

Neural Networks Training, SGD and Backpropagation

Machine Learning Course - CS-433

4 Nov 2025

Robert West

(Slide credits: Nicolas Flammarion)



Recap

Neural Networks: Key Facts

Supervised learning: we observe some data $S_{\text{train}} = \{x_n, y_n\}_{n=1}^N \in \mathcal{X} \times \mathcal{Y}$

→ given a new x , we want to predict its label y

Linear prediction (with augmented features): $y = f_{\text{Lin}}(x) = \phi(x)^T w$

Features are given

Prediction with a NN:

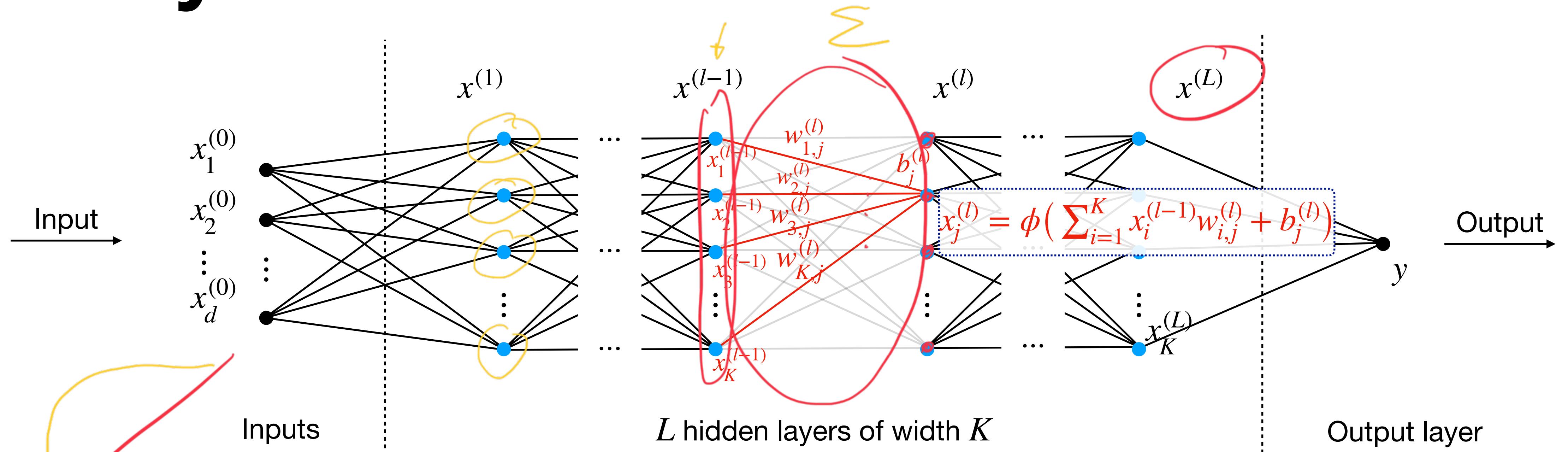
$$y = f_{\text{NN}}(x) = f(x)^T w$$

Function implemented by the NN
parameters: weights and biases

First layers transform the input
into a good representation

Last layer is performing
a linear prediction

Fully Connected Neural Networks



$$x_j^{(l)} = \phi\left(\sum_{i=1}^K x_i^{(l-1)} w_{i,j}^{(l)} + b_j^{(l)}\right)$$

Important: ϕ is non-linear
otherwise we can only
represent linear functions

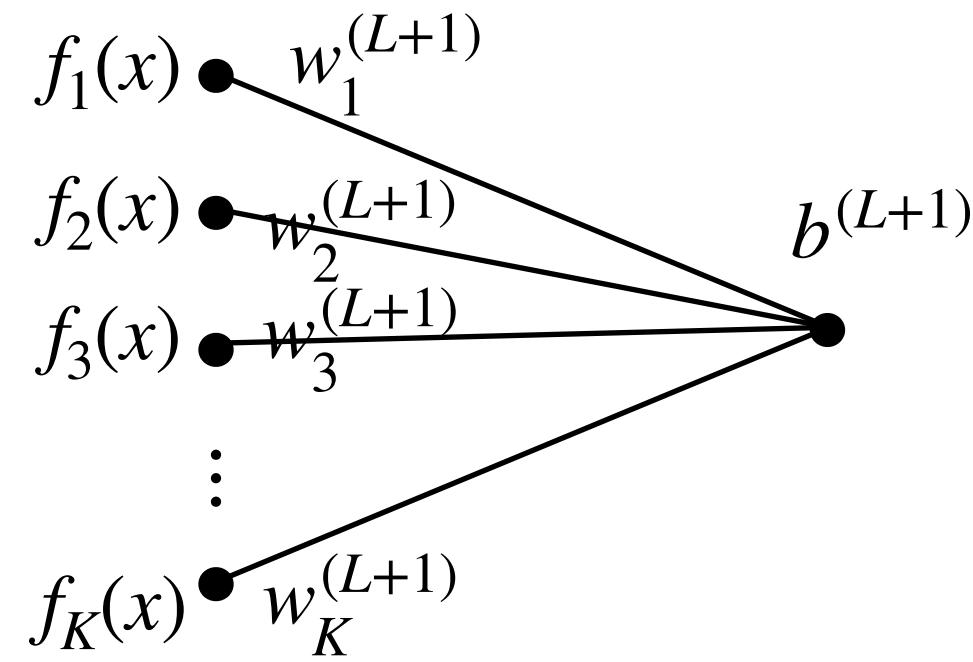
weight of the edge going from node i
in layer $l-1$ to node j in layer l

bias term associated with
node j in layer l

NNs: Inference vs. Training

Linear prediction on features $f(x)$

$$h(x) = f(x)^T w^{(L+1)} + b^{(L+1)}$$



Regression
with $y \in \mathbb{R}$

Inference

$$h(x)$$

Training

$$\ell(y, h(x)) = (h(x) - y)^2$$

Binary Classification
with $y \in \{-1, 1\}$

$$\text{sign}(h(x))$$

Multi-Class Classification
with $y \in \{1, \dots, K\}$

$$\text{argmax}_{c \in \{1, \dots, K\}} h(x)_c$$

$$\ell(y, h(x)) = \log(1 + \exp(-yh(x)))$$

$$\ell(y, h(x)) = -\log \frac{e^{h(x)y}}{\sum_{i=1}^K e^{h(x)_i}}$$

With a suitable representation of the data $f(x)$ learned by the network, the last layer only performs a linear regression or classification step

Today: How do we train a NN?

