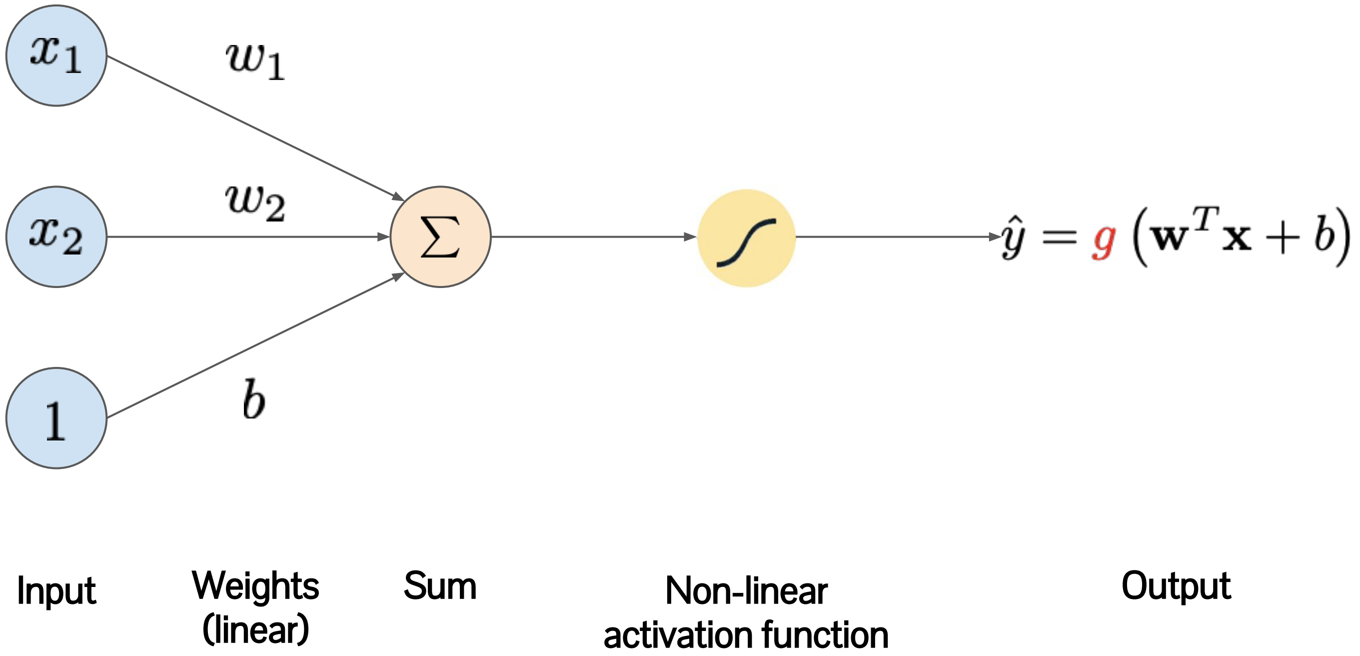
Neural Network Optimization

Instructor: Seunghoon Hong

Announcement

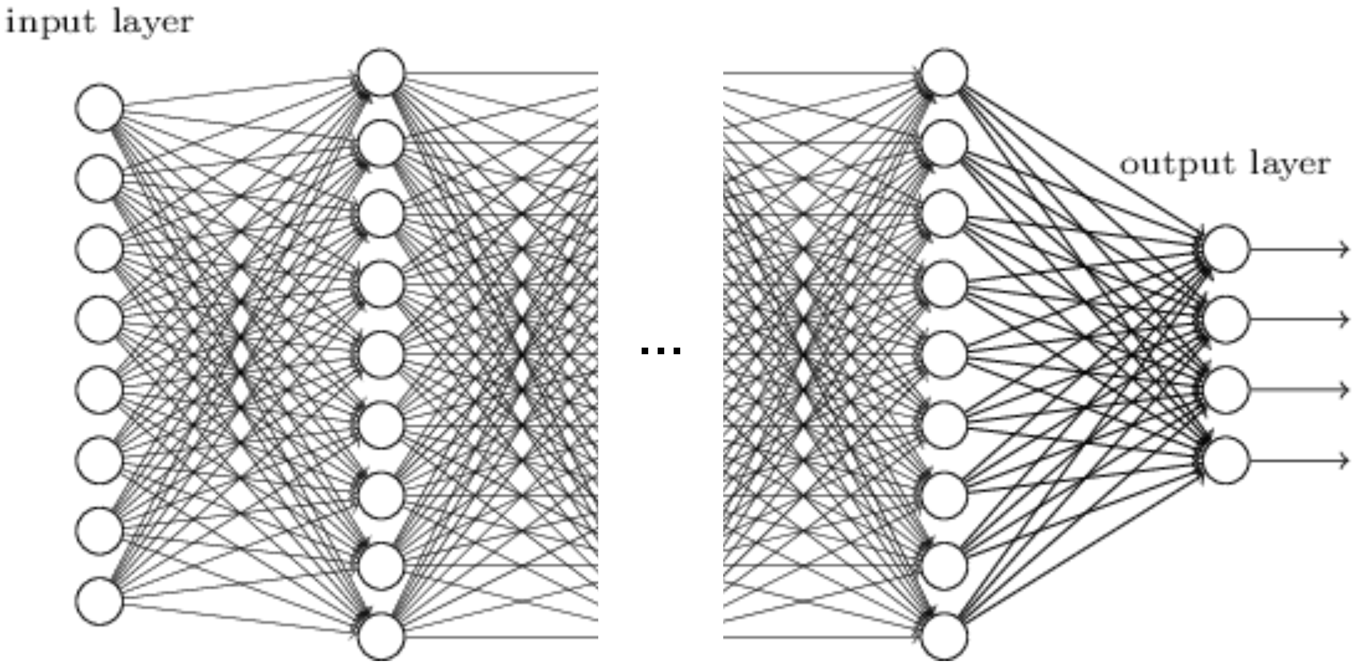
- Team formation deadline is **9/11**!
- There is a pre-class materials before the next class.
    - Colab notebook: Pytorch Tutorial **1-4**
    - Additional material: 60 minutes blitz of PyTorch

Recap: perceptron



Recap: (deep) neural network

- A stack of perceptrons (weights, nonlinear activation)



Recap: training neural network

- Objective: find a set of parameters that minimize the loss on the dataset
- Notations
  - Datasets: $\left\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \ldots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\right\} \rightarrow$ ***N*** number of training data
  - Parameters: $\left\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(L)}\right\} \rightarrow$ ***L*** number of layers

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \frac{1}{N} \sum_{I=1}^{N} \mathcal{L}(f(\mathbf{x}^{(i)}; \mathbf{W}), \mathbf{y}^{(i)})$$

Loss function

- Measurement on the mismatch between the model prediction and the true label
- There are many ways to define the degree of mismatch (i.e. misprediction, error)

Recap: optimization via gradient descent

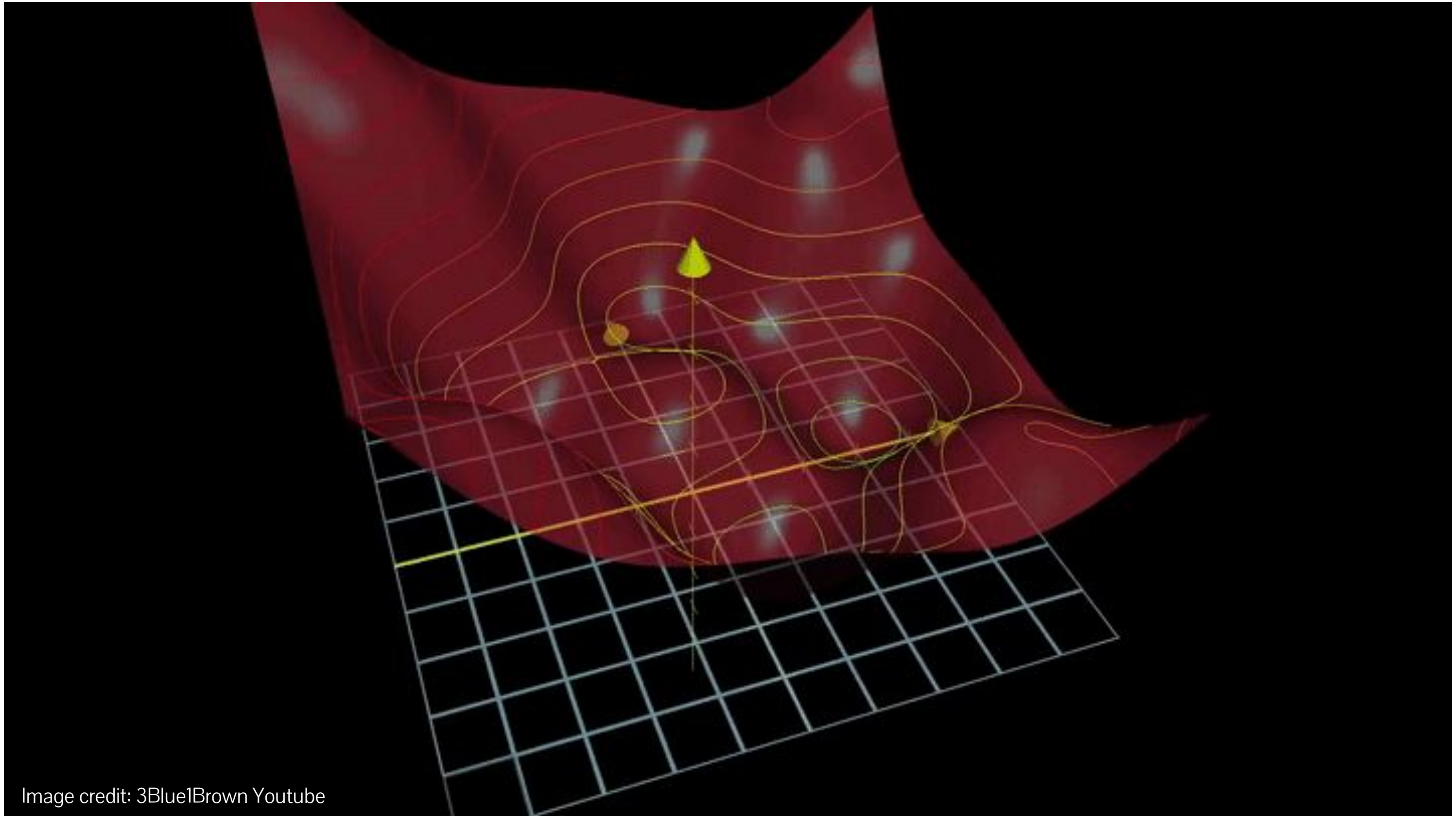
- 

Image credit: 3Blue1Brown Youtube

Recap: optimization via gradient descent

Algorithm (gradient descent)

1. Randomly initialize the parameters $\mathbf{W} \leftarrow \mathbf{W}_0$
2. Repeat until convergence:
3. Compute gradient $\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}}$
4. Update the parameters by $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}}$
5. $\mathbf{W}^* \leftarrow \mathbf{W}$

$$\mathcal{J}(\mathbf{W}) = \frac{1}{N} \sum_{I=1}^{N} \mathcal{L}(f(\mathbf{x}^{(i)}; \mathbf{W}), \mathbf{y}^{(i)})$$

Recap: optimization via stochastic gradient descent

Algorithm (stochastic gradient descent)

1. Randomly initialize the parameters $\mathbf{W} \leftarrow \mathbf{W}_0$
2. Repeat until convergence:
3. Sample a data $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$
4. Compute gradient $\frac{\partial \mathcal{J}^{(i)}(\mathbf{W})}{\partial \mathbf{W}}$
5. Update the parameters by $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial \mathcal{J}^{(i)}(\mathbf{W})}{\partial \mathbf{W}}$
6. $\mathbf{W}^* \leftarrow \mathbf{W}$

$$\mathcal{J}^{(i)}(\mathbf{W}) = \mathcal{L}(f(\mathbf{x}^{(i)}; \mathbf{W}), \mathbf{y}^{(i)})$$

Point-wise loss & gradient estimation
Very efficient to compute but **very noise**

Recap: optimization via stochastic gradient descent

Algorithm (minibatch stochastic gradient descent)

1. Randomly initialize the parameters $\mathbf{W} \leftarrow \mathbf{W}_0$
2. Repeat until convergence:
3. Sample a **batch** of data $\mathcal{B} = \{(\mathbf{x}^{(k)}, \mathbf{y}^{(k)})\}_{k=1}^{B}$
4. Compute gradient $\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial \mathcal{J}^{(k)}(\mathbf{W})}{\partial \mathbf{W}}$
5. Update the parameters by $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}}$
6. $\mathbf{W}^* \leftarrow \mathbf{W}$

How can we compute the gradient by the way?

Algorithm (minibatch stochastic gradient descent)

1. Randomly initialize the parameters $\mathbf{W} \leftarrow \mathbf{W}_0$
2. Repeat until convergence:
3. Sample a **batch** of data $\mathcal{B} = \{(\mathbf{x}^{(k)}, \mathbf{y}^{(k)})\}_{k=1}^{B}$
4. Compute gradient $\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial \mathcal{J}^{(k)}(\mathbf{W})}{\partial \mathbf{W}}$
5. Update the parameters by $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}}$
6. $\mathbf{W}^* \leftarrow \mathbf{W}$

Gradient of the loss for all parameters!

