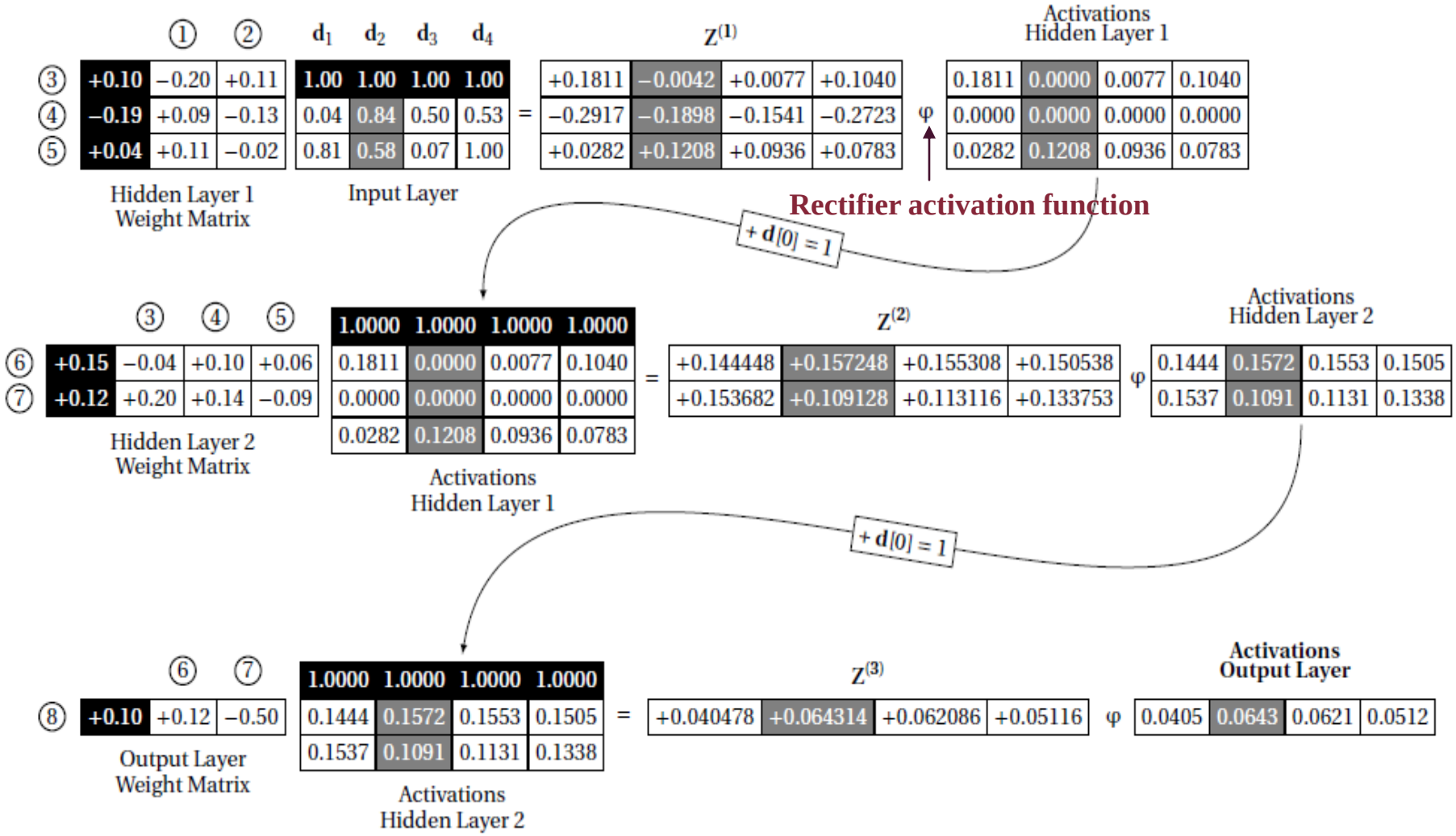
Hourly samples of ambient factors and full load electrical power output of a combined cycle power plant.

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

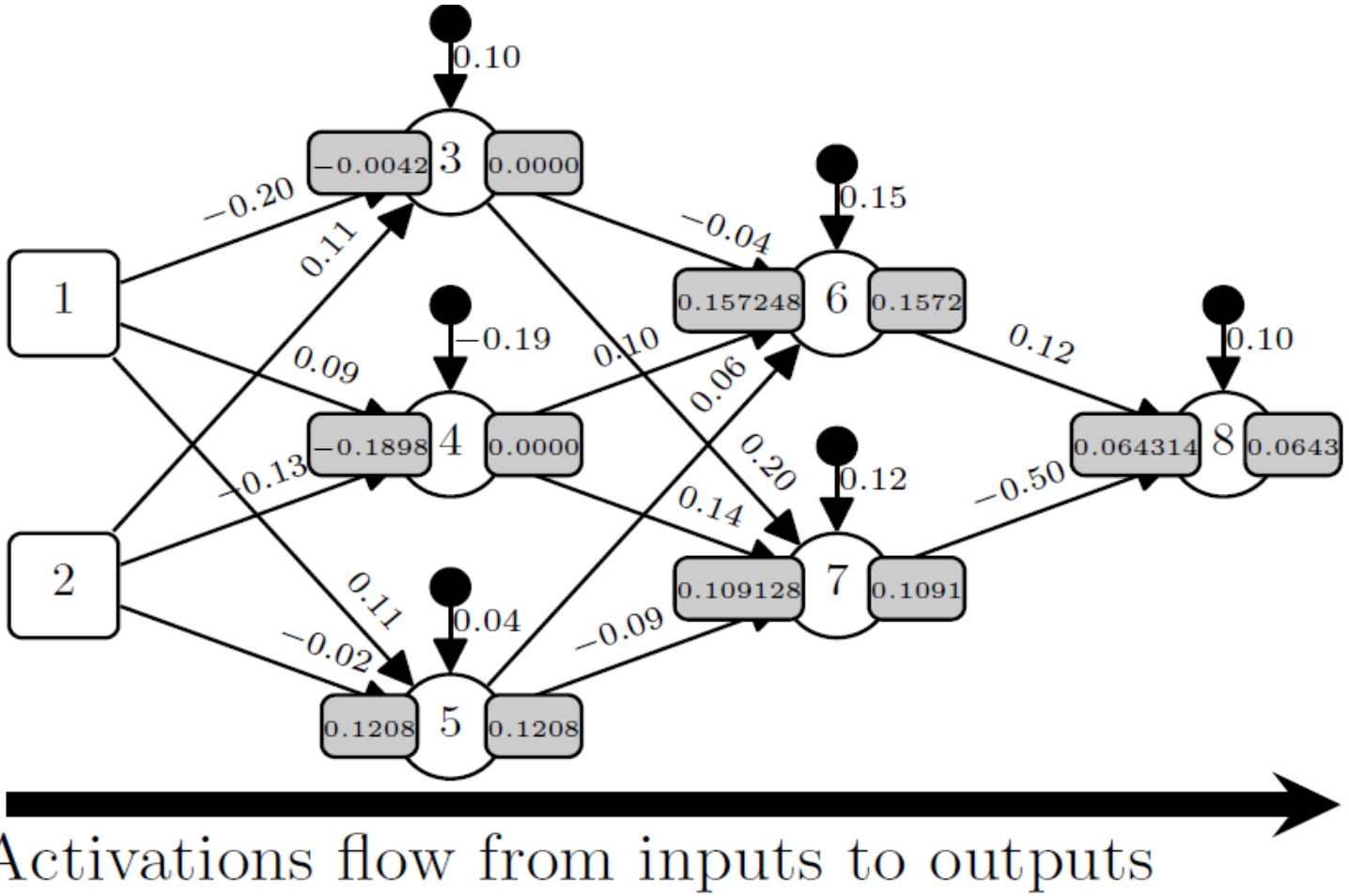
- The range-normalized data in [0,1]

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

The Forward Pass of Network with ReLU neurons



The Forward Propagation of $\mathbf{d}_2$ through the ReLU network



Each Example Error after the Forward Pass

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

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

- This table shows the error per training sample for the ReLU network after the forward pass, including individual derivative with respect to the activation of the output node (neuron 8), and the overall error of the network with SSE.