Training of NNs

Training loss for a regression problem with $S_{\text{train}} = \{(x_n, y_n)\}_{n=1}^N$:

$$\mathcal{L}(f) = \frac{1}{2N} \sum_{n=1}^N \underbrace{(y_n - f(x_n))^2}_{\text{error}}$$

where f is the function represented by a NN with weights $(w_{i,j}^{(l)})$ and biases $(b_i^{(l)})$

Task:

$$\min_{w_{i,j}^{(l)}, b_i^{(l)}} \mathcal{L}(f) + \lambda \sum_{i,j,l} w_{ij}^{(l)} 2$$

L_2 norm

Remarks:

- Regularization can be added to avoid overfitting and is easy to implement
- Non-convex optimization problem
→ not guaranteed to converge to a global minimum

Training of NNs with SGD

SGD algorithm: Uniformly sample n , compute the gradient of $\mathcal{L}_n = \frac{1}{2}(y_n - f(x_n))^2$ to update:

$$(w_{i,j}^{(l)})_{t+1} = (w_{i,j}^{(l)})_t - \gamma \frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} \quad (b_i^{(l)})_{t+1} = (b_i^{(l)})_t - \gamma \frac{\partial \mathcal{L}_n}{\partial b_i^{(l)}}$$

In Practice: Step size schedule, mini-batch, momentum, Adam

Training of NNs with SGD

SGD algorithm: Uniformly sample n , compute the gradient of $\mathcal{L}_n = \frac{1}{2}(y_n - f(x_n))^2$ to update:

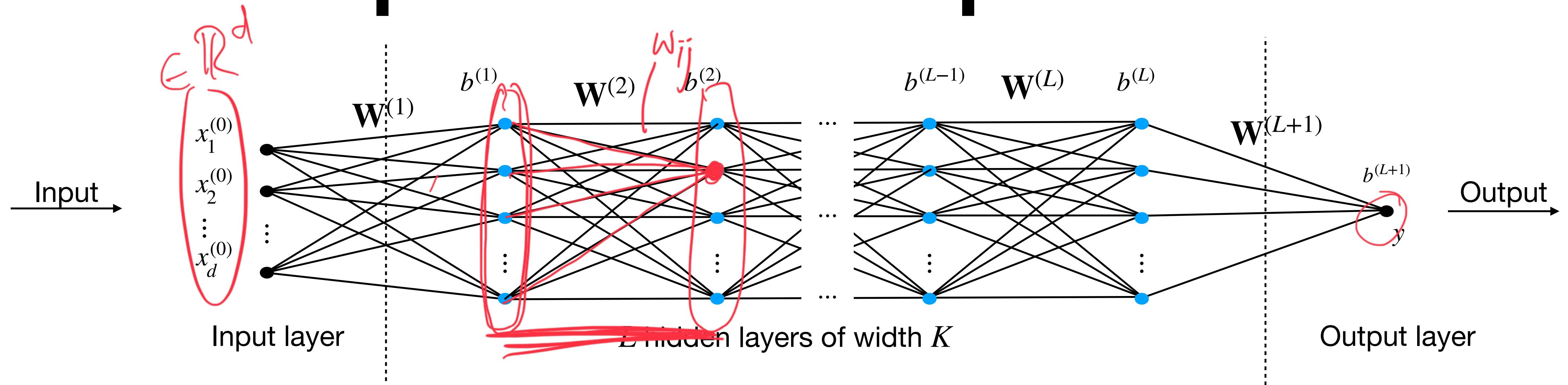
$$(w_{i,j}^{(l)})_{t+1} = (w_{i,j}^{(l)})_t - \gamma \frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} \quad (b_i^{(l)})_{t+1} = (b_i^{(l)})_t - \gamma \frac{\partial \mathcal{L}_n}{\partial b_i^{(l)}}$$

In Practice: Step size schedule, mini-batch, momentum, Adam

Problem: With $\cancel{O(K^2 L)}$ parameters, applying chain-rules **independently** is inefficient due to the compositional structure of f

Solution: the **Backpropagation algorithm** computes gradients via the chain rule but reuses intermediate computations

Description of NN parameters



Weight matrices: $\mathbf{W}^{(l)}$ such that $\mathbf{W}_{i,j}^{(l)} = w_{i,j}^{(l)}$, of size

- $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times K}$
- $\mathbf{W}^{(l)} \in \mathbb{R}^{K \times K}$ for $2 \leq l \leq L$
- $\mathbf{W}^{(L+1)} \in \mathbb{R}^K$

$\not\in$

Bias vectors: $b^{(l)}$ such that the i -th component is $b_i^{(l)}$

- $b^{(l)} \in \mathbb{R}^K$ for $1 \leq l \leq L$
- $b^{(L+1)} \in \mathbb{R}$

Compact description of output

The functions implemented by each layer can be written as:

$$\bullet \underline{x^{(1)}} = f^{(1)}(\underline{x^{(0)}}) := \phi((\mathbf{W}^{(1)})^\top \underline{x^{(0)}} + b^{(1)})$$

...

$$\bullet \underline{x^{(l)}} = f^{(l)}(\underline{x^{(l-1)}}) := \phi((\mathbf{W}^{(l)})^\top \underline{x^{(l-1)}} + b^{(l)})$$

...

$$\bullet y = f^{(L+1)}(\underline{x^{(L)}}) := (\mathbf{W}^{(L+1)})^\top \underline{x^{(L)}} + b^{(L+1)}$$

$$f \circ g(x) = f(g(x))$$

The overall function $y = f(x^{(0)})$ is just the composition of the layer functions:

$$\min_{w^{(e)}, b^{(e)}, \lambda} \mathcal{L}(f)$$

$$f = f^{(L+1)} \circ f^{(L)} \circ \dots \circ f^{(l)} \circ \dots \circ f^{(2)} \circ f^{(1)}$$