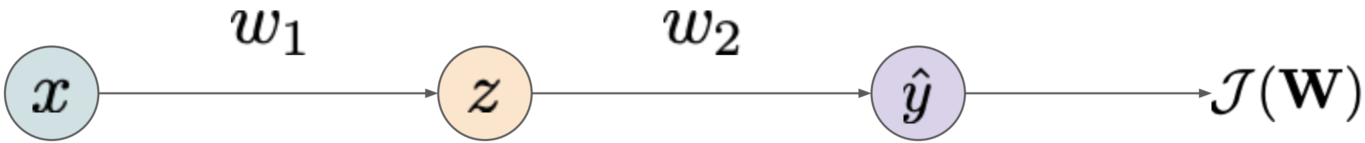
Today’s agenda

- Optimization of Neural Network
  - Backpropagation
- Improving neural network training
  - Normalization, initialization, regularization
- Practical tips for neural network training
  - Learning rate scheduling, hyper-parameter tuning

Computing gradients of weights in neural network

- Simplest example: two-layer neural network with one hidden node

$$\hat{y} = f(x; \underbrace{\mathbf{W}}_{\boxed{\{w_1, w_2\}}})$$



Computing gradients of weights in neural network

- Simplest example: two-layer neural network with one hidden node

$$\hat{y} = f(x; \mathbf{W})$$

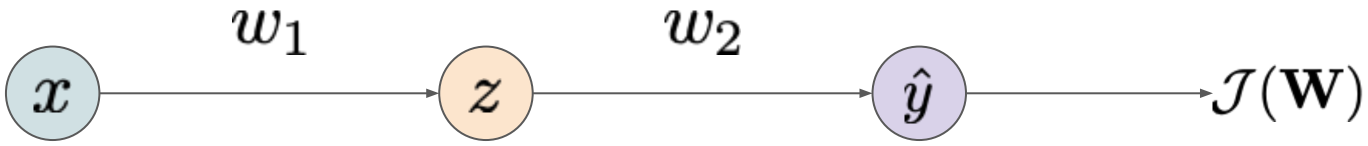
$$x \xrightarrow{w_1} z \xrightarrow{w_2} \hat{y} \longrightarrow \mathcal{J}(\mathbf{W})$$

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial w_1} = ?$$

Computing gradients of weights in neural network

- Simplest example: two-layer neural network with one hidden node

$$\hat{y} = f(x; \mathbf{W})$$

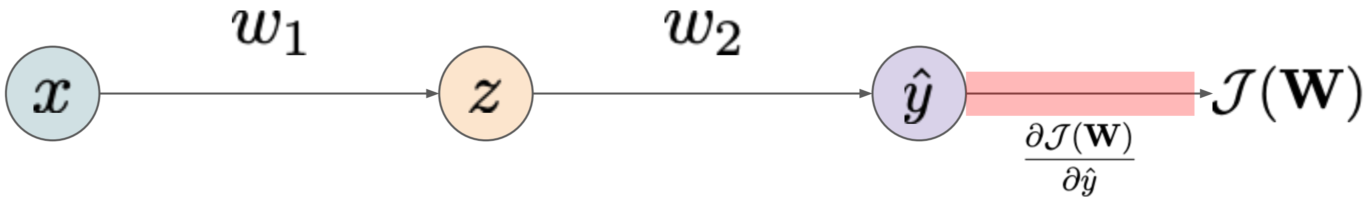


Chain rule: propagating the gradient across the layers $\dfrac{\partial \mathcal{J}(\mathbf{W})}{\partial w_1} = \dfrac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{y}} \cdot \dfrac{\partial \hat{y}}{\partial z} \cdot \dfrac{\partial z}{\partial w_1}$

Computing gradients of weights in neural network

- Simplest example: two-layer neural network with one hidden node

$$\hat{y} = f(x; \mathbf{W})$$



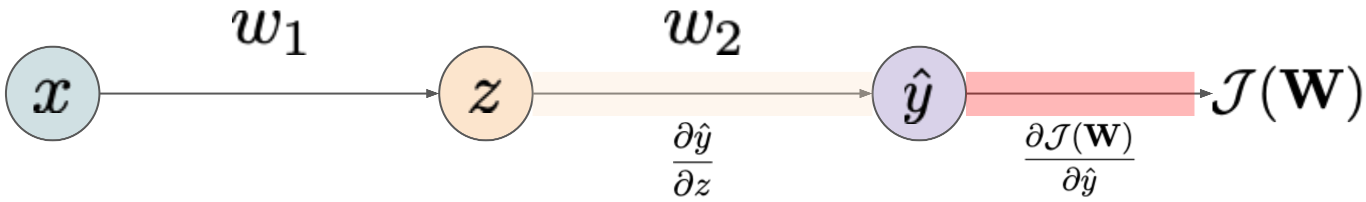
Chain rule: propagating the gradient across the layers

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial w_1} = \boxed{\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{y}}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

Computing gradients of weights in neural network

- Simplest example: two-layer neural network with one hidden node

$$\hat{y} = f(x; \mathbf{W})$$



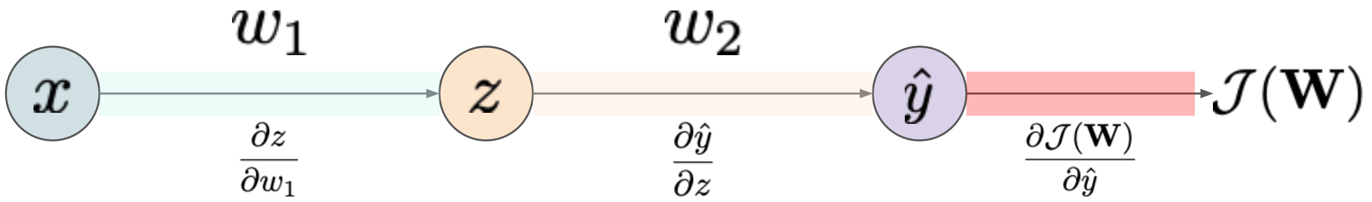
Chain rule: propagating the gradient across the layers

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial w_1} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

Computing gradients of weights in neural network

- Simplest example: two-layer neural network with one hidden node

$$\hat{y} = f(x; \mathbf{W})$$

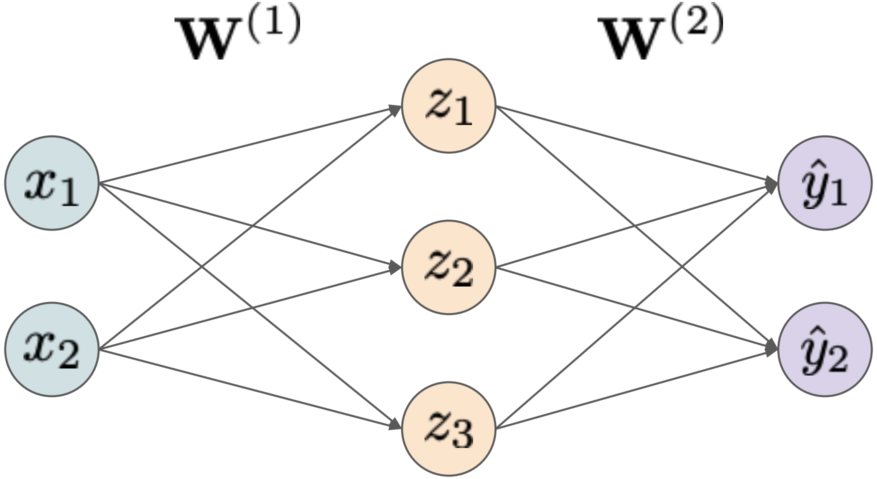


Chain rule: propagating the gradient across the layers

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial w_1} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

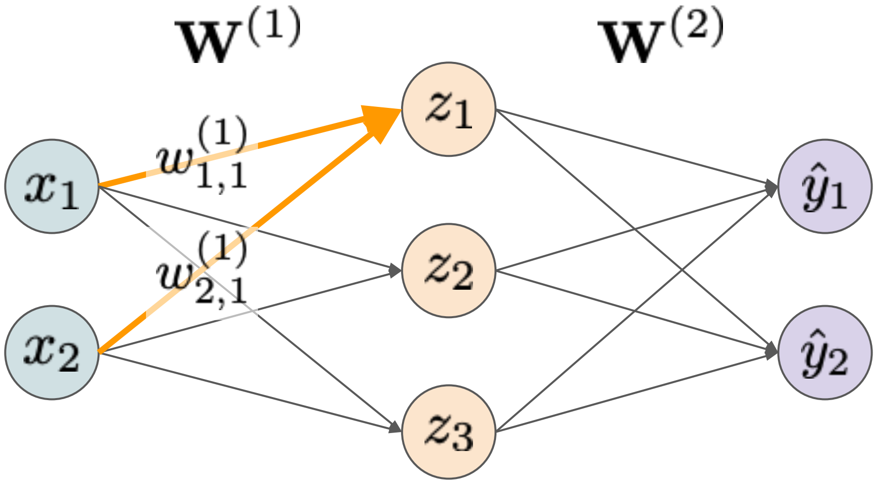
Computing gradients of weights in neural network

- Fully-connected network



Computing gradients of weights in neural network

- Fully-connected network



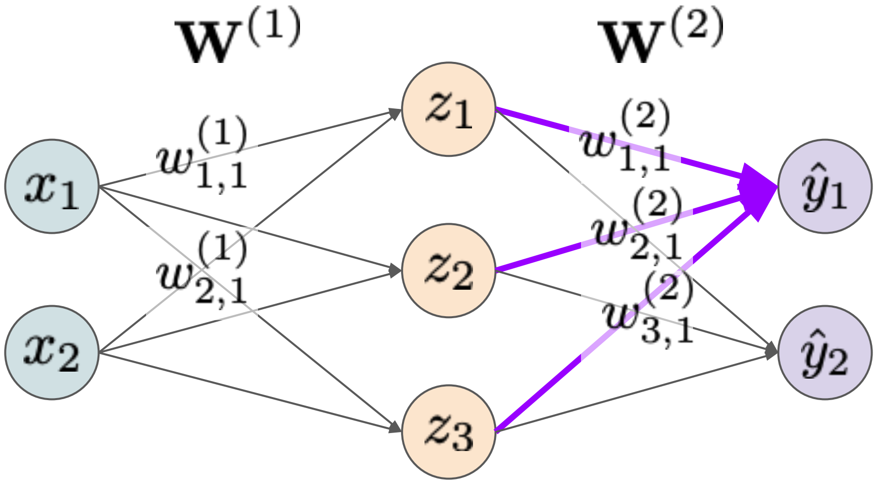
$$z_j = \sigma\left(\sum_{I=1}^{2} w_{i,j}^{(1)} x_i + b_j\right)$$

Example:

$$z_1 = \sigma(w_{1,1}x_1 + w_{2,1}x_2 + b_1)$$

Computing gradients of weights in neural network

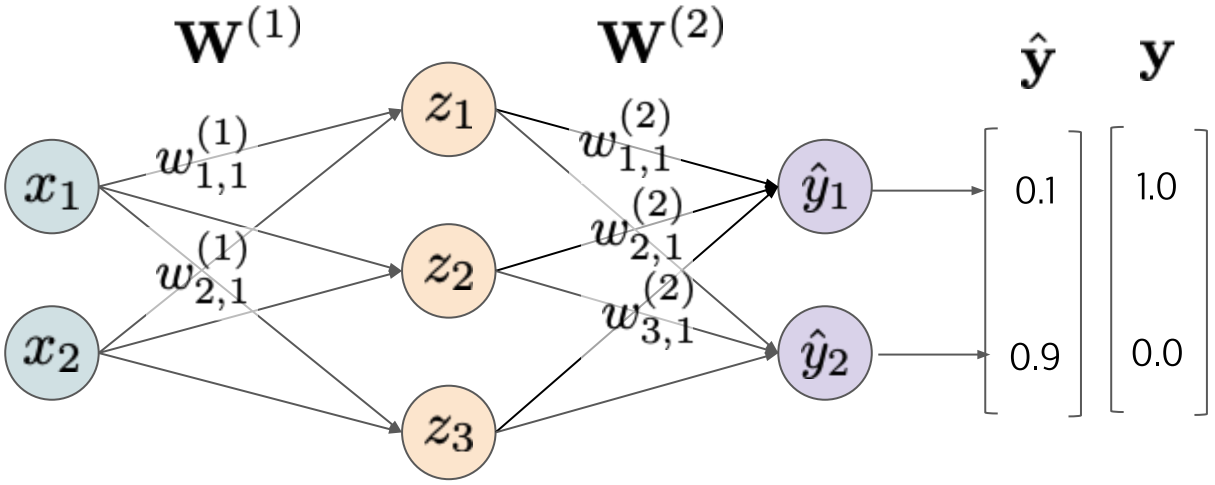
- Fully-connected network



$$z_j = \sigma\left(\sum_{I=1}^{2} w_{i,j}^{(1)} x_i + b_j\right) \qquad \hat{y}_k = \sum_{I=1}^{2} w_{i,k}^{(2)} z_i + b_k$$

Computing gradients of weights in neural network

- Fully-connected network



$$z_j = \sigma\left(\sum_{I=1}^{2} w^{(1)}_{i,j} x_i + b_j\right) \quad \hat{y}_k = \sum_{I=1}^{2} w^{(2)}_{i,k} z_i + b_k \quad \mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