Calculation of Error Gradient δs through Backward Pass

Error gradient $\boldsymbol{\delta}$ of output node k

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

- Error gradient δ of output node 8 for d_2
  - $\frac{\partial \mathcal{E}}{\partial a_8} = \text{Network Error} \times -1 = -(t_k - a_k) = -0.0657$
  - $\delta_8 = \frac{\partial \mathcal{E}}{\partial a_8} \times \frac{\partial a_8}{\partial z_8}$

    $= -0.0657 \times 1 = -0.0657$

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

Table: The $\frac{\partial a_k}{\partial z_k}$ s computed during the forward pass

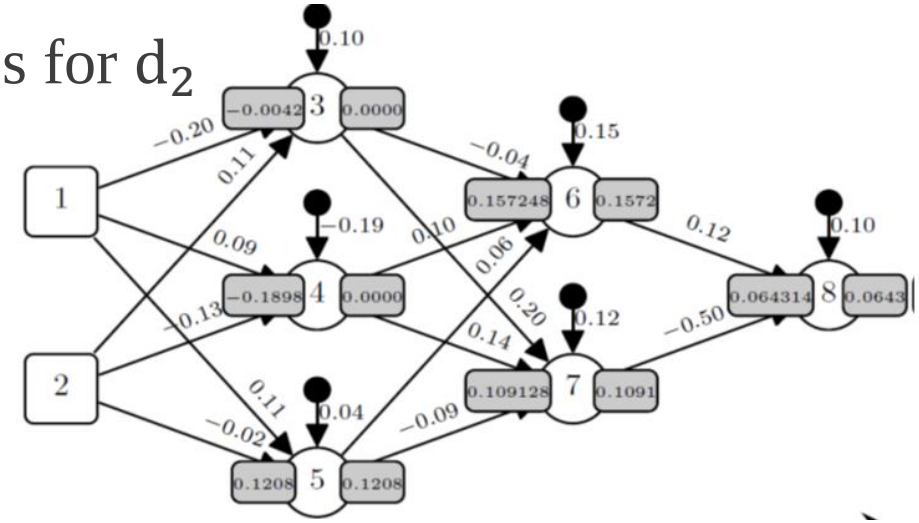
$$\frac{\partial a_k}{\partial z_k} = \frac{\partial}{\partial z} rectifier(z) = \begin{cases} 1 & if\ z > 0 \\ 0 & otherwise \end{cases}$$

Calculation of Error Gradient δs through Backward Pass

Error gradient $\boldsymbol{\delta}$ of hidden node k

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

- Error gradient δ of hidden nodes for d_2
  - $\delta_7 = \frac{\partial a_7}{\partial z_7} \times (\delta_8 \times w_{8,7})$
    $= 1 \times (-0.0657 \times -0.5)$
    $= 0.0329$
  - $\delta_6 = \frac{\partial a_6}{\partial z_6} \times (\delta_8 \times w_{8,6})$
    $= 1 \times (-0.0657 \times -0.12)$
    $= -0.0079$



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

Calculation of Error Gradient δs through Backward Pass

- Error gradient δ of hidden nodes for d_2 (cont.)
  - $\delta_5 = \frac{\partial a_5}{\partial z_5} \times \left((\delta_6 \times w_{6,5}) + (\delta_7 \times w_{7,5}) \right)$

    $= 1 \times ((-0.0079 \times 0.06) + (0.0329 \times -0.09))$

    $=$ -0.0034
  - $\delta_4 = \frac{\partial a_4}{\partial z_4} \times \left((\delta_6 \times w_{6,4}) + (\delta_7 \times w_{7,4}) \right)$

    $= 0 \times ((-0.0079 \times 0.1) +$

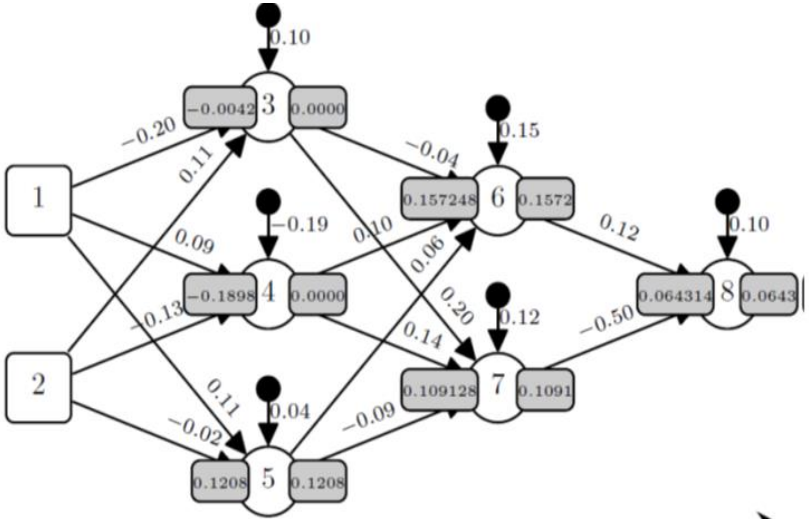
    $(0.0329 \times 0.14))$

    $= 0$
  - $\delta_3 = \frac{\partial a_3}{\partial z_3} \times \left((\delta_6 \times w_{6,3}) + (\delta_7 \times w_{7,3}) \right)$

    $= 0 \times ((-0.0079 \times -0.04) +$

    $(0.0329 \times 0.2))$

    $= 0$



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

Comparisons of $\frac{\partial a_k}{\partial z_k}$ and δs

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

Table: The **ReLU** network

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

Table: The **Logistic** network

Error Gradient	**the ReLU network**	**the Logistic network**
$\boldsymbol{\delta_8}$	-0.0657	0.0852
$\boldsymbol{\delta_7}$	0.0329	-0.0105
$\boldsymbol{\delta_6}$	-0.0079	0.0025
$\boldsymbol{\delta_5}$	-0.0034	0.0003
$\boldsymbol{\delta_4}$	0	-0.0003
$\boldsymbol{\delta_3}$	0	-0.0006

In computation of $\delta_i = \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial z_i} = w_{i,j} \times w_{j,k} \times \cdots \times \frac{\partial a_i}{\partial z_i} \times \frac{\partial a_j}{\partial z_j} \times \frac{\partial a_k}{\partial z_k} \times \frac{\partial a_k}{\partial z_k} \times \cdots \times \frac{\partial \mathcal{E}}{\partial a_k}$

, the derivative of the ReLU has not pushed to the δ to 0.

Error After Training has Converged

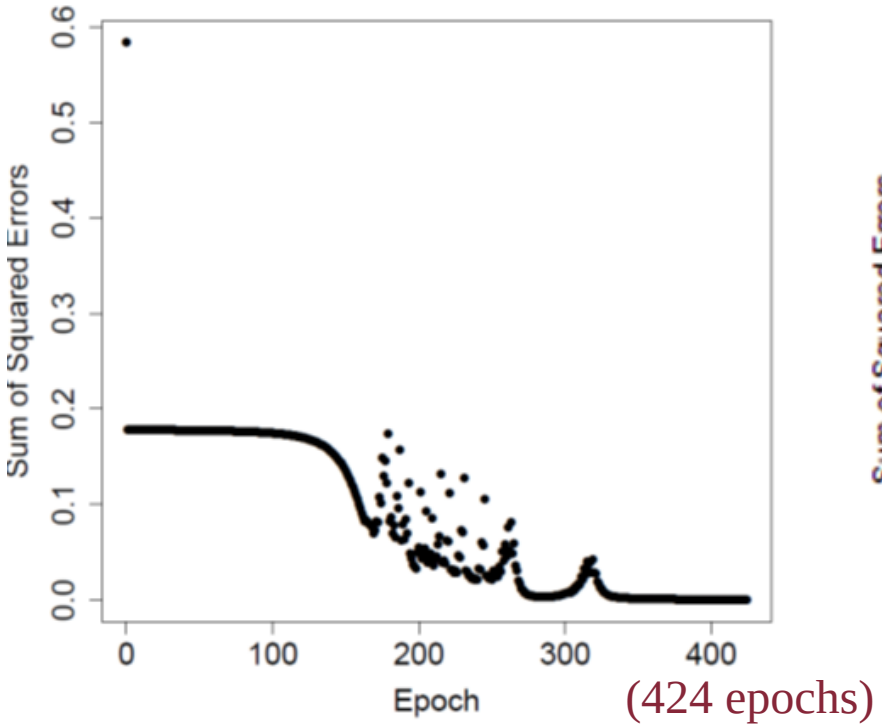
- The **ReLU network**'s per example prediction, error, and the sum of squared errors after training has converged to an *SSE* < 0.0001 (after 424 epochs)

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

- The **Logistic network**'s per example prediction, error, and the sum of squared errors after training has converged to an *SSE* < 0.0001 (after 7767 epochs)