Cost function

Cost function:

$$\mathcal{L} = \frac{1}{2N} \sum_{n=1}^N (y_n - f^{(L+1)} \circ \dots \circ f^{(2)} \circ f^{(1)}(x_n))^2$$

Remarks:

- The specific form of the loss is not crucial
- \mathcal{L} is a function of all weight matrices and bias vectors
- Each function $f^{(l)}$ is parameterized by weights $\mathbf{W}^{(l)}$ and biases $b^{(l)}$

Individual loss for SGD:

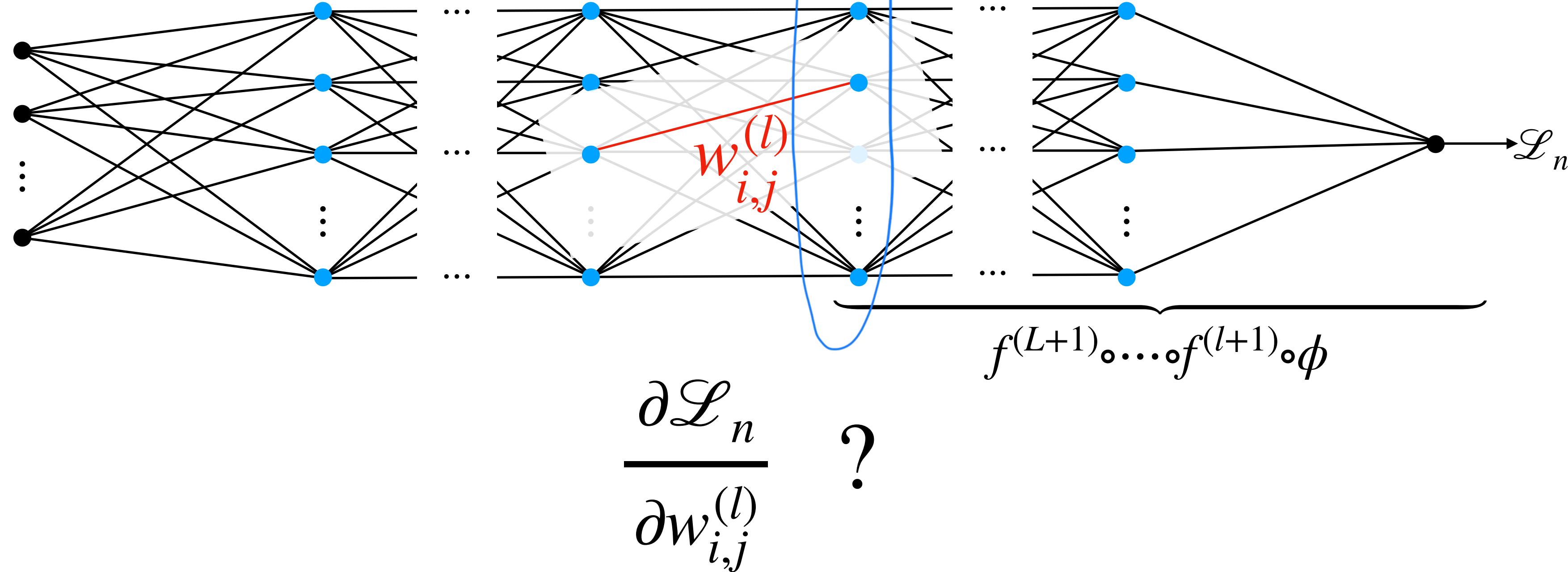
$$\mathcal{L}_n = \frac{1}{2} (y_n - f^{(L+1)} \circ \dots \circ f^{(2)} \circ f^{(1)}(x_n))^2$$

Goal: Compute for all (i, j, l)

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} \quad \text{and} \quad \frac{\partial \mathcal{L}_n}{\partial b_i^{(l)}}$$

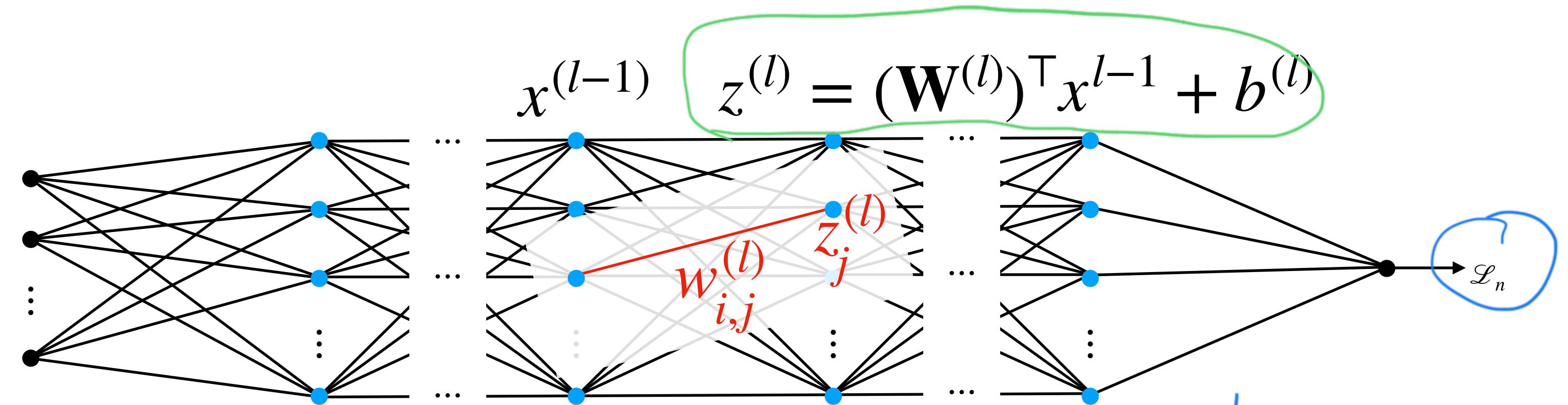
Chain rule

$$\mathcal{L}_n = \frac{1}{2} (y_n - f^{(L+1)} \circ \dots \circ f^{(l+1)} \circ \phi((\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + b^{(l)}))^2$$



Chain rule

$$\mathcal{L}_n = \frac{1}{2} (y_n - f^{(L+1)} \circ \dots \circ f^{(l+1)} \circ \phi(z^{(l)}))^2$$