Computing gradients of weights in neural network

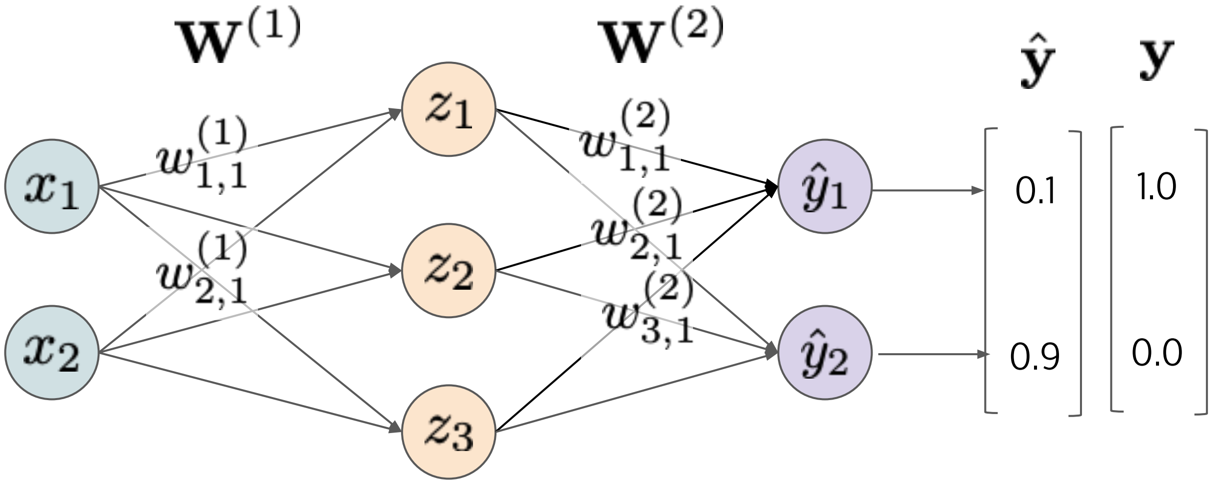
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



Computing gradients of weights in neural network

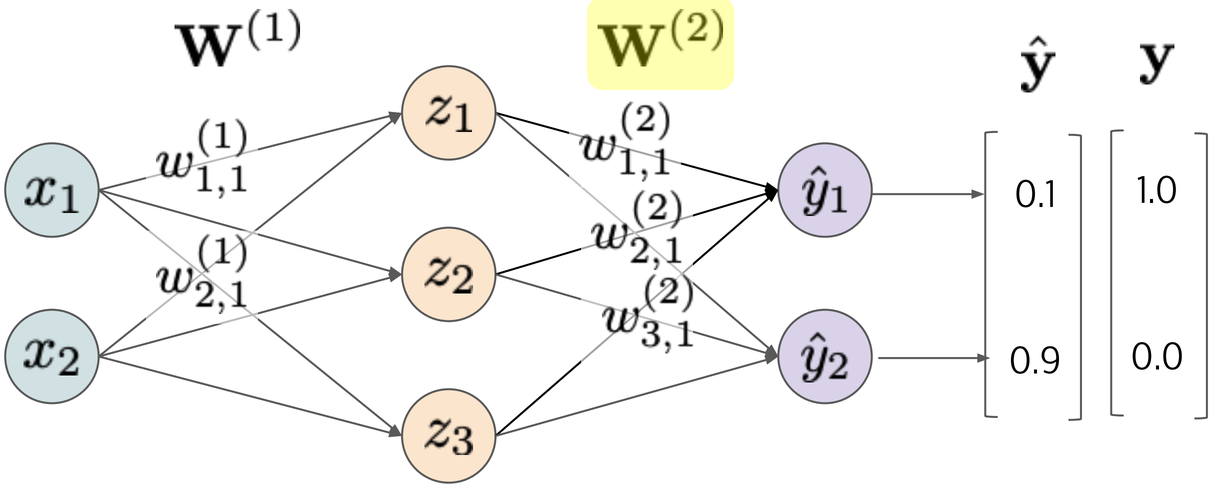
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



Backwards (gradients)

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(2)}} =$$

Computing gradients of weights in neural network

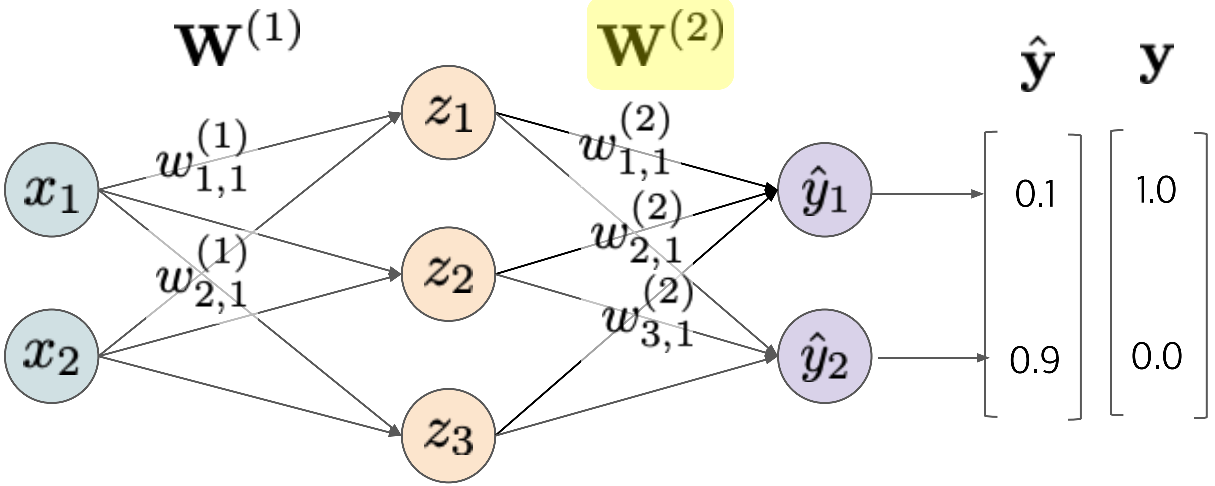
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



Backwards (gradients)

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^{(2)}}$$

Computing gradients of weights in neural network

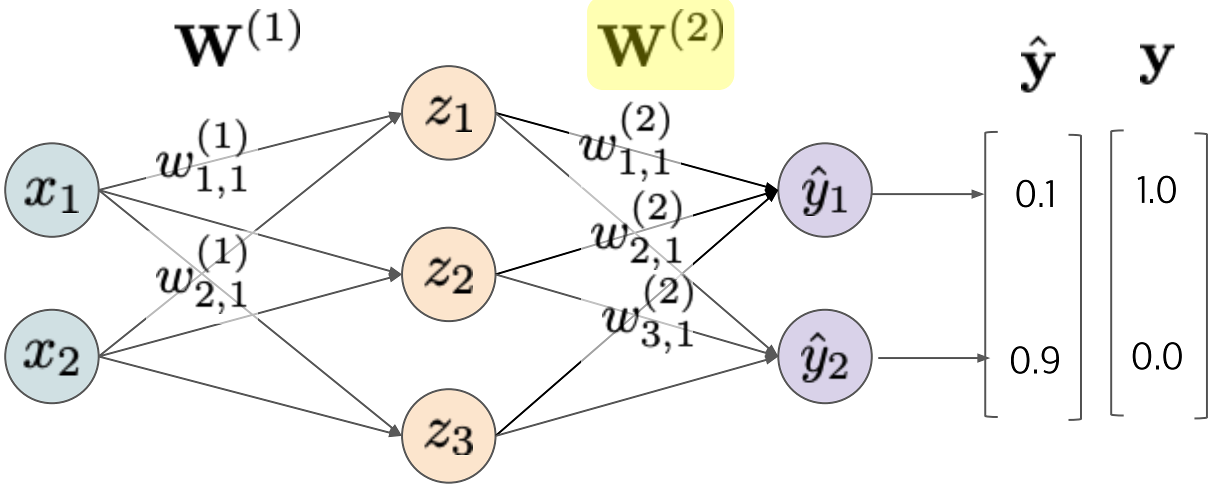
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



Backwards (gradients)

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^{(2)}}$$

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} = 2(\hat{\mathbf{y}} - \mathbf{y})$$

Computing gradients of weights in neural network

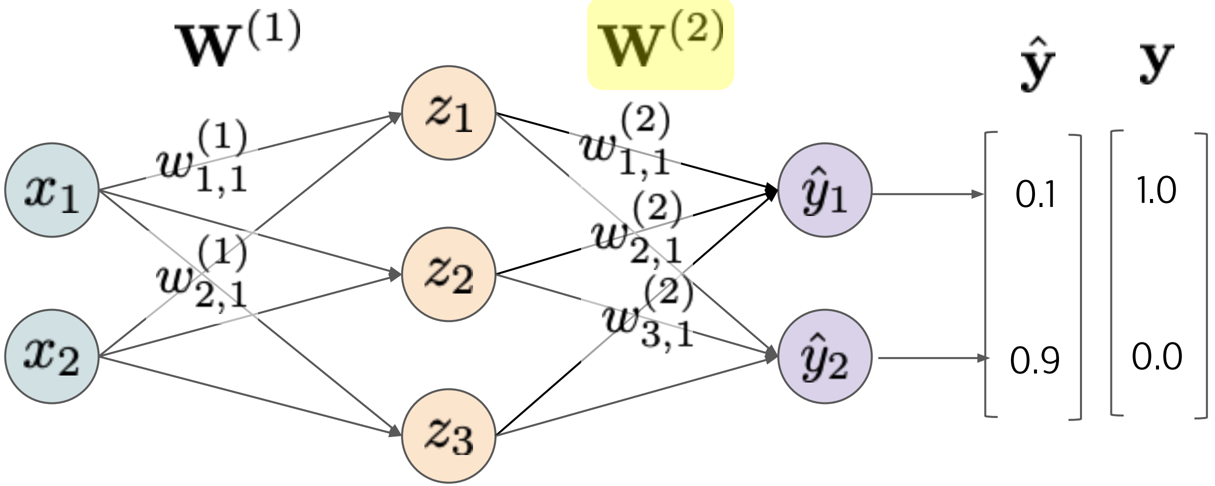
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



Backwards (gradients)

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^{(2)}}$$

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} = 2(\hat{\mathbf{y}} - \mathbf{y}) \qquad \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^{(2)}} = \mathbf{z}^{\mathrm{T}}$$

Computing gradients of weights in neural network

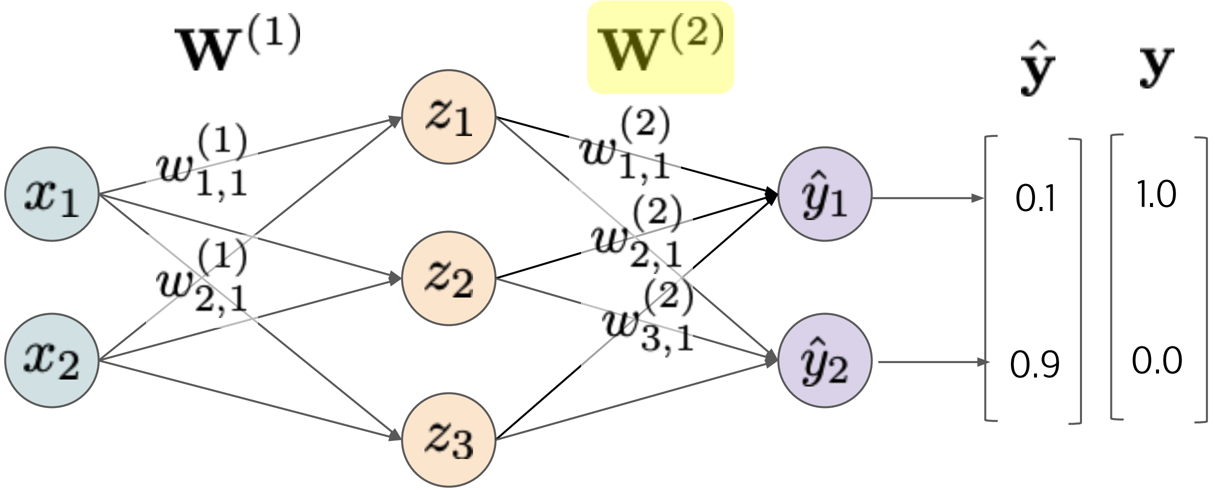
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



Backwards (gradients)

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^{(2)}} = 2(\hat{\mathbf{y}} - \mathbf{y})\mathbf{z}^{\mathrm{T}}$$

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} = 2(\hat{\mathbf{y}} - \mathbf{y}) \qquad \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^{(2)}} = \mathbf{z}^{\mathrm{T}}$$

Computing gradients of weights in neural network

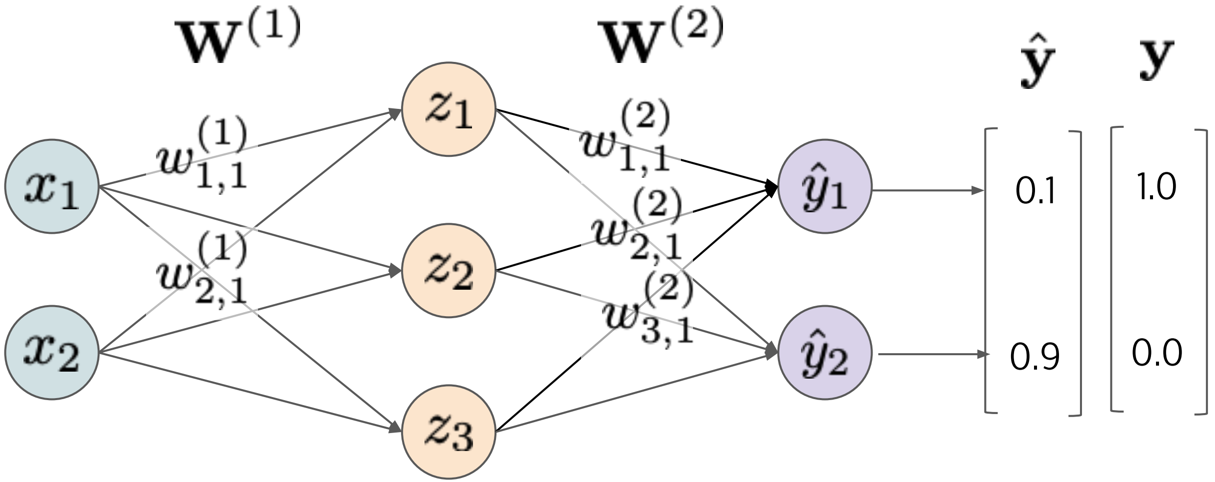
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



Backwards (gradients)