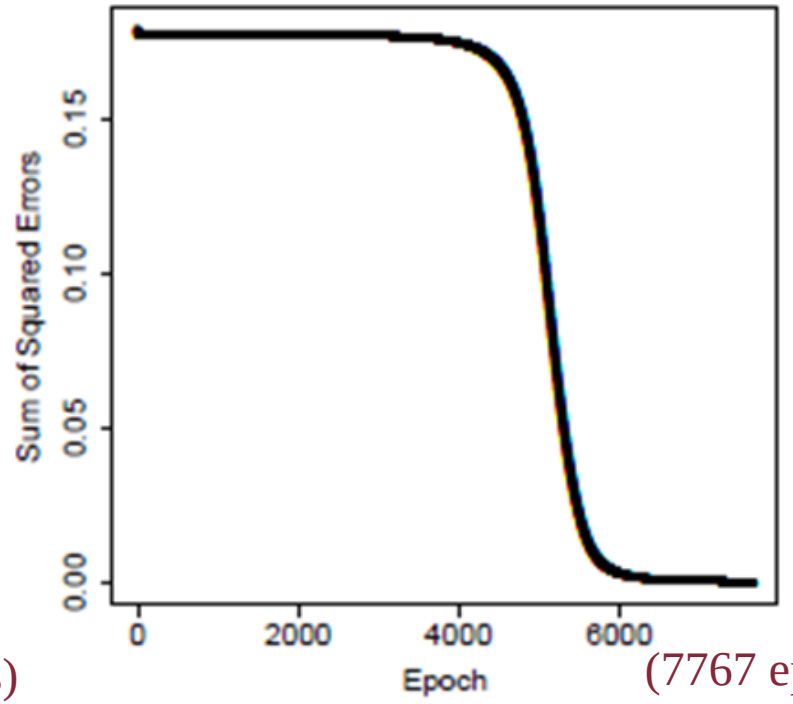
	$\mathbf{d}_1$	$\mathbf{d}_2$	$\mathbf{d}_2$	$\mathbf{d}_2$
Target	0.9400	0.1300	0.5700	0.3600
Prediction	0.9266	0.1342	0.5700	0.3608
Error	0.0134	-0.0042	0.0000	-0.0008
Error^2	0.00017956	0.00001764	0.00000000	0.00000064
SSE:				0.00009892

Speed Up in Training



(424 epochs)

Figure: The **ReLU** network



(7767 epochs)

Figure: The **Logistic** network

Outline

- Vanishing Gradients and ReLUs
- ☞ **Dying ReLUs and ReLU's variants**
- Weight Initialization and Unstable Gradients
- Handling Categorical Target Feature
- Early Stopping and Dropout: Preventing Overfitting

Dying **ReLU**

- In the ReLU network, only a subset of neurons will activate (i.e., $a_i > 0$) due to the nature of the rectified linear function, which results in a zero output for all $z < 0$.
- The issue where a considerable number of neurons in the ReLU network do not activate is referred to as the **dying ReLU proble**.

Problems of Dying **ReLU**

- In the dying ReLU scenario, a neuron remains unresponsive for all training samples, leading to a persistent lack of activity.
- This characteristic leads to a **sparse representation** learned by the network.
  - In contrast, a Logistic network typically has the majority of neurons active for all inputs.
- Excessive sparsity in the network can reduce its capacity to represent information effectively, potentially **resulting in decreased performance.**
- A large number of non-active neurons can hinder the network's convergence during training.

Heuristic to Avoid Dead ReLUs

- **To prevent dead ReLUs,**
  1. Initiate all bias weights of a network with small positive values, such as 0.1.
  2. Modify the rectified linear functions to prevent saturation for $z < 0$
     - **Case 1**: Use the Leaky Rectified Linear Unit (**Leaky ReLU**)

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

       - Ensure that the derivative required for backpropagating a δ through the Leaky ReLU function is computed accordingly.

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

Heuristic to Avoid Dead ReLUs (Cont.)

- **Case 2**: Parametric Rectified Linear Unit (**Parametric ReLU**)

$$rectifier_{parametric}(z_i) = \begin{cases} z_i & \text{if } z_i > 0 \\ \lambda_i z_i & \text{if } z_i \leq 0 \end{cases}$$

  - The derivative used for backpropagating a δ through the $rectifier_{parametric}$ function is not a constant, predefined value for $z_i \leq 0$. Instead, each neuron independently learns its gradient for the activation function PReLU_i when $z \leq 0$.

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

    - The adjustment of λ is a proportionate to the error gradient of the network concerning variations in this parameter, much like how weights within the network are adapted during training

$$\lambda_i \leftarrow \lambda_i - \alpha \times \frac{\partial \mathcal{E}}{\partial \lambda_i} \qquad \frac{\partial \mathcal{E}}{\partial \lambda_i} = \frac{\partial \mathcal{E}}{\partial a_i} \times \frac{\partial a_i}{\partial \lambda_i}, \quad \frac{\partial a_i}{\partial \lambda_i} = \begin{cases} 0 & \text{if } z_i > 0 \\ z_i & otherwise \end{cases}$$

Summary

- Historically, the **logistic function** was the primary choice for activation function in neural networks, largely due to its straightforward derivative.
- However, the logistic function's derivative, which has a maximum value of 0.25, presents challenges by contributing to the **vanishing gradient problem** in neural networks.
- In response to these challenges, contemporary researchers often prefer the **rectifier function** (**or its variants**) as the standard activation function.
- Like the logistic function, the rectifier function also encounters **saturation** in specific regions, resulting in the "**dying ReLU**" phenomenon. This situation arises when a neuron remains consistently inactive across all training instances, rendering it perpetually non-functional.

Summary (cont.)

- To address this issue, alternative variants such as the **leaky ReLU** and **parametric ReLU** were introduced.
- The choice of ReLU variants often depends on the specific network architecture and the nature of the task at hand.
- An effective heuristic to prevent saturation in the rectifier function is initializing bias weights with small positive values, typically 0.1.
- This underlines the intricate relationship between weight initialization and the propagation of error gradients during backpropagation.
- In determining two important hyperparameters in neural networks: **(1) the activation function** and **(2) the initialization of the** weights , for keeping error gradients stable.
  - Like many hyperparameters, **the optimal choice often emerges through experimentation and fine-tuning in various experiments.**

Outline

- Vanishing Gradients and ReLUs
- Dying ReLUs and ReLU's variants
- ☞ **Weight Initialization and Unstable Gradients**
- Handling Categorical Target Feature
- Early Stopping and Dropout: Preventing Overfitting

Weight Initialization

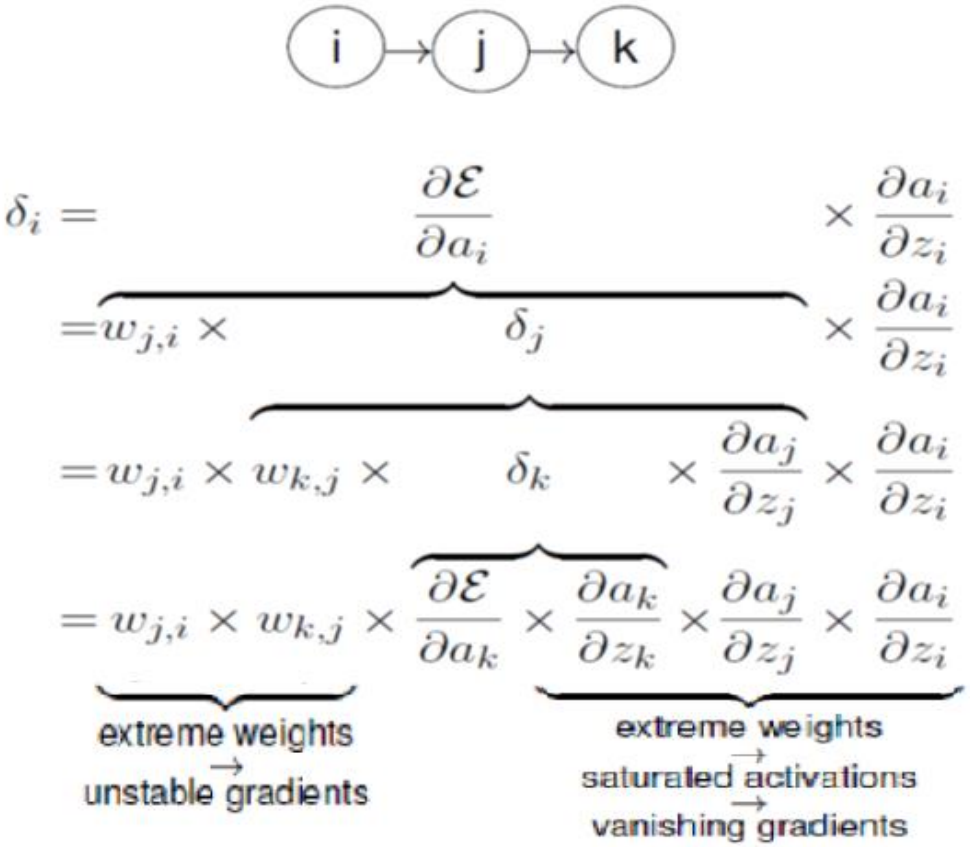
- In the previous example, both bias terms and weights were randomly selected from a uniform distribution within the range of [-0.5, +0.5].
- However, assigning weights without thoughtful consideration can result in problems such as vanishing gradients and unresponsive neurons.