Apply the chain rule:

$$\begin{aligned} \frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} &= \sum_{k=1}^K \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} \\ &= \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}} \\ &= \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \cdot s_j^{(l)} \end{aligned}$$

$$\text{since } \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = 0 \text{ for } k \neq j$$

$$\text{since } z_j^{(l)} = \sum_{k=1}^K w_{k,j}^{(l)} x_k^{(l-1)} + b_j^{(l)}$$

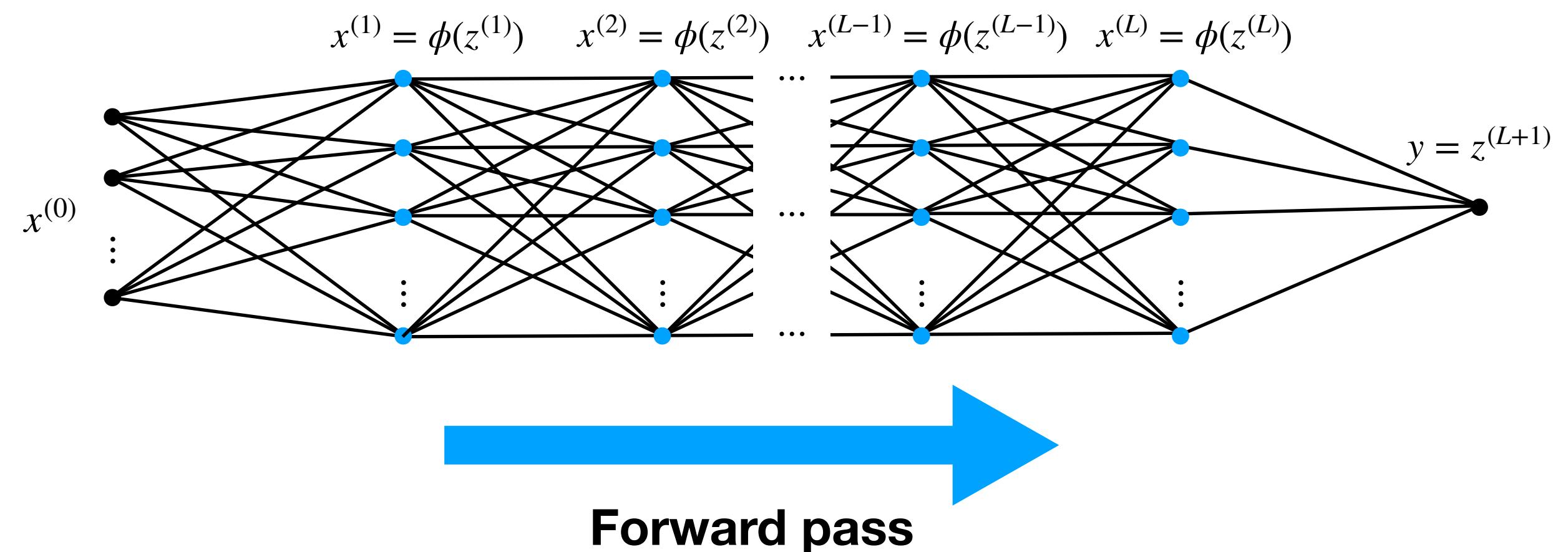
$$\begin{aligned} \frac{d}{dx} f(g(x)) &= f'(g(x)) \cdot g'(x) \\ &= \frac{df(g(x))}{dg(x)} \cdot \frac{dg(x)}{dx} \end{aligned}$$

We need to compute $\frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}$, $z^{(l)}$, $x_i^{(l-1)}$ and reuse them for different $\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}}$

Forward Pass

We can compute $z^{(l)}$ and $x^{(l)}$ by a forward pass in the network:

$$\begin{aligned}x^{(0)} &= x_n \in \mathbb{R}^d \\z^{(l)} &= (\mathbf{W}^{(l)})^\top x^{(l-1)} + b^{(l)} \\x^{(l)} &= \phi(z^{(l)})\end{aligned}$$

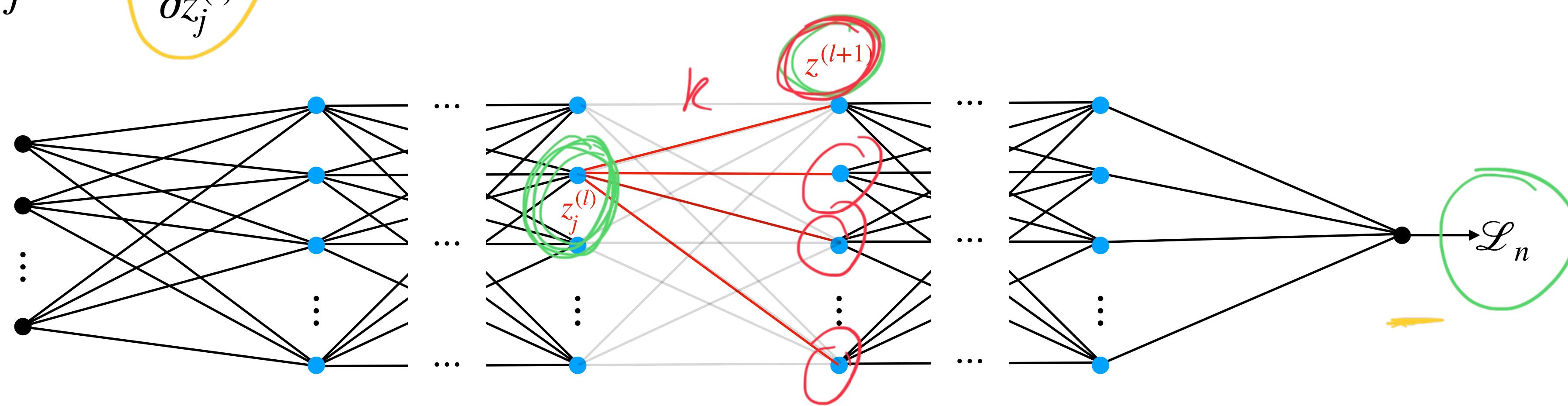


Computational complexity:

- one pass over the network $O(K^2L)$

Backward pass (I)

Define $\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}$



Chain rule:

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

The diagram shows the chain rule application. A blue bracket labeled l encloses the term $\frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}$. A red arrow points to the term $\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$. A green oval highlights the term $\frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}}$. A yellow oval highlights the term $\delta_k^{(l+1)}$.