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{b}^{(2)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{b}^{(2)}} = 2(\hat{\mathbf{y}} - \mathbf{y})$$

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} = 2(\hat{\mathbf{y}} - \mathbf{y}) \qquad \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{b}^{(2)}} = 1$$

Computing gradients of weights in neural network

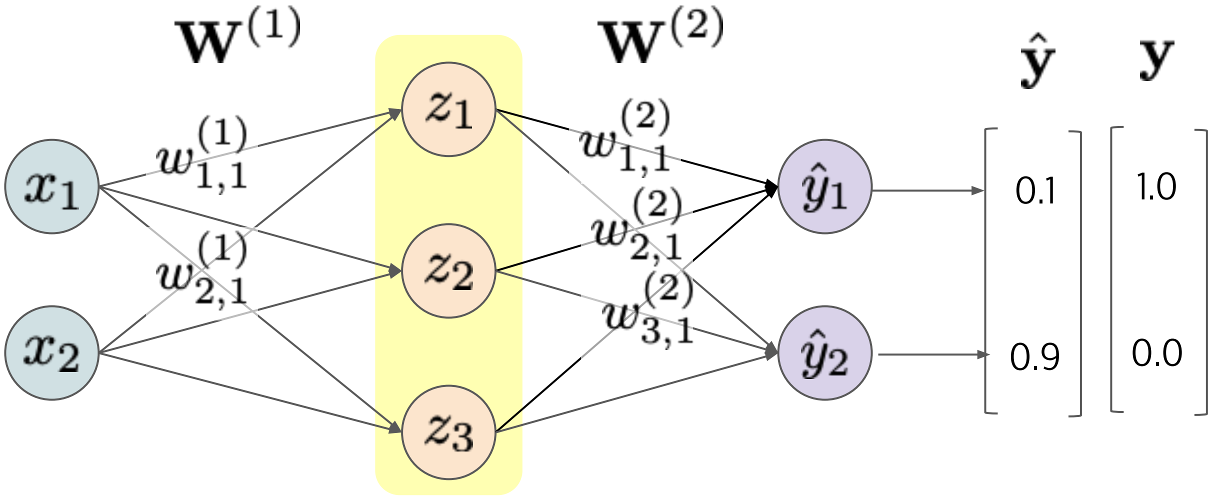
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



Backwards (gradients)

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{z}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} = 2\left(\mathbf{W}^{(2)}\right)^{\mathrm{T}}(\hat{\mathbf{y}} - \mathbf{y}) \qquad \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} = 2(\hat{\mathbf{y}} - \mathbf{y}) \quad \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} = \left(\mathbf{W}^{(2)}\right)^{\mathrm{T}}$$

Computing gradients of weights in neural network

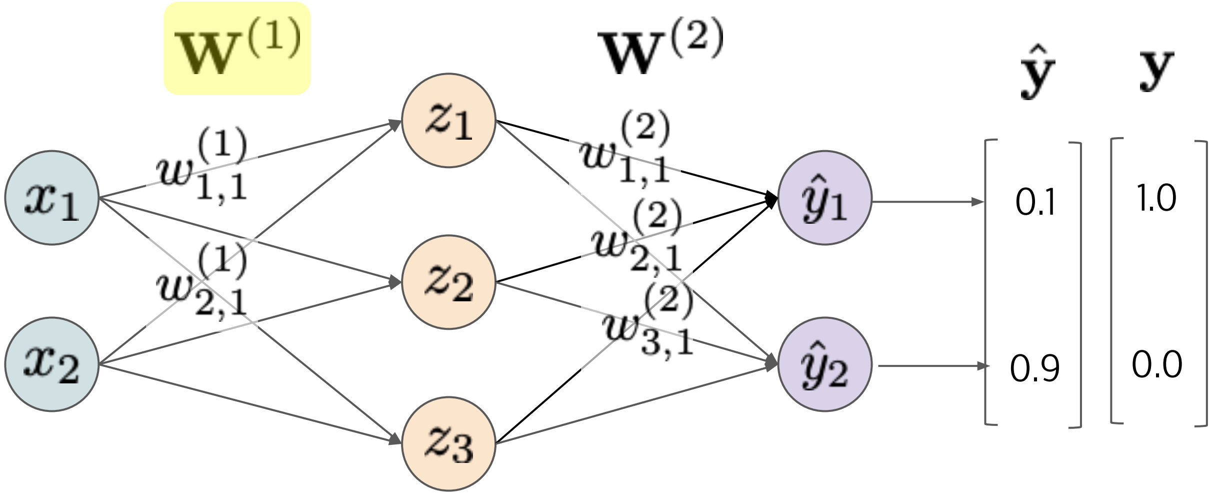
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



Backwards (gradients)

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}$$

Computing gradients of weights in neural network

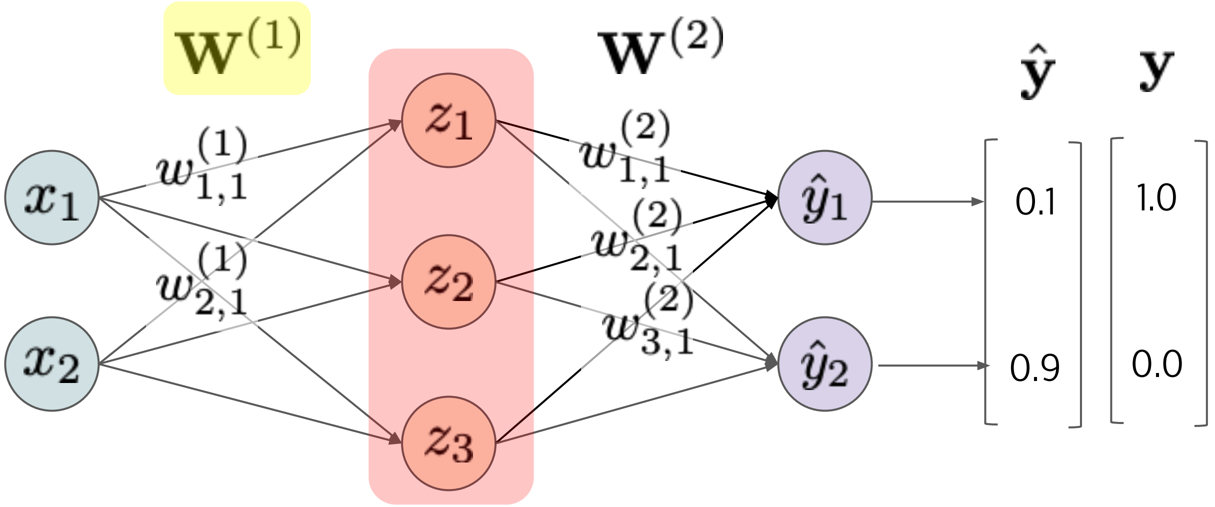
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



Backwards (gradients)

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}$$

Backpropagated gradient up to output z

Computing gradients of weights in neural network

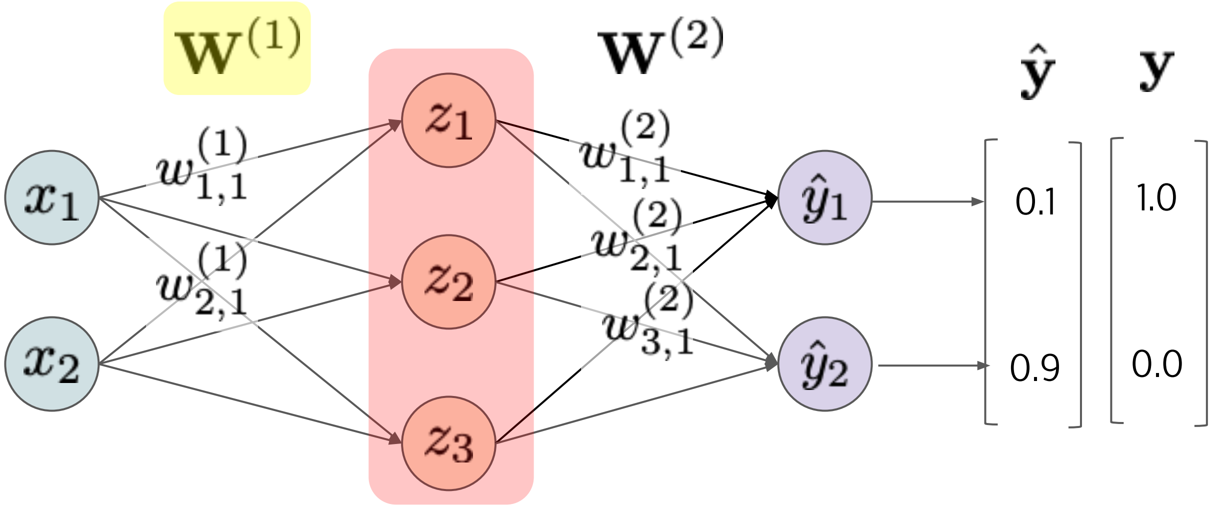
- Fully-connected network

Forwards

$$\mathcal{J}(\mathbf{W}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

$$\mathbf{z} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{z} + \mathbf{b}^{(2)}$$



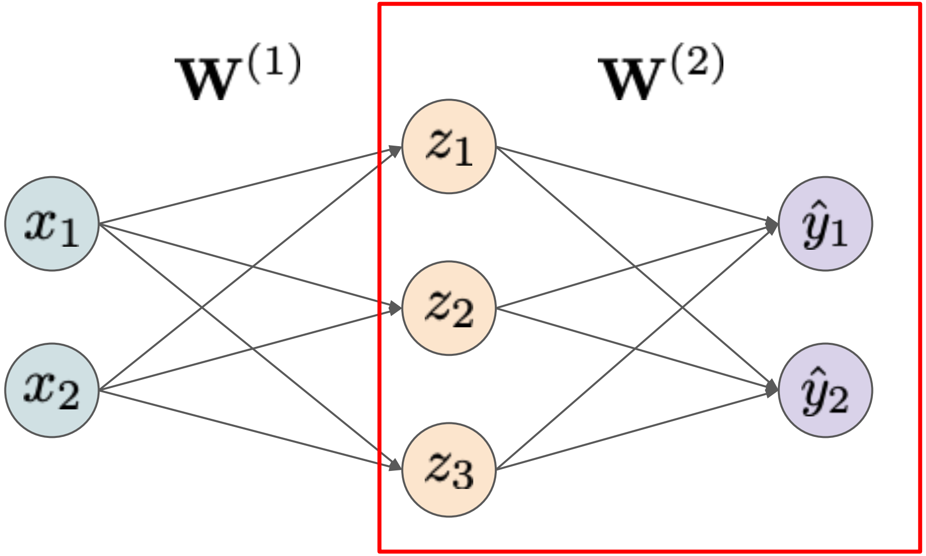
Backwards (gradients)

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{z}} \cdot \mathbf{x}^{\mathrm{T}}$$

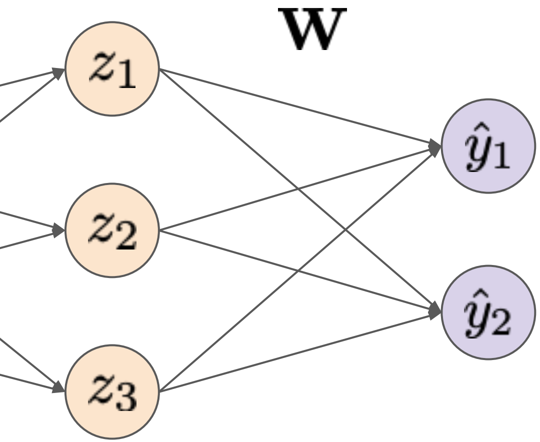
Take one step closer

- Fully-connected network

Let's see what happens at this layer more closely



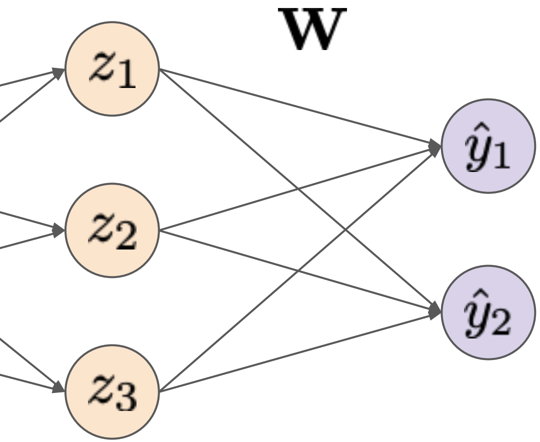
What is happening in backpropagation exactly?



Using matrix notation (without bias for simplicity)

$$\left.\begin{array}{l}\mathbf{z} \in \mathbb{R}^{m\times 1} \\ \hat{\mathbf{y}} \in \mathbb{R}^{n\times 1} \\ \mathbf{W} \in \mathbb{R}^{n\times m}\end{array}\right\} \quad \hat{\mathbf{y}} = \mathbf{W}\mathbf{z}$$

What is happening in backpropagation exactly?



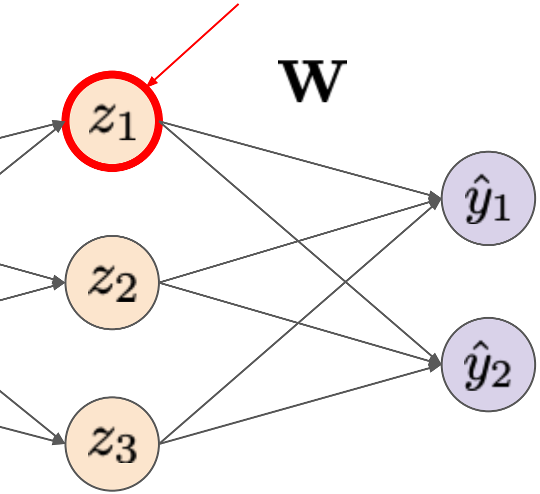
Using matrix notation (without bias for simplicity)

In this example, m=3, n=2

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{2,1} & w_{3,1} \\ w_{1,2} & w_{2,2} & w_{2,3} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

What is happening in backpropagation exactly?