Issues of Weight Initialization

- **[Problem 1]** Neuron connections with excessively large or very small weights can lead to instability during training.
  - When the weights become too high or too low, the resultant z value for the neuron can become excessively large or small.
  - Consequently, the neuron's activation might reach its saturation point.
    - The logistic function saturates if $z > 2$ or $z < -2$.
    - The rectified linear function saturates for significantly negative *z* values ($z < 0$).
  - Neurons with saturated activation functions are at risk of becoming stagnant because their weights undergo minimal changes during training. Updates are either negligible or non-existent, causing them to learn slowly or not learn at all.
- **To prevent saturation during initialization and avoid excessively large or small z values, it's advisable to set the initial weights close to 0.**

Issues of Weight Initialization (cont.)

- **[Problem 2]** Extremely small or large weight values can result in gradient instability.
  - When gradients decrease to the point of becoming nearly negligible, it's referred to as the **vanishing gradient problem**.
  - Conversely, when gradients become excessively large, it leads to the **exploding gradient issue**.



Here, the error term $\frac{\partial \mathcal{E}}{\partial a_k}$ is multiplied by two weights $w_{j,i}$ and $w_{k,j}$

The problems of Weight Initialization

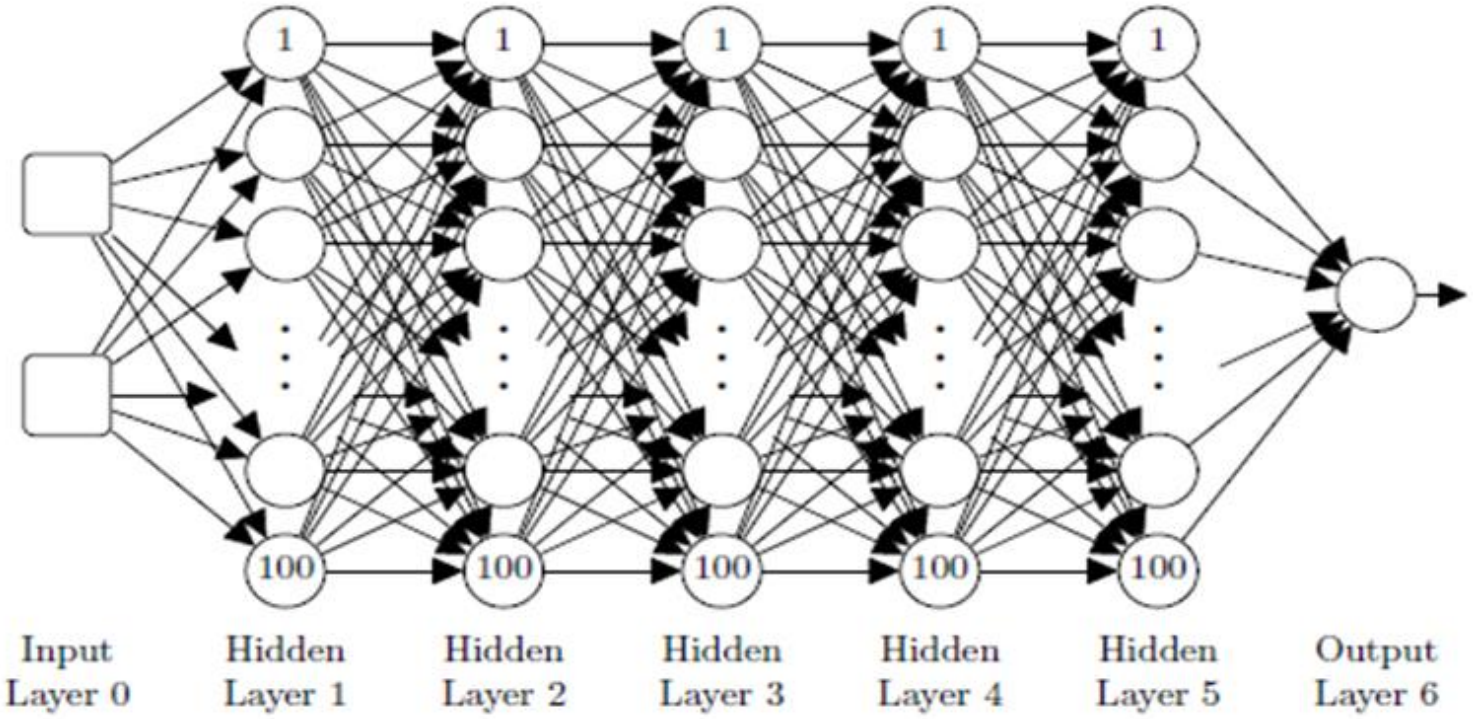
- **[Problem 3]** The variability of z (resulting from a weighted sum) is influenced by the number of inputs to this sum, the variance in these inputs, and the variance of the weights.
  - When there's a mismatch between the number of inputs and the variances of the weights, the output of the weighted sum can exhibit significant deviations, having either higher or lower variance than its inputs.
  - These disparities can result in unpredictable behaviors during both the forward and backward passes of backpropagation.

How to Initialize the Weights

How should we initialize the weights?

- Weight initialization methods often incorporate heuristics to avoid weights that are excessively large or small.
- Typically, bias terms are initialized to 0.
  - In the case of ReLUs, a small positive value like 0.1 is frequently employed.
- Historically, a prevalent approach for neural network weight initialization was to randomly sample values from a normal or uniform distribution centered around 0 (e.g., μ=0 and σ=0.01).
  - Modifying the variance of the normal distribution, allow us to control the initial scale of the weights.

Weight Initialization Experiment

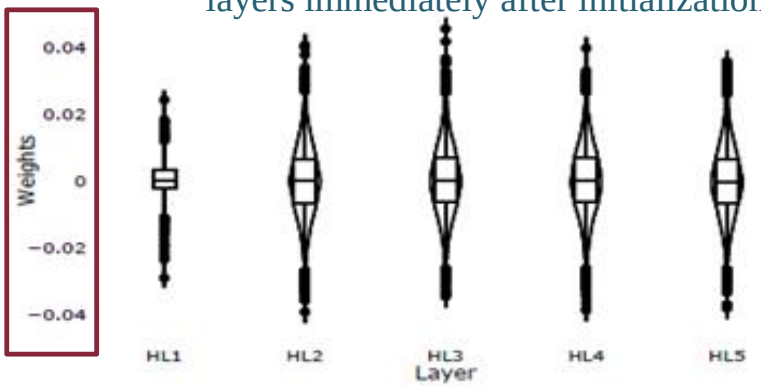


Network Architecture:

- The neurons in this network employee a linear activation function: $a_i = z_i$.
- The derivative of this activation function with respect to z is always 1. Each unit change in the value of z_i results in a unit change in the value of a_i. So the gradients in this network will not be affected by saturated activation functions.
- For the experiment, a sample of 100 examples has been standardized with a mean of 0 and a standard deviation of 1.

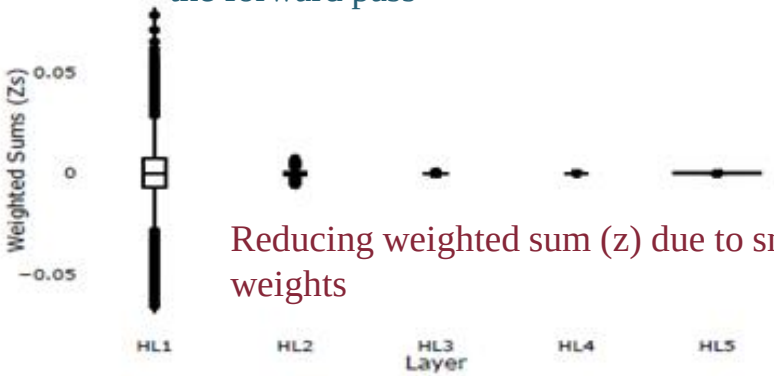
Experiment with Smaller Initial Weights

The distribution of weight values across layers immediately after initialization



(a) Weights by Layer

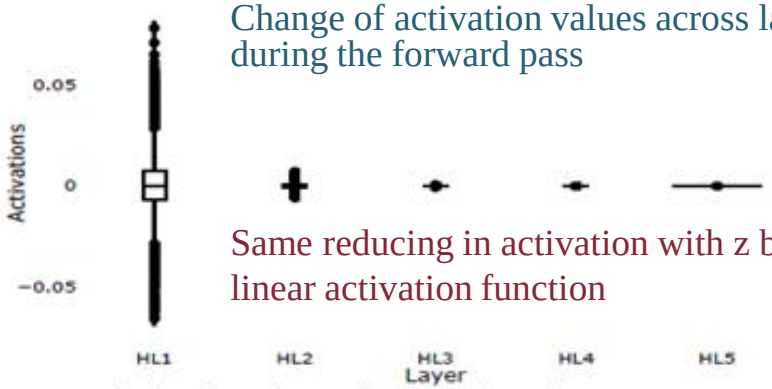
Change of z values across layers during the forward pass