Backward pass (II)

Using $z_k^{(l+1)} = \sum_{i=1}^K w_{i,k}^{(l+1)} x_i^{(l)} + b_k^{(l+1)} = \sum_{i=1}^K w_{i,k}^{(l+1)} \phi(z_i^{(l)}) + b_k^{(l+1)}$

We obtain $\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \phi'(z_j^{(l)}) w_{j,k}^{(l+1)}$

Thus

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} \phi'(z_j^{(l)}) w_{j,k}^{(l+1)}$$

It can be written in vector form:

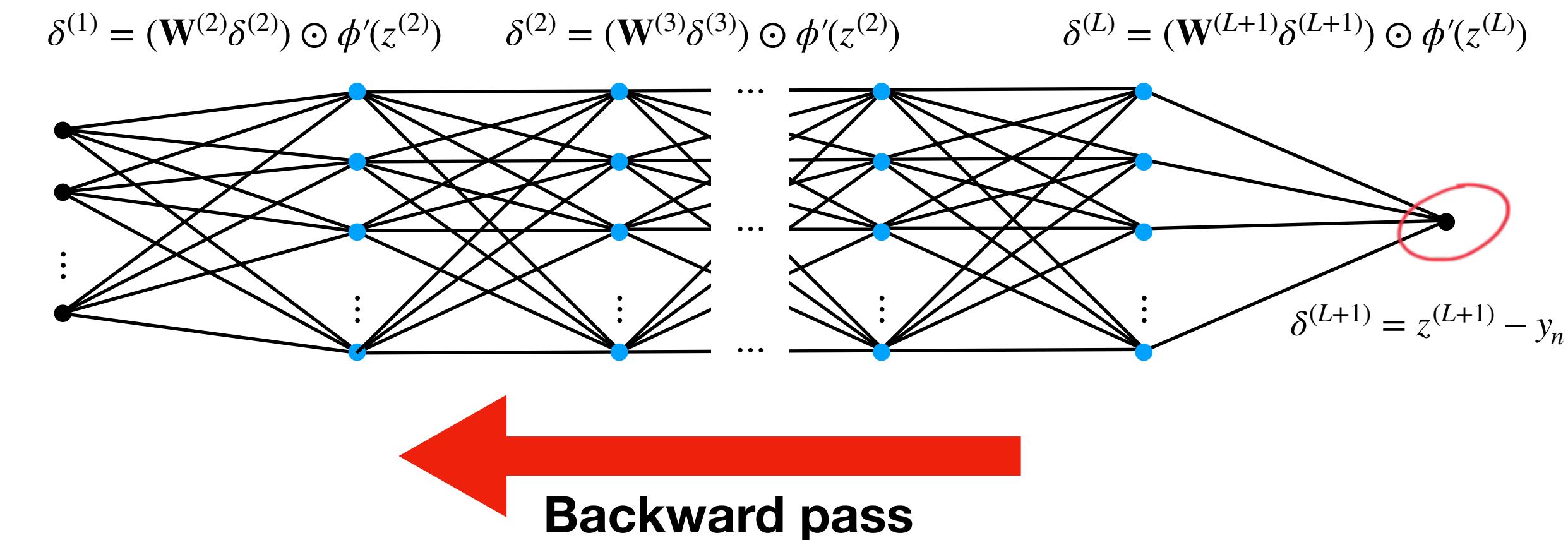
$$\delta^{(l)} = (\mathbf{W}^{(l+1)} \delta^{(l+1)}) \odot \phi'(z^{(l)})$$

⊗: Hadamard product, i.e.,
pointwise multiplication of vector

Backward pass (III)

Initialization:

$$\begin{aligned}\delta^{(L+1)} &= \frac{\partial}{\partial z^{(L+1)}} \frac{1}{2} (y_n - z^{(L+1)})^2 \\ &= z^{(L+1)} - y_n\end{aligned}$$

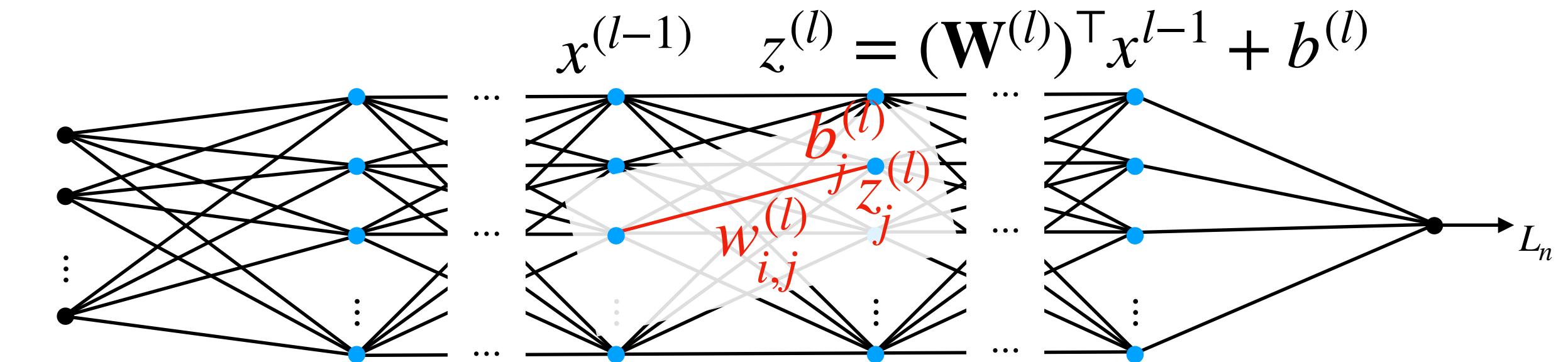


Compute all the $\delta^{(l)}$ by a backward pass in the network:

$$\delta^{(l)} = (\mathbf{W}^{(l+1)} \delta^{(l+1)}) \odot \phi'(z^{(l)})$$

Computational complexity: one pass over the network $O(K^2 L)$

Derivatives computation



Using that $z_m^{(l)} = \sum_{k=1}^K w_{k,m}^{(l)} x_k^{(l-1)} + b_m^{(l)}$:

- $\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \sum_{k=1}^K \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)}$
- $\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^K \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} \cdot x_i^{(l-1)}$

Backpropagation algorithm

Forward pass:

$$x^{(0)} = x_n \in \mathbb{R}^d$$

$$z^{(l)} = (\mathbf{W}^{(l)})^\top x^{(l-1)} + b^{(l)}$$

$$x^{(l)} = \phi(z^{(l)})$$

Backward pass:

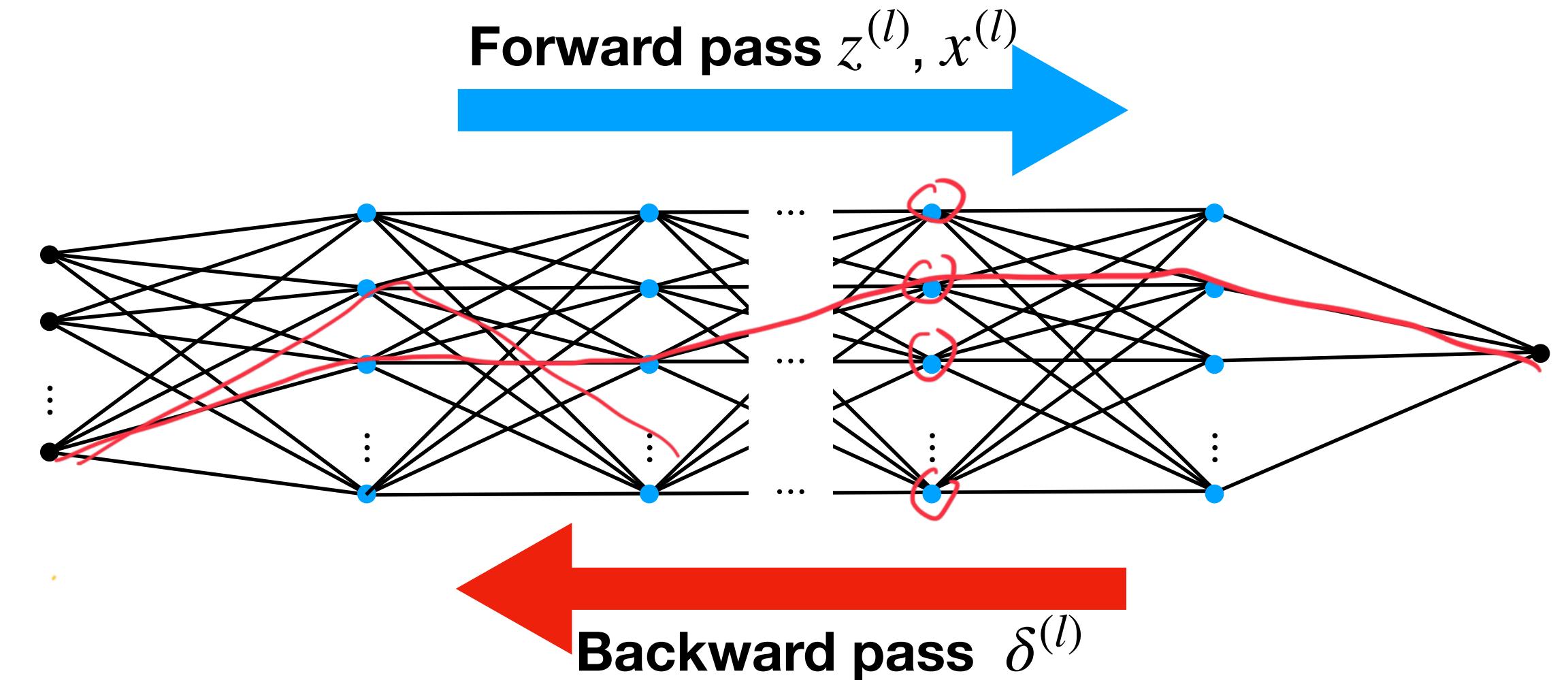
$$\delta^{(L+1)} = z^{(L+1)} - y_n$$

$$\delta^{(l)} = (\mathbf{W}^{(l+1)} \delta^{(l+1)}) \odot \phi'(z^{(l)})$$

Compute the derivatives:

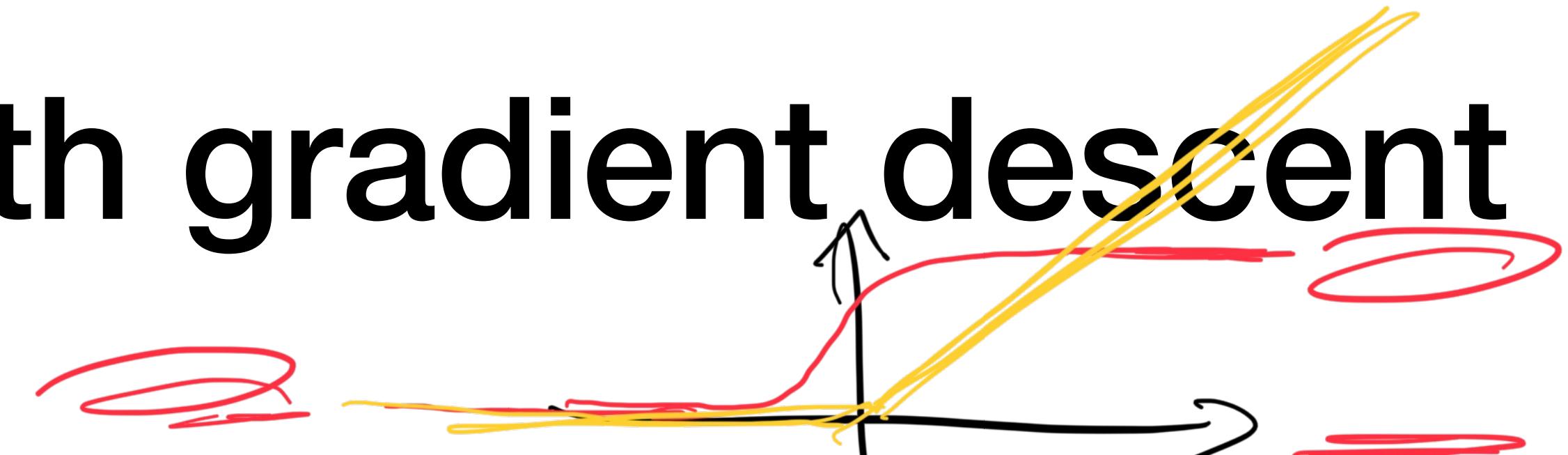
$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} x_i^{(l-1)}$$