Let's see what gradient this guy gets



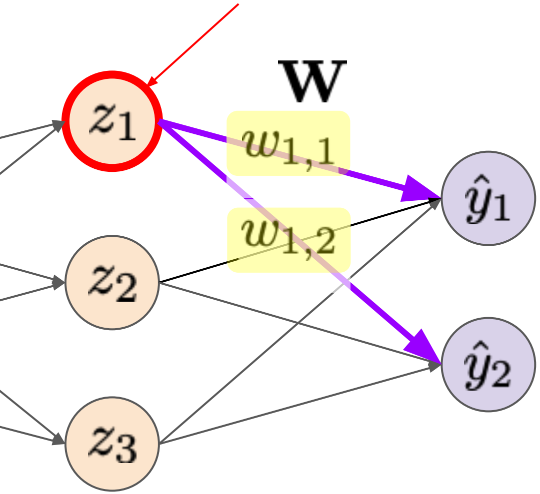
Using matrix notation (without bias for simplicity)

In this example, m=3, n=2

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{2,1} & w_{3,1} \\ w_{1,2} & w_{2,2} & w_{2,3} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

What is happening in backpropagation exactly?

Let's see what gradient this guy gets



Using matrix notation (without bias for simplicity)

In this example, m=3, n=2

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{2,1} & w_{3,1} \\ w_{1,2} & w_{2,2} & w_{2,3} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

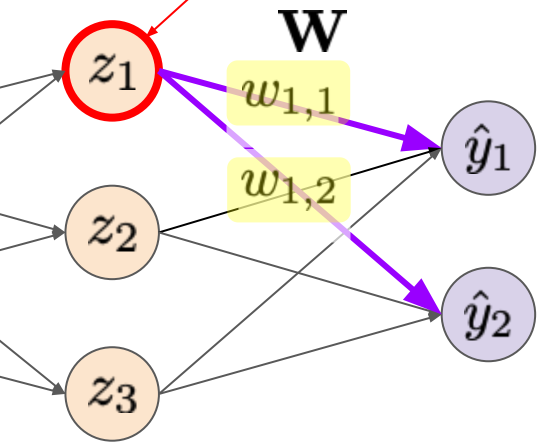
$$\hat{\mathbf{y}} = \boxed{\mathbf{w}_1} z_1 + \mathbf{w}_2 z_2 + \mathbf{w}_3 z_3$$

Aggregating gradients via outgoing edges from z1 !!

$$\frac{\partial \mathcal{J}}{\partial z_1} = \frac{\partial \mathcal{J}}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial z_1} = \mathbf{w}_1^{\mathrm{T}} \cdot \frac{\partial \mathcal{J}}{\partial \hat{\mathbf{y}}} = \begin{bmatrix} w_{1,1} & w_{1,2} \end{bmatrix} \begin{bmatrix} \frac{\partial \mathcal{J}}{\partial \hat{y}_1} \\ \frac{\partial \mathcal{J}}{\partial \hat{y}_2} \end{bmatrix} = w_{1,1} \frac{\partial \mathcal{J}}{\partial \hat{y}_1} + w_{1,2} \frac{\partial \mathcal{J}}{\partial \hat{y}_2}$$

What is happening in backpropagation exactly?

Let's see what gradient this guy gets



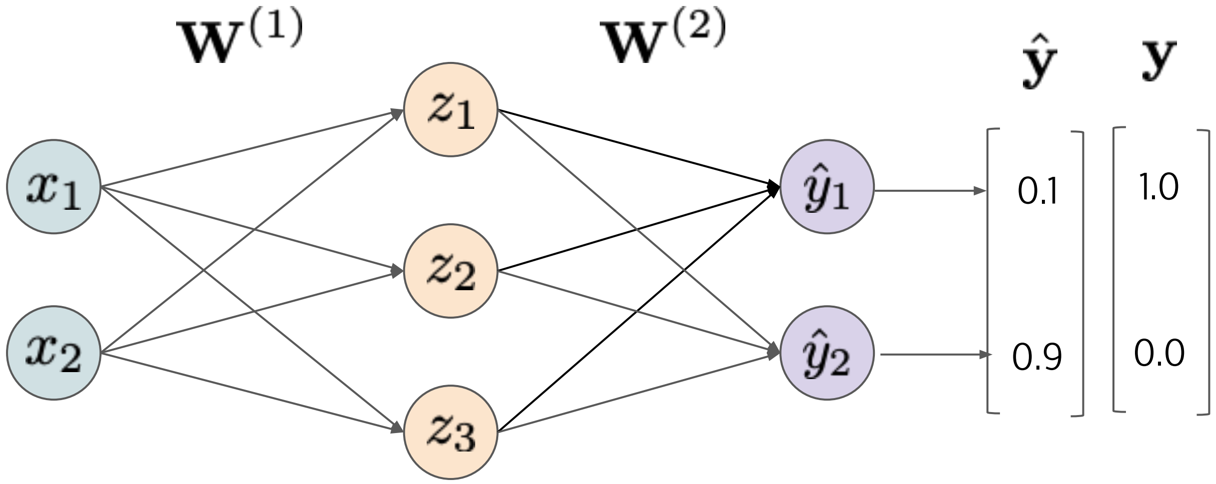
General rules for gradient computation:

- Gradient flows through only connected edges
- If there are multiple edges (gradients), gradient of the node is computed by aggregating them

Aggregating gradients via outgoing edges from z1 !!

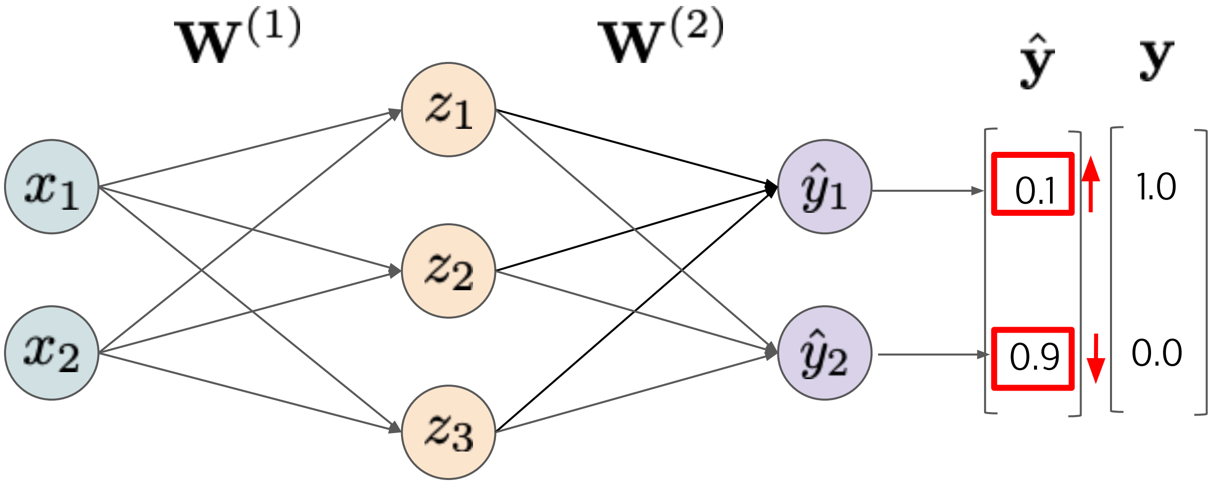
$$\frac{\partial \mathcal{J}}{\partial z_1} = \frac{\partial \mathcal{J}}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial z_1} = \mathbf{w}_1^{\mathrm{T}} \cdot \frac{\partial \mathcal{J}}{\partial \hat{\mathbf{y}}} = \begin{bmatrix} w_{1,1} & w_{1,2} \end{bmatrix} \begin{bmatrix} \frac{\partial \mathcal{J}}{\partial \hat{y}_1} \\ \frac{\partial \mathcal{J}}{\partial \hat{y}_2} \end{bmatrix} = w_{1,1} \frac{\partial \mathcal{J}}{\partial \hat{y}_1} + w_{1,2} \frac{\partial \mathcal{J}}{\partial \hat{y}_2}$$

What does it mean by updating parameters?



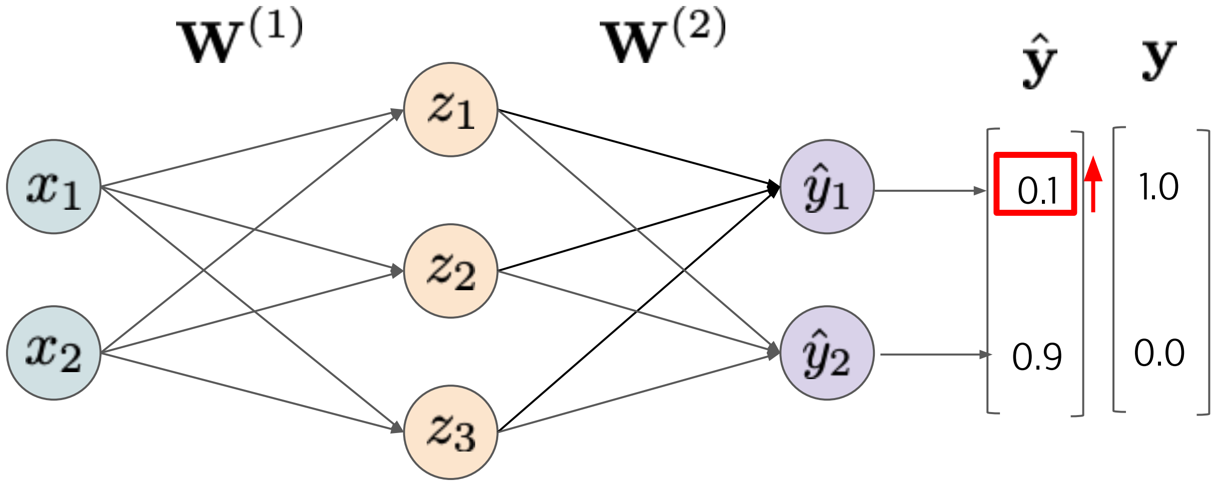
What does it mean by updating parameters?

we want to increase $\hat{y}_1$ and suppress $\hat{y}_2$ to match the prediction to the label



What does it mean by updating parameters?

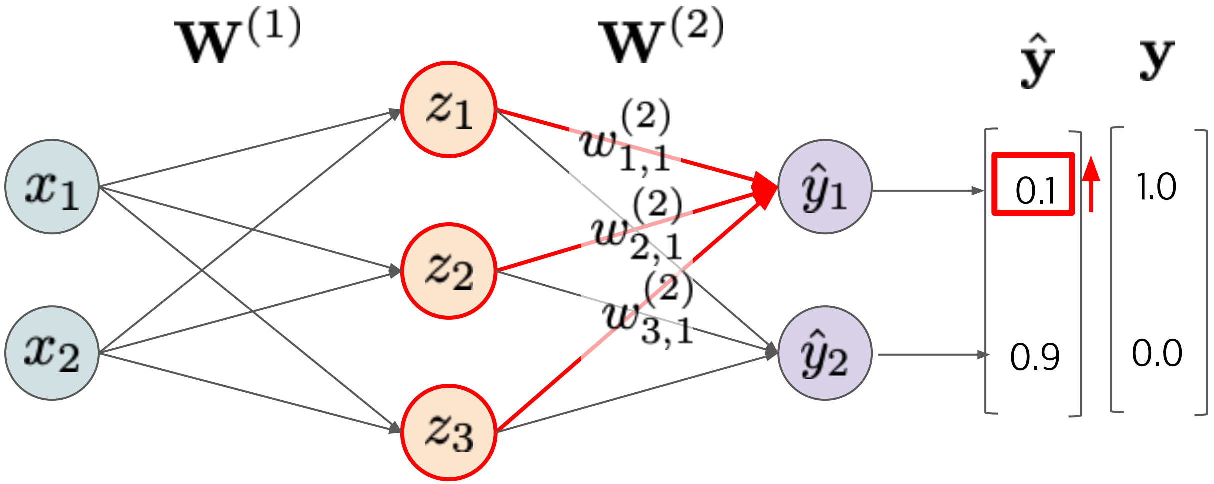
we want to increase $\hat{y}_1$ and suppress $\hat{y}_2$ to match the prediction to the label



Let's consider increasing y1 first

What does it mean by updating parameters?

The output is a function of the weights (+bias) and previous outputs



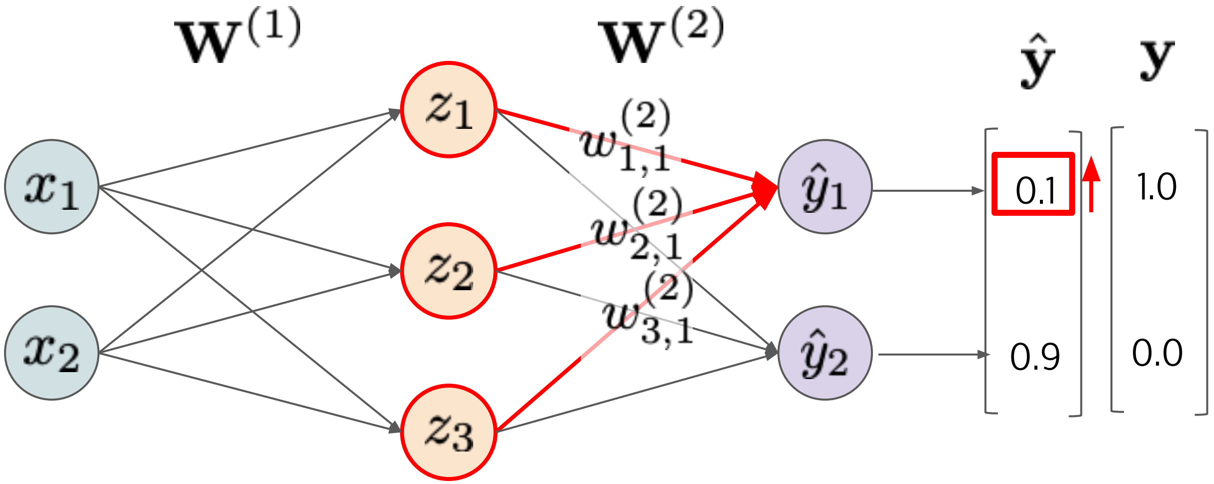
$$\hat{y}_k = \sum_{I=1}^{2} w_{i,k}^{(2)} z_i + b_k$$

To increase y1, we should

1. Increase the weights (+bias)
2. Increase the previous output z

What does it mean by updating parameters?