Reducing weighted sum (z) due to small weights

(b) Weighted Sum (z) by Layer

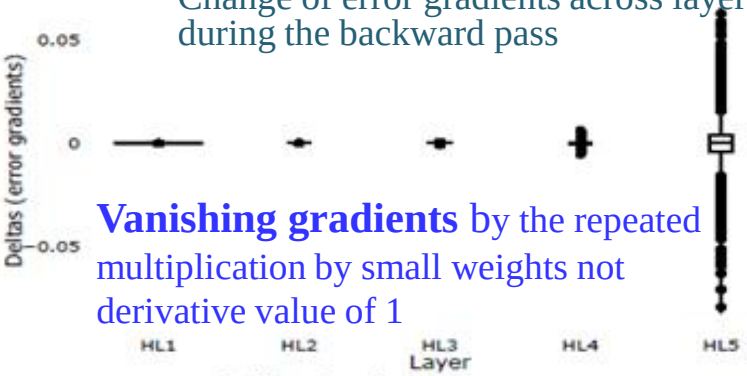
Change of activation values across layers during the forward pass



Same reducing in activation with z by the linear activation function

(c) Activations by Layer

Change of error gradients across layers during the backward pass



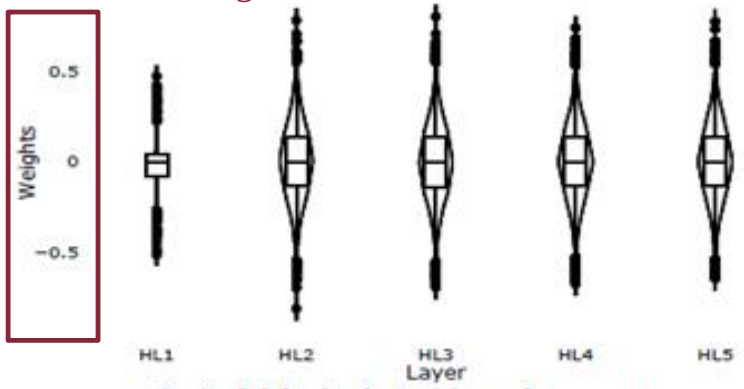
Vanishing gradients by the repeated multiplication by small weights not derivative value of 1

(d) δs by Layer

Figure. The internal dynamics of the network of **linear activation function** $a_i = z_i$ with the distribution of **a network property (weights, weighted sums, activation, or δs)** during the first training iteration when the weights were initialized using a normal distribution with μ=0.0, ,σ=0.01.

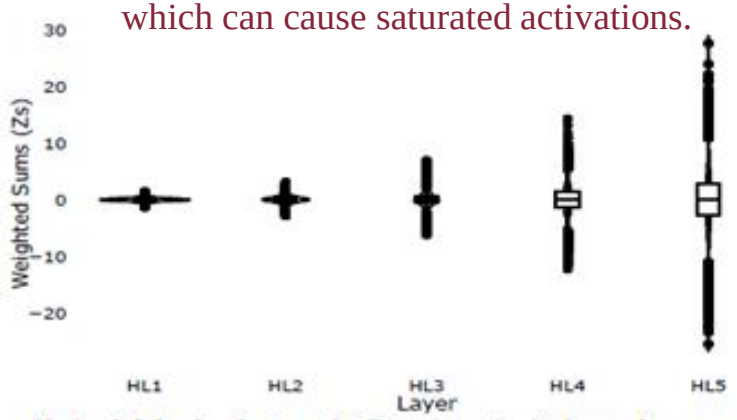
Experiment with Larger Initial Weights

a larger variance due to larger weights from the median 0

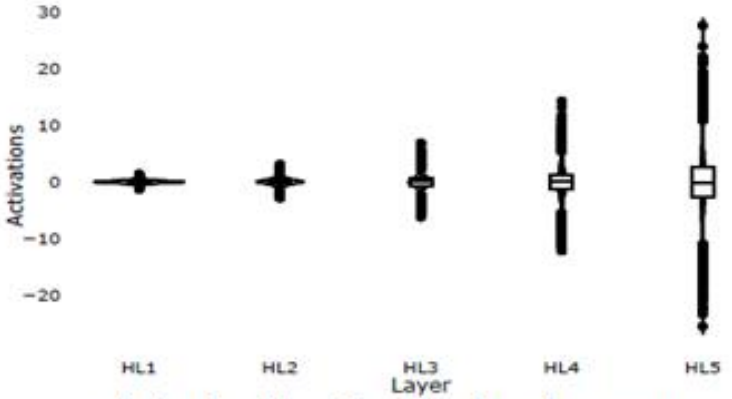


(a) Weights by Layer

Increasing (exploding) z due to larger weights which can cause saturated activations.

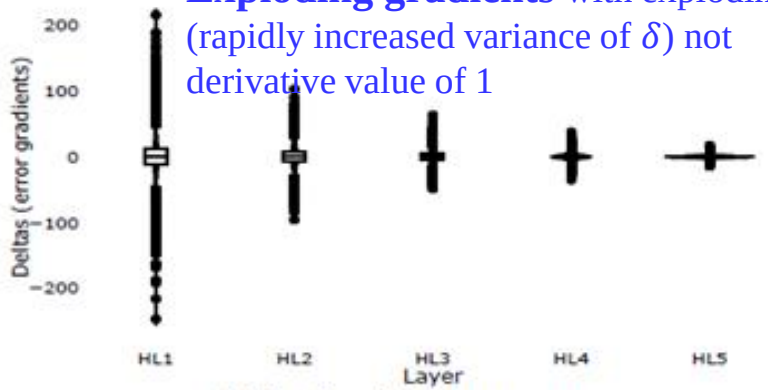


(b) Weighted Sum (z) by Layer



(c) Activations by Layer

Exploding gradients with exploding z (rapidly increased variance of δ) not derivative value of 1



(d) δs by Layer

Figure The internal dynamics of the network with linear activation function $a_i = z_i$ during the first training iteration when the weights were initialized using a normal distribution with μ=0.0, ,σ=0.2.

Relationship of Weights and Weighted Sum (z)

- The relationship between the variance of z for a single neuron in the first hidden layer of a network and the variance of the weights used in calculating $z = (w_1 \times d_1) + (w_2 \times d_2) + \cdots + (w_{n_{in}} \times d_{n_{in}})$ is as follows:

$$\boldsymbol{var(z)} = \sum_{i=1}^{n_{in}} var(\, w_i \times d_i)$$

$$\boldsymbol{var(z)} = n_{in} \times var(\mathbf{W}) \times var(\mathbf{d})$$

The valiance of z is equal to the variance of the inputs ($var(\mathbf{d})$) multiplied by $\boldsymbol{n_{in} \times var(\mathbf{W})}$.

- n_{in} is the number of inputs
- It is assume that each weight w_i is independent of the corresponding input d_i
- $var(w \times d) = [E(\mathbf{W})]^2 var(\mathbf{d}) + [E(\mathbf{d})]^2\, var(\mathbf{W}) + var(\mathbf{W})\, var(\mathbf{d})$
  $= var(\mathbf{W})\, var(\mathbf{d})$
  , where $E(\mathbf{W})$ = for weights from a distribution with mean 0.

Relationship between Weights and Z

- Architecture of the experimental network
  - A linear activation function is employed and the weights are sampled from $N(0, 0.01)$
  - The number of input for the neurons in the first hidden layer, $n_{in}^{(HL1)}$, is 2 and all the other hidden layers have n_{in}=100 per layer.
  - The input data has been standardized with a mean of 0 and a standard deviation of 1.
- **The variances of hidden layers**
  - In all the layers, the variance of the weights, $var(\mathbf{W}^{(HLk)})$, $\sigma^2 = 0.01^2 = 0.0001$

$$var(Z^{(HL1)}) = n_{in}^{(HL1)} \times var(\mathbf{W}^{(HL1)}) \times var(\mathbf{d}^{(HL1)})$$
$$= 2 \times 0.0001 \times 1 = \mathbf{0.0002}$$

because $var(Z^{(HL1)}) = var(\mathbf{A}^{(HL1)}) = var(\mathbf{d}^{(HL2)})$

By the linear activation and the output of activation in *HL1* is the input of *HL2*

$$var(Z^{(HL2)}) = n_{in}^{(HL2)} \times var(\mathbf{W}^{(HL2)}) \times var(\mathbf{d}^{(HL2)})$$
$$= \mathbf{100} \times \mathbf{0.0001} \times \mathbf{0.0002} = 0.000002$$

$$\dots$$

$$var(Z^{(HLk)}) = n_{in}^{(HLk)} \times var(\mathbf{W}^{(HLk)}) \times var(Z^{(HLk-1)})$$

In this network, because of the use of linear activations, the variance of z for layer k in the network is the variance of z in the preceding layer, $var(Z^{(k-1)})$, scaled by $n_{in}^{(k)} \times var(\mathrm{W}^{(k)})$

Relationship between Weights and Z (cont.)