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \delta_j^{(l)}$$



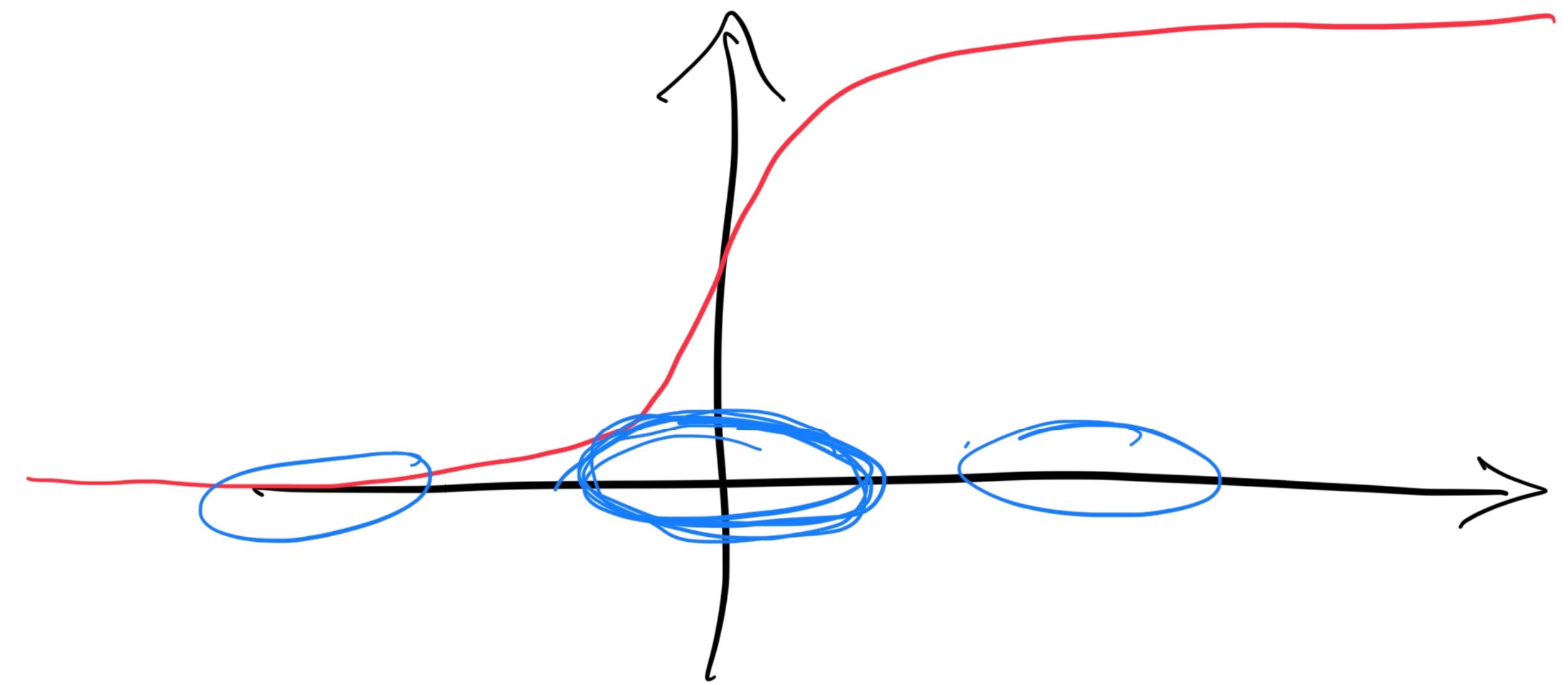
Overall Complexity: $O(K^2 L)$

Common issues with gradient descent



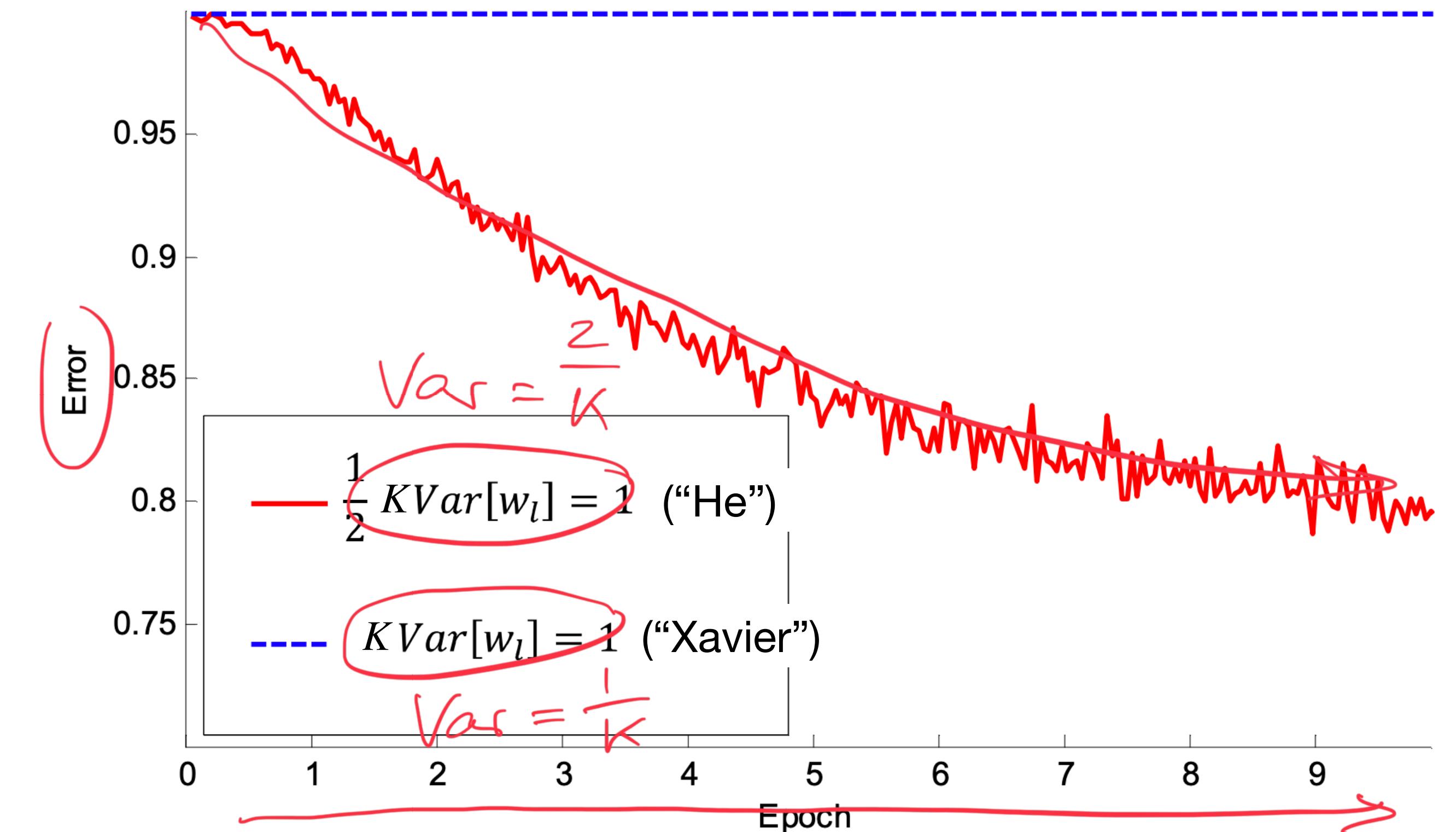
- Gradient exploding / vanishing: As the network depth L increases, the gradient magnitude can decrease or grow uncontrollably, slowing the training process.
- Cause: Back propagation uses chain rule. When there are L times multiplication of small or big values, gradients decrease or grow exponentially.
- Remedy: Effective strategies include choosing suitable activation functions, using weight normalization, initializing weights properly, and implementing skip connections.

Parameter Initialization



Importance of Parameter Initialization