The output is a function of the weights (+bias) and previous outputs



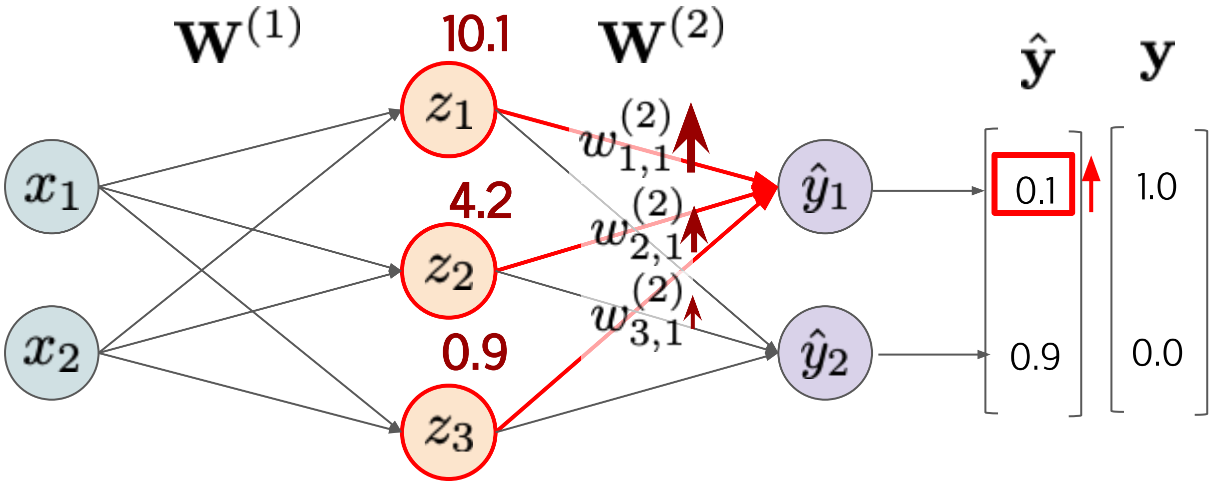
$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \mathbf{z}^{\mathrm{T}}$$

The gradient will update the parameters such that

1. **Update the weights (+bias) proportionally to z**
2. Increase the previous output z

What does it mean by updating parameters?

The output is a function of the weights (+bias) and previous outputs



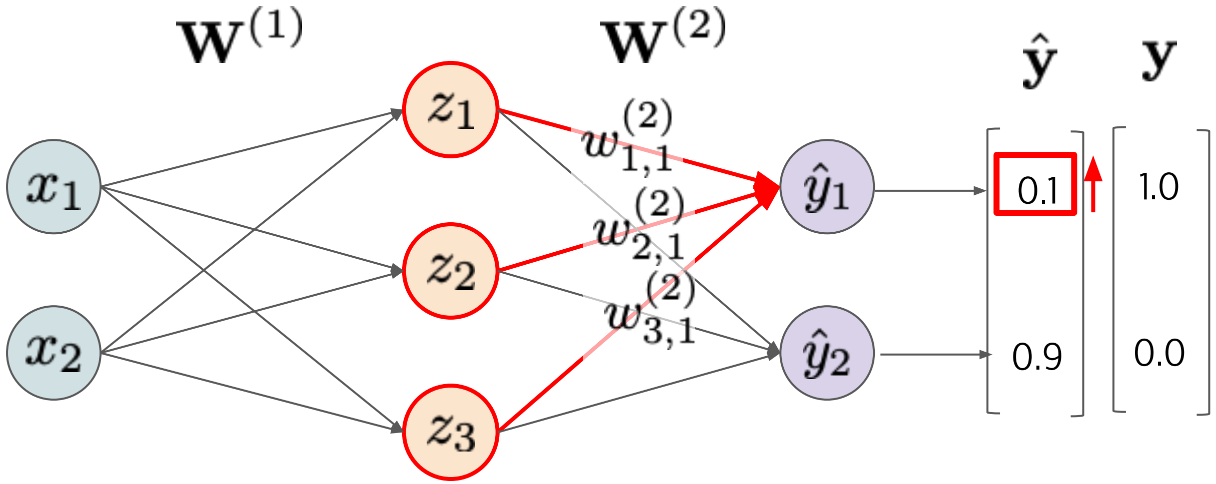
$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}} \cdot \mathbf{z}^{\mathrm{T}}$$

The gradient will update the parameters such that

1. **Update the weights (+bias) proportionally to z**
2. Increase the previous output z

What does it mean by updating parameters?

The output is a function of the weights (+bias) and previous outputs



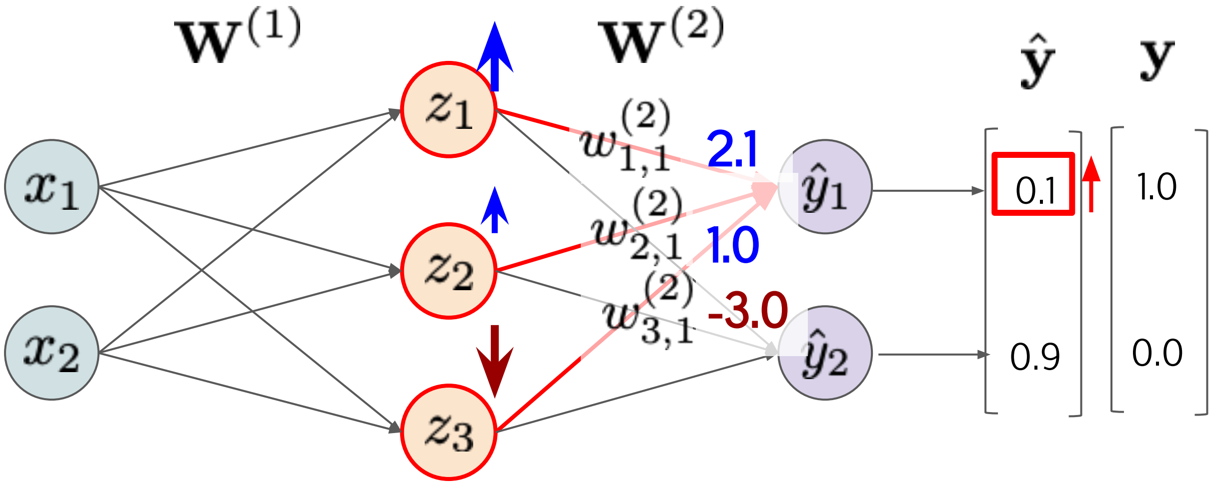
$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{z}} = \left(\mathbf{W}^{(2)}\right)^{\mathrm{T}} \cdot \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}}$$

The gradient will update the parameters such that

1. Update the weights (+bias) proportionally to z
2. **Update the activation z proportionally to weights**

What does it mean by updating parameters?

The output is a function of the weights (+bias) and previous outputs



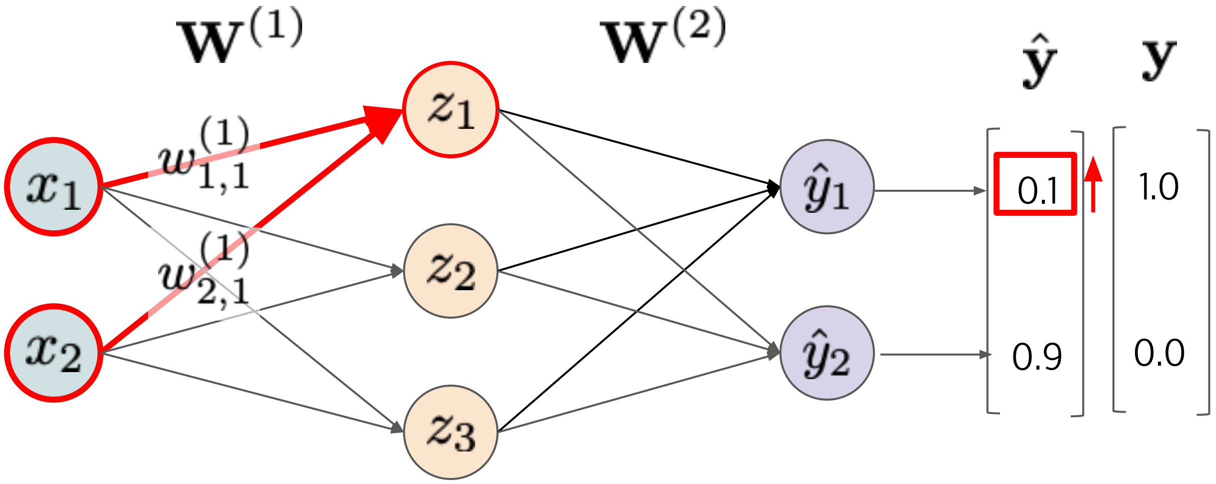
$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{z}} = \left(\mathbf{W}^{(2)}\right)^{\mathrm{T}} \cdot \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{\mathbf{y}}}$$

The gradient will update the parameters such that

1. Update the weights (+bias) proportionally to z
2. **Update the activation z proportionally to weights**

What does it mean by updating parameters?

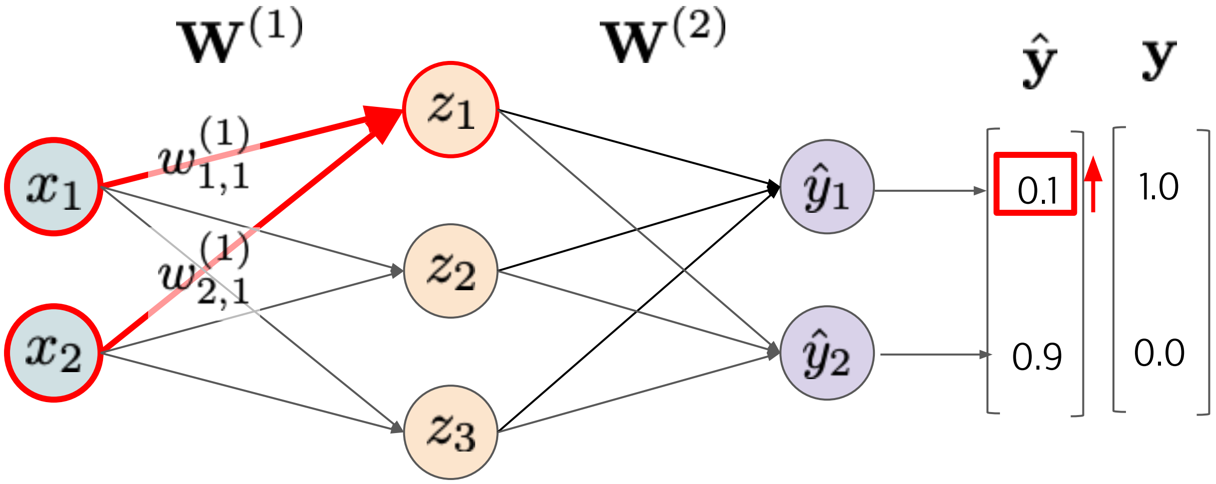
Updating the activation also depends on the previous layers



$$z_j = \sigma\left(\sum_{I=1}^{2} w_{i,j}^{(1)} x_i + b_j\right)$$

What does it mean by updating parameters?

Such information on the desired outputs are **propagated** via chain rule



$$z_j = \sigma\left(\sum_{I=1}^{2} w_{i,j}^{(1)} x_i + b_j\right) \qquad \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}$$

Summary: backpropagation

- Algorithm to compute the loss gradient w.r.t parameters
- The update signals propagate from the output to the input layers via chain rule
- Gradient always flows through the connected edges!
- It naturally encourages the neurals to be correlated
    - “Fire together, wire together”
- Assumption
    - Neural network is fully differentiable
    - What if it is not differentiable (e.g. discrete output/activation function)?

Today’s agenda

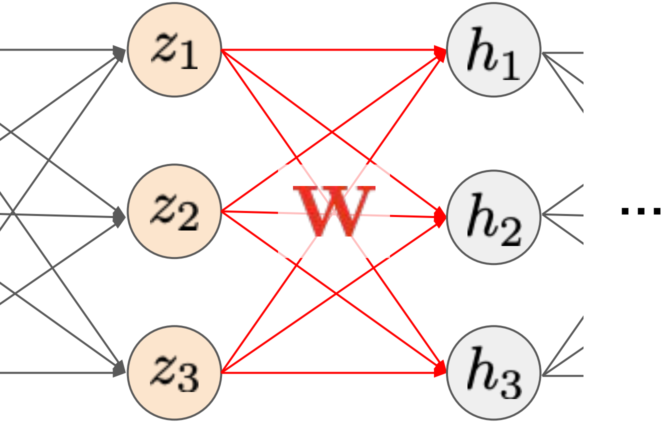
- Optimization of Neural Network
  - Backpropagation
- **Improving neural network training**
  - Normalization, initialization, regularization
- Practical tips for neural network training
  - Learning rate scheduling, hyper-parameter tuning

So far we learned that ...