- This means that the multiplication by a small value, represented by $n_{in}^{(k)} \times var(\mathrm{W}^{(k)})$, causes a significant decrease in the variance of z as we move through each layer of the network. (In this case, 0.0002 for the first hidden layer and 0.01 (=100×0.0001) for subsequent layers)

Relationship between Weights and Z (cont.)

- In a network with linear activation function and weights sampled from $N(0, 0.2)$, the number of input for the neurons in the first hidden layer, $n_{in}^{(HL1)}$ is 2. For neurons in all the other hidden layers, n_{in}=100. In all the layers, the variance of the weights, $var(\mathbf{W}^{(HLk)})$, $\sigma^2 = 0.2^2 = 0.04$

$$var(Z^{(HL1)}) = n_{in}^{(HL1)} \times var(\mathbf{W}^{(HL1)}) \times var(\mathbf{d}^{(HL1)})$$
$$= 2 \times 0.04 \times 1 = 0.08$$
$$var(Z^{(HL2)}) = n_{in}^{(HL2)} \times var(\mathbf{W}^{(HL2)}) \times var(\mathbf{d}^{(HL2)})$$
$$= 100 \times 0.04 \times 0.08 = 0.32$$
$$\dots$$
$$var(Z^{(k)}) = n_{in}^{(k)} \times var(\mathbf{W}^{(k)}) \times var(Z^{(k-1)})$$

➔ This implies that the multiplication by a large value, represented by $n_{in}^{(k)} \times var(\mathbf{W}^{(k)})$, causes a significant increase in the variance of z as we advance through each layer of the network (In this case, 0.08 for the first hidden layer and 4 (= 100×0.04) for subsequent layers)

Relationship between Weights and Gradient δ

- Similary, the variance of the δ values, when backpropagated through each layer k , depends on the number of outputs from each neuron in that layer, (here $n_{out}^{(k)}$), multiplied by the variance of the weights for that layer $var(\mathbf{W}^{(k)})$.

$$var\left(\delta^{(HLk)}\right) = n_{out}^{(HLk)} \times var(\mathbf{W}^{(HLk)}) \times var(\mathbf{d}^{(HLk)})$$
$$= n_{out}^{(HLk)} \times var(\mathbf{W}^{(HLk)}) \times var(\delta^{(HLk+1)})$$

- E.g., In a network with a linear activation function and weights from $N(0, 0.2)$, if we assume that the variance of the δs backpropagated to a neuron in HL5 is 1 and the variance of the weights in all the layers $var\left(W^{(HLk)}\right)$, is $\sigma^2 = 0.2^2 = 0.04$, it indicates the scale at which changes in δ values occur as they are propagated backward through the layers.

$$var\left(\delta^{(HL4)}\right) = n_{out}^{(HL4)} \times var(W^{(HL4)}) \times var(d^{(HL4)})$$
$$= 100 \times 0.04 \times 1 = \mathbf{4}$$

$$var\left(\delta^{(HL3)}\right) = n_{in}^{(HL3)} \times var(W^{(HL3)}) \times var(d^{(HL3)})$$
$$= 100 \times 0.04 \times \mathbf{4} = 16$$

➔ The variance of δ will increase by a factor of 4.

…

Summary

- The analysis highlights that the variance of output of a weighted sum is dependent on the number of inputs, whether it's n_{in} in forward propagation or n_{out} in backward propagation.
- In a fully connected network, setting $var\left(\mathbf{W}^{(k)}\right) = 1/n_{in}^{(k)}$ ensures that the variance of the z values in layer k is primarily determined by the variance of its inputs. Standardizing these inputs helps maintain a consistent variance for the z values in that layer.
- By adopting this weight sampling approach across all layers of the network, it ensures that the variance of the z values remains stable throughout.
- Different weight initialization strategies takes into account both n_{in} and n_{out}, and their compatibility with various activation functions may vary.

Other Weight Initialization Schemes

- One of the most recognized weight initialization methods is **Xavier initialization:**

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

  - A straightforward variant of Xavier initialization is :

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

- An modified version of Xavier initialization, known as **He initialization** (also termed **Kaiming initialization**) is:

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

- Combining both **Xavier** and **He** methods

Example

- Consider a fully connected 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 is initialized with $\mathbf{W}^{(k)} \sim N(\mu, \sigma)$ with different initialization schemes.
  - From $\mathbf{W}^{(1)} \sim N\left(0, \sqrt{\frac{1}{100}}\right)$, weights were sampled using Xavier initialization
  - From $\mathbf{W}^{(2)} \sim N(0, \sqrt{\frac{2}{80}})$, sampled using He initialization
  - Form $\mathbf{W}^{(3)} \sim N(0, \sqrt{\frac{2}{50}})$, sampled using He initialization

Internal Dynamics of the Network

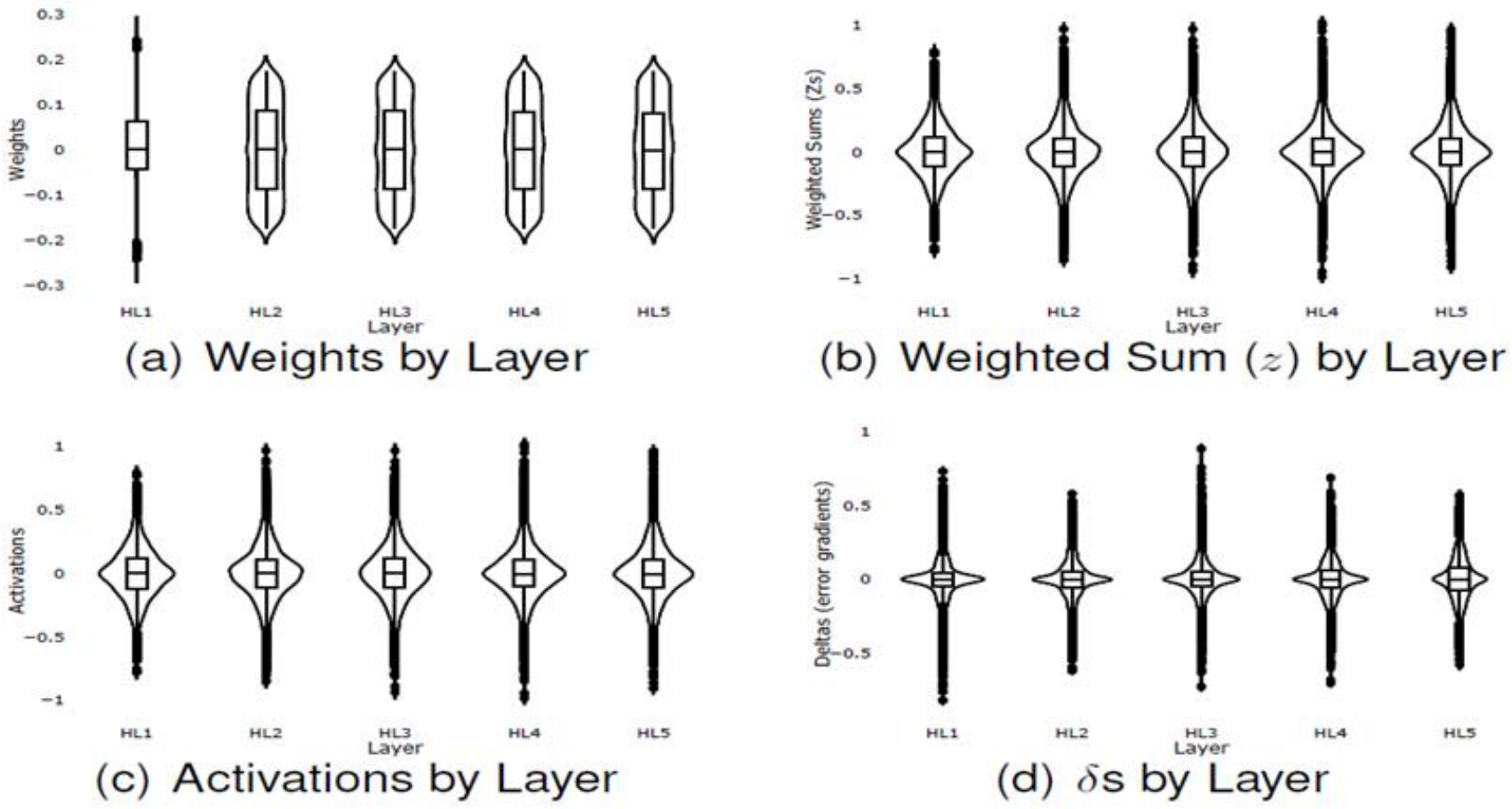


Figure: The internal dynamics of the network in the previous figure during the first training iteration when the weights were initialized using Xavier initialization.