- Neural network training is performed by gradient update
- What if the gradient goes wrong? (e.g. **zero-gradient**)

So far we learned that ...

- Neural network training is performed by gradient update
- What if the gradient goes wrong? (e.g. **zero-gradient**)



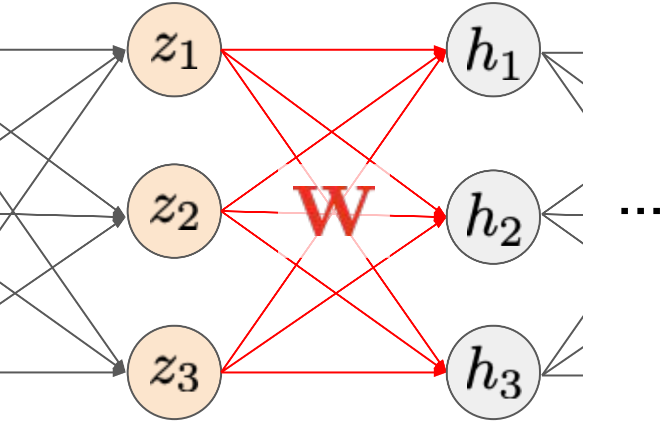
$$\mathbf{h} = \sigma\left(\mathbf{W}\mathbf{z} + \mathbf{b}\right)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot (\mathbf{z}^{\mathrm{T}})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = (\sigma'(\cdot)\mathbf{W}^{\mathrm{T}}) \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}}$$

So far we learned that ...

- Neural network training is performed by gradient update
- What if the gradient goes wrong? (e.g. **zero-gradient**)



$$\mathbf{h} = \sigma\left(\mathbf{W}\mathbf{z} + \mathbf{b}\right)$$

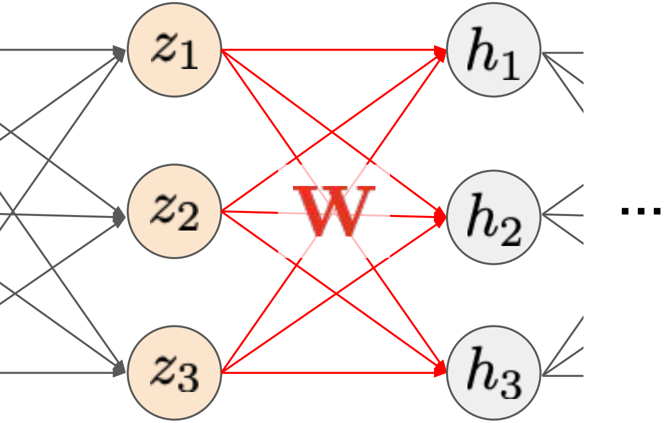
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot (\mathbf{z}^{\mathrm{T}})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = (\sigma'(\cdot)\mathbf{W}^{\mathrm{T}}) \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}}$$

Zero gradient when we reach the saddle point (local optima)
→ **good**

So far we learned that ...

- Neural network training is performed by gradient update
- What if the gradient goes wrong? (e.g. **zero-gradient**)



$$\mathbf{h} = \sigma\left(\mathbf{W}\mathbf{z} + \mathbf{b}\right)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \left(\mathbf{z}^{\mathrm{T}}\right)$$

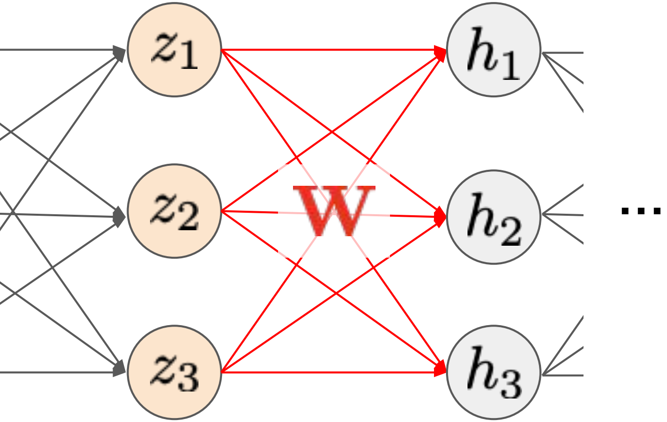
$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \left(\sigma'(\cdot)\mathbf{W}^{\mathrm{T}}\right) \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}}$$

Zero gradient when the activations (**z**) are all zero

→ **bad (no updates in the parameters)**

So far we learned that ...

- Neural network training is performed by gradient update
- What if the gradient goes wrong? (e.g. **zero-gradient**)



$$\mathbf{h} = \sigma\left(\mathbf{W}\mathbf{z} + \mathbf{b}\right)$$

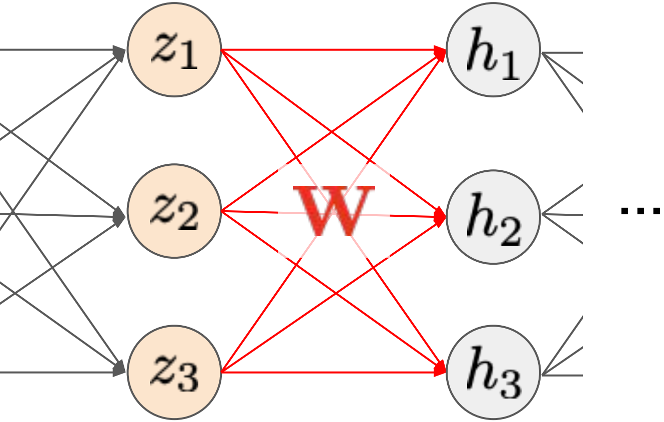
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \left(\mathbf{z}^{\mathrm{T}}\right)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \left(\sigma'(\cdot)\mathbf{W}^{\mathrm{T}}\right) \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}}$$

Zero gradient when the weights are all zero
→ bad (no downstream gradient)

So far we learned that ...

- Neural network training is performed by gradient update
- What if the gradient goes wrong? (e.g. **zero-gradient**)



$$\mathbf{h} = \sigma\left(\mathbf{W}\mathbf{z} + \mathbf{b}\right)$$

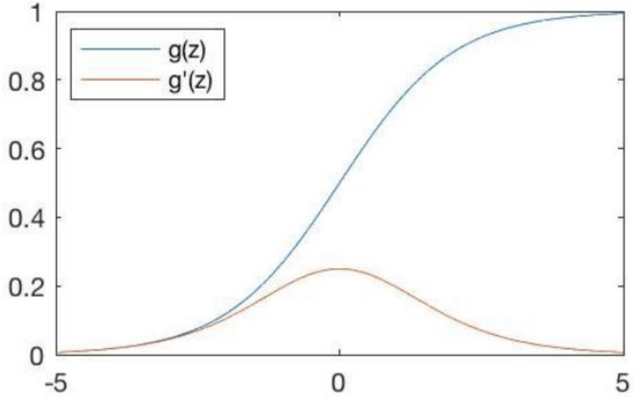
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot (\mathbf{z}^{\mathrm{T}})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = (\sigma'(\cdot)\mathbf{W}^{\mathrm{T}}) \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}}$$

Zero gradient when the derivative of nonlinear function goes zero
→ bad (no downstream gradient)

Revisiting nonlinear activation functions

- Sigmoid



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

$$g'(z) = g(z)(1 - g(z))$$

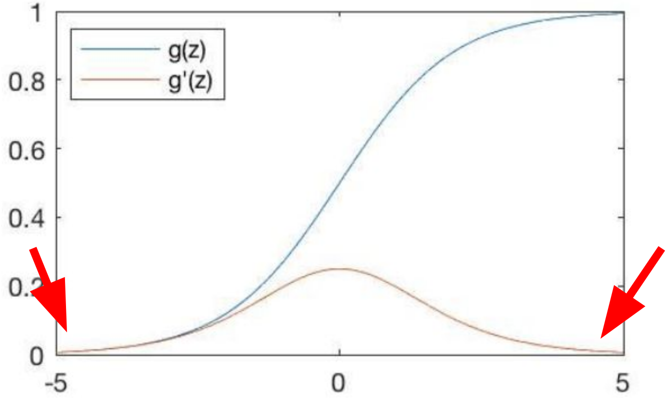
Pros

- Bounding the activation value range [0,1]

Figure credit: © MIT 6.S191: Introduction to Deep Learning

Revisiting nonlinear activation functions

- Sigmoid



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

$$g'(z) = g(z)(1 - g(z))$$

Pros

- Bounding the activation value range [0,1]

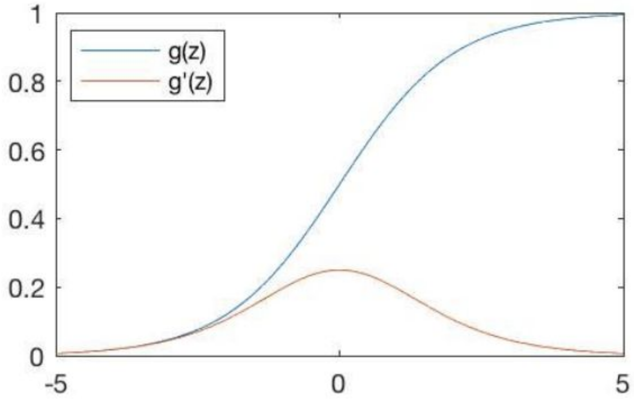
Cons

- Zero gradient on saturated neurons

Figure credit: © MIT 6.S191: Introduction to Deep Learning

Revisiting nonlinear activation functions

- Sigmoid



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

$$g'(z) = g(z)(1 - g(z))$$

Pros

- Bounding the activation value range [0,1]

Cons

- Zero gradient on saturated neurons
- Outputs are not zero-centered (always positive)

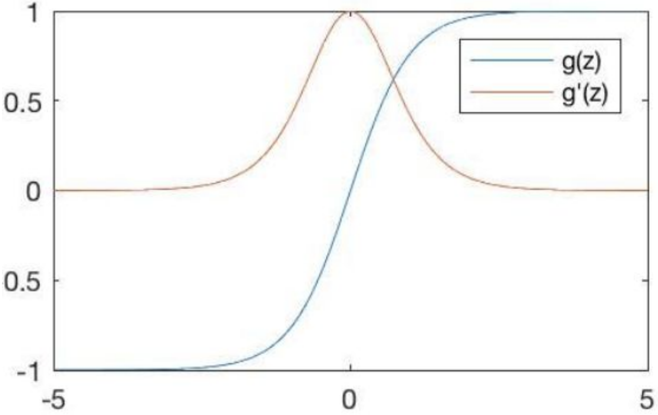
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot (\mathbf{z}^{\mathrm{T}})$$

Moves all weights toward all positive or negative direction

Figure credit: © MIT 6.S191: Introduction to Deep Learning

Revisiting nonlinear activation functions

- Hyperbolic Tangent (Tanh)



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

$$g'(z) = 1 - g(z)^2$$

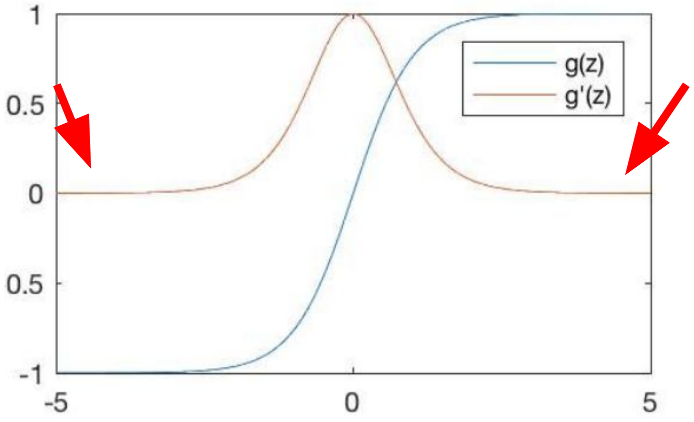
Pros

- Bounding the activation value range [-1,1]
- Outputs are zero centered

Figure credit: © MIT 6.S191: Introduction to Deep Learning

Revisiting nonlinear activation functions

- Hyperbolic Tangent (Tanh)



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

$$g'(z) = 1 - g(z)^2$$

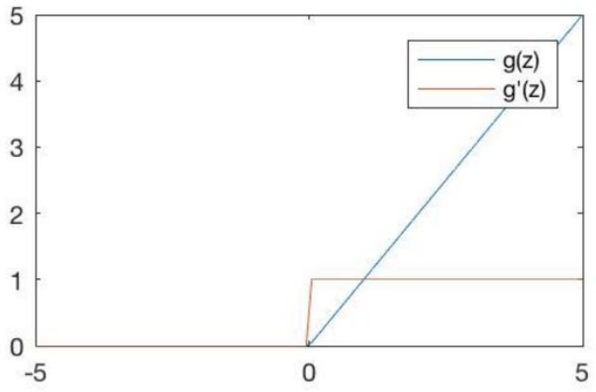
Pros
- Bounding the activation value range [-1,1]
- Outputs are zero centered