Summary

- To ensure effective training of deep networks, it is essential to maintain consistent internal behavior across layers, which includes keeping the variance of z values, activations and δs stable.
- This consistency facilitates the addition of more layers without encountering issues like saturated units, extreme z values or problems with exploding or vanishing δs.

Outline

- Vanishing Gradients and ReLUs
- Dying ReLUs and ReLU's variants
- Weight Initialization and Unstable Gradients
- ☞ **Handling Categorical Target Feature**
- Early Stopping and Dropout: Preventing Overfitting

Neural Network for Classification Problem

- So far, all the cases we've discussed have been centered around regression problems.
- When designing a neural network for predicting multi-class categorical variables, three key modifications are required.
  1. Employee **one-hot encoding** for the target variable.
  2. Adapt the network's output layer to include a **softmax** layer
  3. Change the training loss function to the **cross-entropy** function.

One-hot Encoding

- **One-hot encoding** is a method used to represent categorical feature values in a vector format.
- Example

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

Table: 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.**

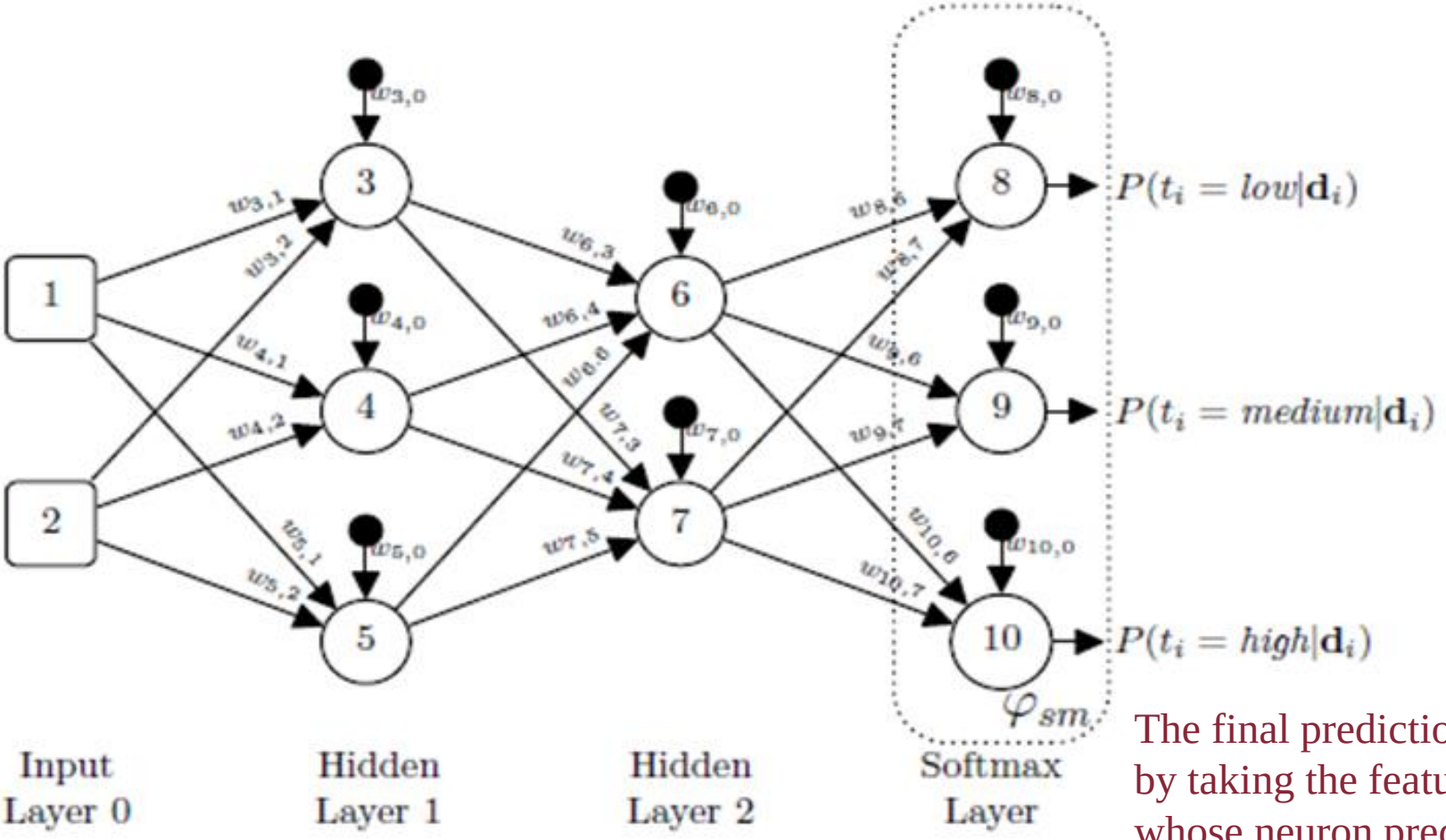
Softmax Output Layer

- The network's output layer is adjusted to function as a **softmax layer**. In such as layer, each target feature is represented by a dedicated neuron.
- The activation function utilized by the neuron in a softmax layer is the **softmax function.** For an output layer containing *m* neurons, the formula of activation is : $\varphi_{sm}(z_i) = \frac{e^{z_i}}{\sum_j^m e^{z_j}}$
- In the context of the softmax layer, the non-normalized z values are typically referred to as **logits**
  - E.g., When calculating the softmax activation function φ_{sm} over a vector, the three values are often termed logits.

	l_0	l_1	l_2
l	1.5	-0.9	0.6
e^{l_i}	4.48168907	0.40656966	1.8221188
$\sum_i e^{l_i}$			6.71037753
$\varphi_{sm}(l_i)$	0.667874356	0.060588195	0.27153745

$$\varphi_{sm}(l_1) = \frac{4.48168907}{7.71037753} = 0.667874356$$

An ANN with a Three-Neuron Softmax Output Layer



The final prediction is made by taking the feature level whose neuron predicts the highest probability

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

Cross-Entropy Error Function

- The **cross-entropy error** (or **loss**) **function**

$$L_{CE}(t, \hat{P}) = -\sum_j t_j \ln(\hat{P}_j)$$

$$L_{CE}(t, \hat{P}) = -\ln(\hat{P}_*)$$

  - 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. $\hat{\mathbf{P}}_*$ is the predicted probability for the true category (i.e., the category encoded as a 1 in the one-hot encoded vector **t**)
  - $$\begin{aligned} L_{CE}\left(\mathbf{t}, \hat{\mathbf{P}}\right) &= -\sum_j \mathbf{t}_j \ln\left(\hat{\mathbf{P}}_j\right) \\ &= -\left(\left(\mathbf{t}_0 \ln\left(\hat{\mathbf{P}}_0\right)\right) + \left(\mathbf{t}_1 \ln\left(\hat{\mathbf{P}}_1\right)\right) + \left(\mathbf{t}_2 \ln\left(\hat{\mathbf{P}}_2\right)\right)\right) \\ &= -\left(\left(0 \ln\left(\hat{\mathbf{P}}_0\right)\right) + \left(1 \ln\left(\hat{\mathbf{P}}_1\right)\right) + \left(0 \ln\left(\hat{\mathbf{P}}_2\right)\right)\right) \\ &= -1 \ln\left(\hat{\mathbf{P}}_1\right) \end{aligned}$$

Cross-Entropy Error Function (Cont.)

- The value of $\boldsymbol{\delta_k}$ for a neuron k in a softmax output layer, when using a cross-entropy loss function, depends on the output of neuron k.

$$\begin{aligned}\delta_k &= \frac{\partial \mathcal{E}}{\partial z_k} \\ &= \frac{\partial L_{CE}\left(\mathbf{t}, \hat{\mathbf{P}}\right)}{\partial \mathrm{l}_k} \\ &= \frac{\partial - \ln\left(\hat{\mathbf{P}}_\star\right)}{\partial \mathrm{l}_k} \\ &= \frac{\partial - \ln\left(\hat{\mathbf{P}}_\star\right)}{\partial\left(\hat{\mathbf{P}}_\star\right)} \times \frac{\partial\left(\hat{\mathbf{P}}_\star\right)}{\partial \mathrm{l}_k}\end{aligned}$$

where $\dfrac{\partial - \ln\left(\hat{\mathbf{P}}_\star\right)}{\partial\left(\hat{\mathbf{P}}_\star\right)} = -\dfrac{1}{\hat{\mathbf{P}}_\star}$

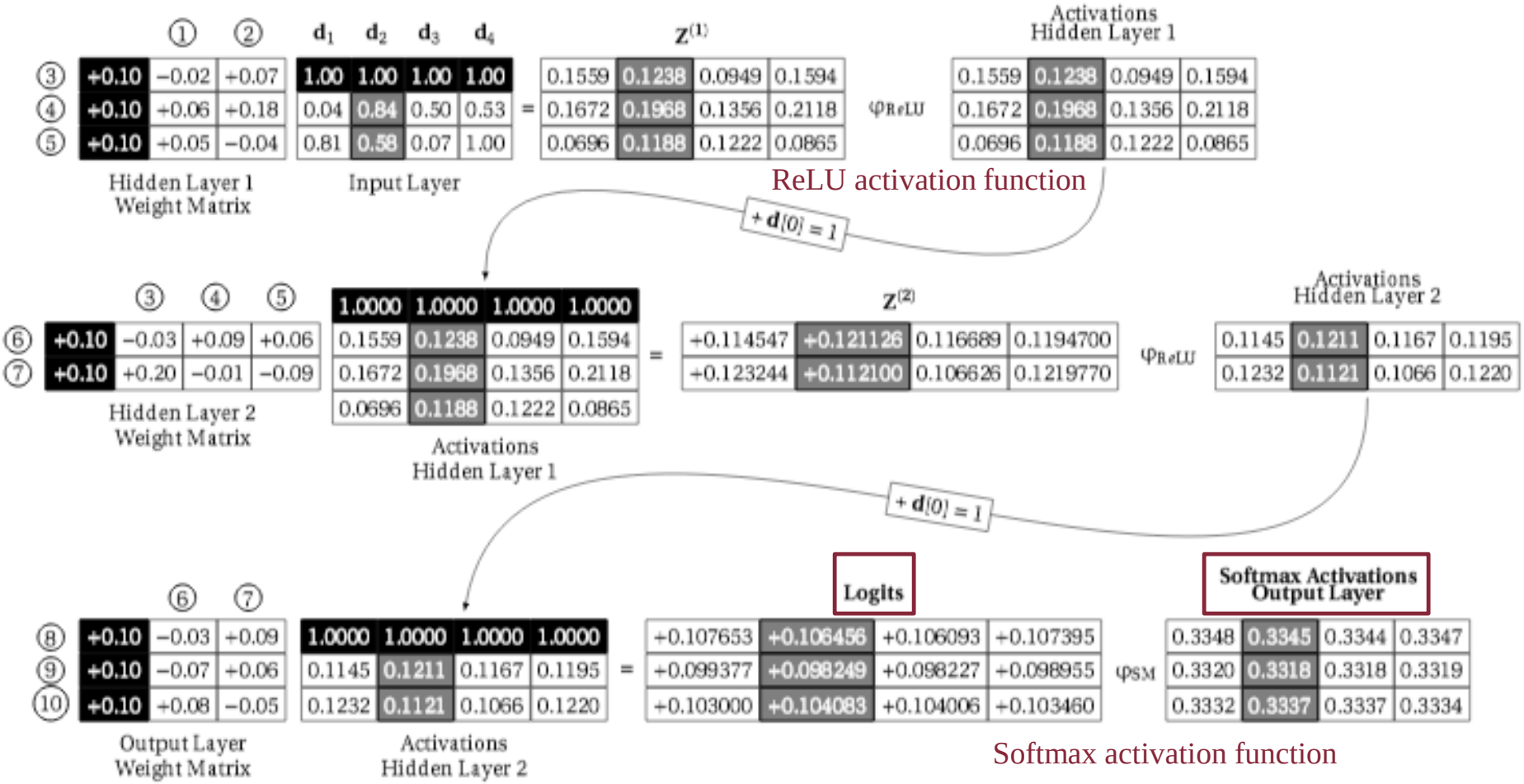
$$\frac{\partial\left(\hat{\mathbf{P}}_\star\right)}{\partial \mathrm{l}_k} = \begin{cases} \hat{\mathbf{P}}_\star\left(1 - \hat{\mathbf{P}}_k\right) & \text{if } k = \star \\ -\hat{\mathbf{P}}_\star \hat{\mathbf{P}}_k & otherwise \end{cases}$$

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

$\delta_{k=\star} = -\left(1 - \hat{\mathbf{P}}_k\right)$, for the category specified by a 1 in the one-hot encoded target vector

$\delta_{k\neq\star} = \hat{\mathbf{P}}_k$, for each of the other neurons in the softmax output layer

Forward Pass of Networks with Softmax Output Layer



Example: Calculation of the Softmax Activations

	$\mathbf{d}_1$	$\mathbf{d}_2$	$\mathbf{d}_3$	$\mathbf{d}_4$
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^{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
$\sum_i e^{l_i}$	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

$$\varphi_{sm}(l_i) = \frac{e^{l_i}}{\sum_j^m e^{l_j}}$$

$$\delta_{k=\star} = -\left(1 - \hat{\mathbf{P}}_k\right)$$

$$\delta_{k\neq\star} = \hat{\mathbf{P}}_k$$

Table: 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.

Network Weight Update

- After calculating the δs for the output neurons, the backpropagation of the δs and the weight updates proceed as shown in the previous case example.
- Example: Updating the weight $w_{9,6}$

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

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

Outline

- Vanishing Gradients and ReLUs
- Dying ReLUs and ReLU's variants
- Weight Initialization and Unstable Gradients
- Handling Categorical Target Feature
- ☞ **Early Stopping and Dropout: Preventing Overfitting**

Prevent Overfitting

- Deep learning models often contains millions of parameters, which can lead to high complexity and overfitting**.**
- **To mitigate overfitting**, two widely used techniques in neural networks are:
  - **Early stopping**
  - **Dropout**

Early Stopping

- The core concept behind **early stopping** is to identify the point during an iterative training process, such as backpropagation, when the model begins to overfit.
- This usually happens when the model's error on a validation dataset starts increasing.
- To implement early stopping, a common approach involves a **'patience' parameter,** which sets a limit on how many consecutive times the validation error can exceed the previously recorded lowest value before terminating the training process.

Dropout

- **Dropout** is a simple yet remarkably effective strategy to combat overfitting.
- During each training instance, a random subset of neurons from the input and hidden layers is chosen and temporarily excluded from the network.
- As a result, the backpropagation is performed on this reduced network without the randomly omitted neurons.

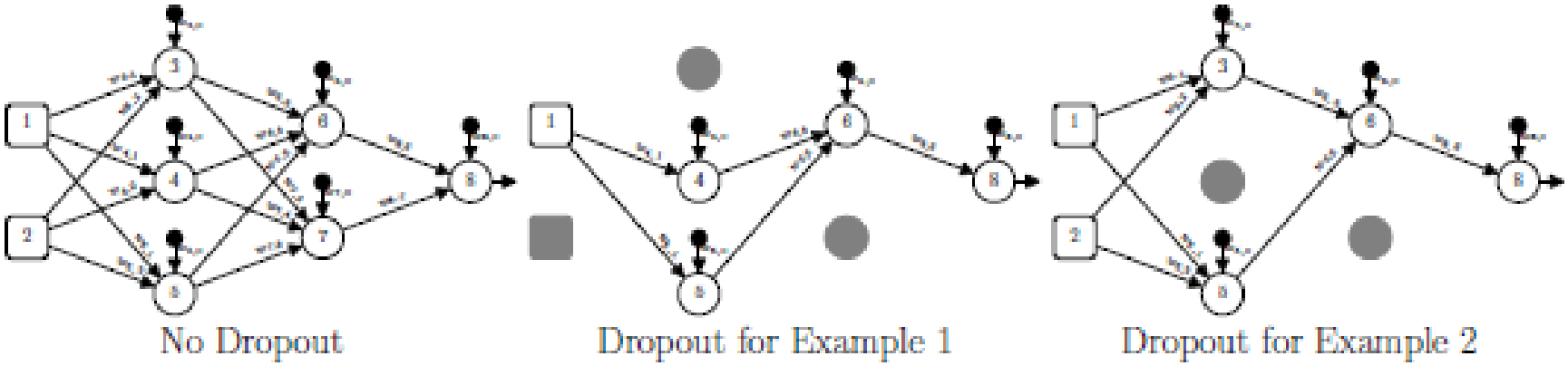


Figure: we randomly choose a new set of notes to drop and then do backpropagation as usual. The gray nodes mark the neurons that have been dropped from the network for the training example

Implementation of Dropout

- **Inverted dropout** is a commonly employed method for implementing dropout.
- This approach deactivates a neuron by zeroing its activation value during the forward pass.
- Consequently, the neuron's activation doesn't contribute to the subsequent layers, rendering it inactive and uninvolved in influencing the model's output.