Cons
- Zero gradient on saturated neurons

Figure credit: © MIT 6.S191: Introduction to Deep Learning

Revisiting nonlinear activation functions

- Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

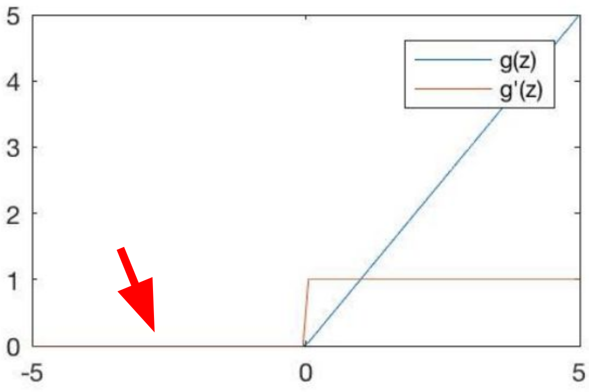
Pros

- No saturation
- Easy to compute

Figure credit: © MIT 6.S191: Introduction to Deep Learning

Revisiting nonlinear activation functions

- Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Pros

- No saturation
- Easy to compute

Cons

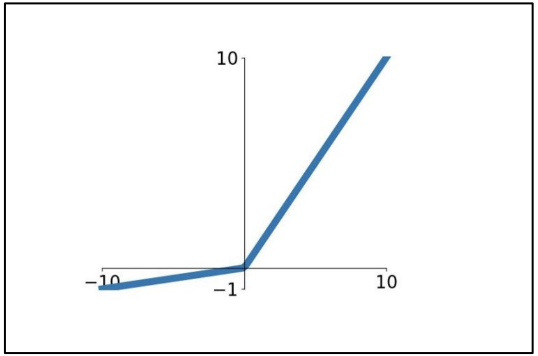
- Not zero-centered output
- Zero gradient for negative activations

Figure credit: © MIT 6.S191: Introduction to Deep Learning

Revisiting nonlinear activation functions

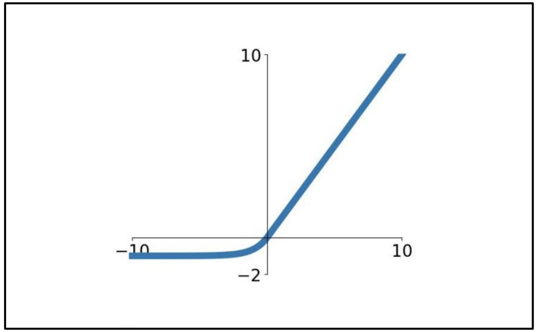
- Other activation functions

Leaky ReLU



$$f(x) = \max(0.01x, x)$$

Exponential Linear Unit (ELU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

Figure credit: © Stanford CS231n: Convolutional Neural Networks for Visual Recognition

Weight initialization

- What happens if we initialize all weights too small?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

Figure credit: © Stanford CS231n: Convolutional Neural Networks for Visual Recognition

Weight initialization

- What happens if we initialize all weights too small?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

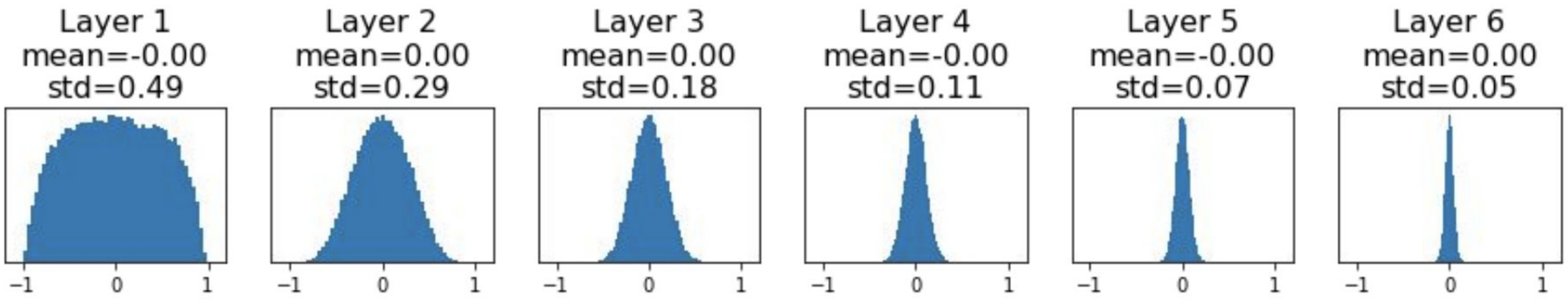


Figure credit: © Stanford CS231n: Convolutional Neural Networks for Visual Recognition

Weight initialization

- What happens if we initialize all weights too small?

```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

Almost zero activations at top layers!

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot (\mathbf{z}^{\mathrm{T}}) = 0$$

→ No learning!

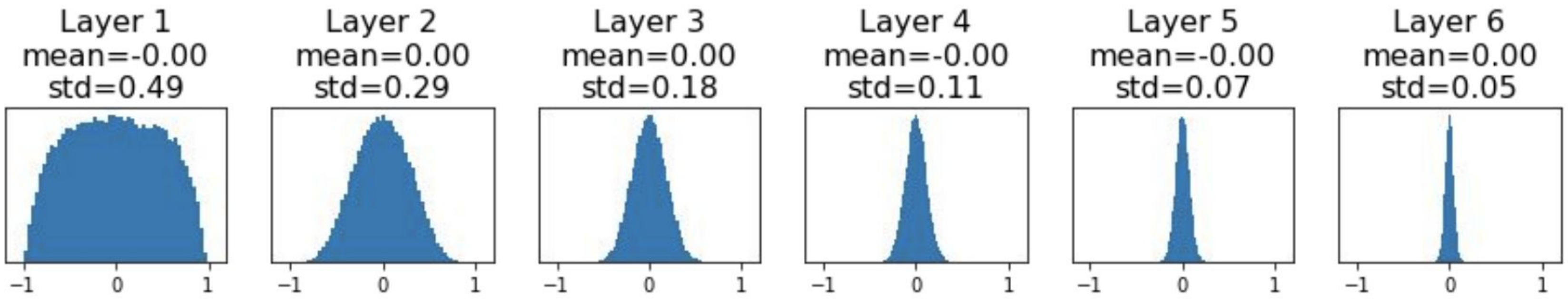


Figure credit: © Stanford CS231n: Convolutional Neural Networks for Visual Recognition

Weight initialization

- What happens if we initialize all weights too **big**?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

Figure credit: © Stanford CS231n: Convolutional Neural Networks for Visual Recognition

Weight initialization

- What happens if we initialize all weights too **big**?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

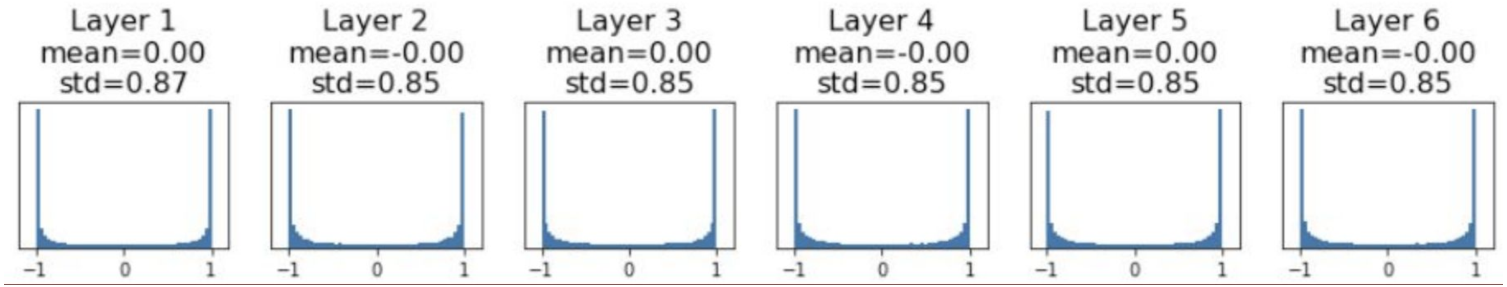


Figure credit: © Stanford CS231n: Convolutional Neural Networks for Visual Recognition

Weight initialization

- What happens if we initialize all weights too **big**?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

Almost zero gradient due to saturation in nonlinear function!

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = (\overset{=0}{\sigma'(\cdot)}\mathbf{W}^{\mathrm{T}}) \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}}$$

→ No learning!

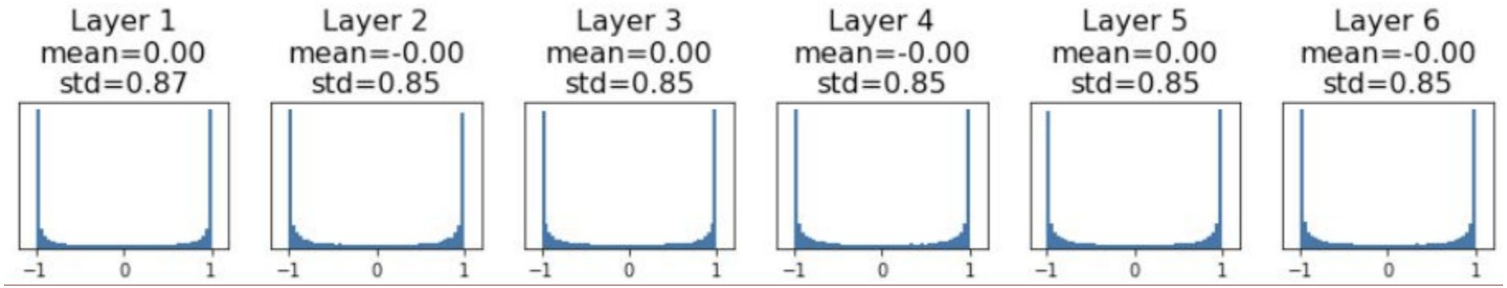


Figure credit: © Stanford CS231n: Convolutional Neural Networks for Visual Recognition

Weight initialization

- **Xavier** (or Glorot) initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

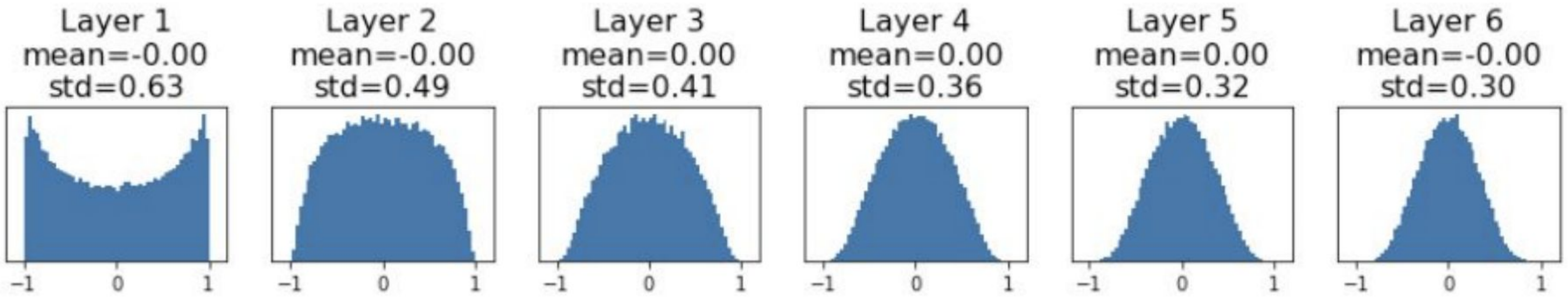


Figure credit: © Stanford CS231n: Convolutional Neural Networks for Visual Recognition

Normalizing activations

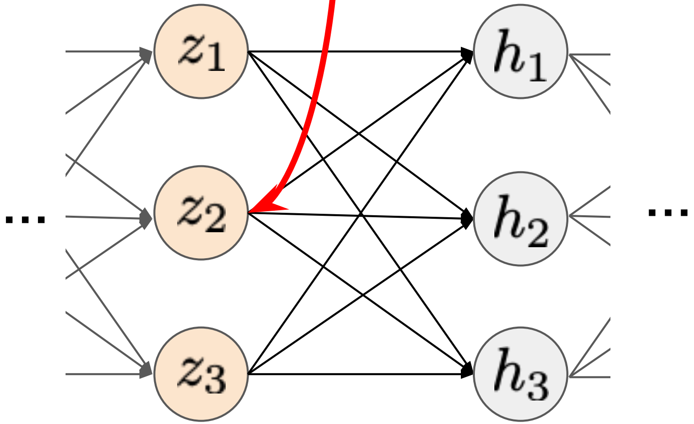
- Seems like normalizing the activations at every layer to standard Gaussian is a good option for optimization
- What if we do this **explicitly**?

Normalizing activations

- Standard normalization

$$\hat{\mathbf{z}} = \frac{\mathbf{z} - \mathbb{E}[\mathbf{z}]}{\sqrt{\mathrm{Var}(\mathbf{z})}}$$

Normalize every outputs using the mean and variance of the **minibatch**



Normalizing activations

- Batch normalization

Generalize the standard normalization using the **learnable parameters** (scaling and shift vectors)

$$\hat{\mathbf{z}} = \gamma \bar{\mathbf{z}} + \beta$$

where

$$\bar{\mathbf{z}} = \frac{\mathbf{z} - \mathbb{E}[\mathbf{z}]}{\sqrt{\mathrm{Var}(\mathbf{z})}}$$

Normalizing activations

- Batch normalization

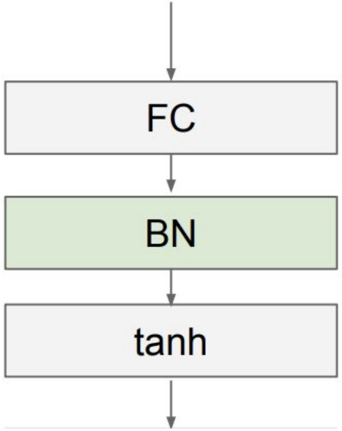
Generalize the standard normalization using the **learnable parameters** (scaling and shift vectors)

$$\hat{\mathbf{z}} = \gamma\bar{\mathbf{z}} + \beta$$

where

$$\bar{\mathbf{z}} = \frac{\mathbf{z} - \mathbb{E}[\mathbf{z}]}{\sqrt{\mathrm{Var}(\mathbf{z})}}$$

Usually injected before every nonlinear activation functions



Summary: Improving neural network training

- Check the gradient!
  - Zero gradient = no learning
  - Gradient can go wrong for various reasons (initialization, nonlinear activation functions, …)
- Design neural network carefully
  - Xavier initialization is usually good
  - ReLU + Batch Normalization is usually a good starting point