- In deep networks, improper parameter initialization can lead to the **vanishing or exploding gradients problem**
- **Problem:** Extremely slow or unstable optimization
- **Solution:** Control the layerwise variance of neurons (aka **He initialization**)
- **Note:** As illustrated, even a two-fold difference in the scale of initialization can be crucial



Source: [Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification \(CVPR 2015\)](#)

Variance-Preserving Initialization

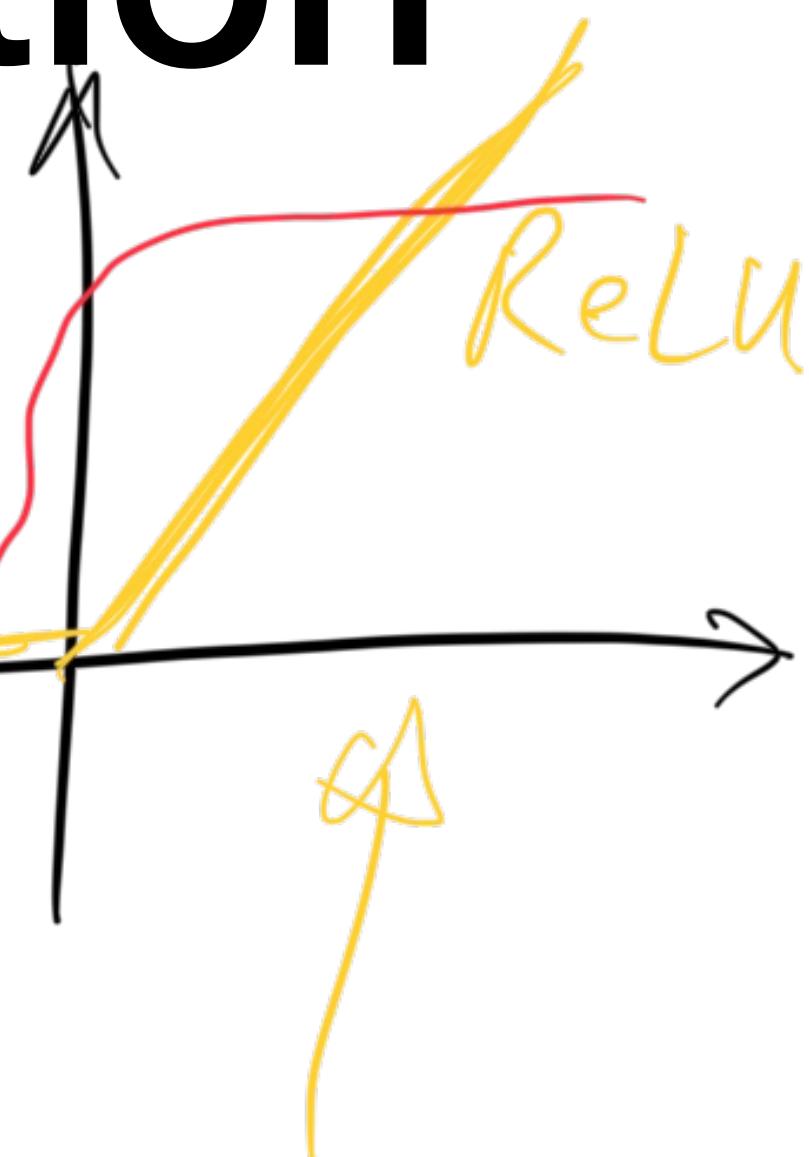
Variance-preserving initialization for ReLU networks:

- $z^{(l)} \sim \mathcal{N}(0, \mathbf{I}_K)$: pre-activations at layer l (note: $\text{Var}[z_i^{(l)}] = 1$)
- $w_i^{(l+1)} \sim \mathcal{N}(0, \sigma^2 \mathbf{I}_K)$: the i -th weight vector at layer $l + 1$
- $z_i^{(l+1)} = \text{ReLU}(z^{(l)})^\top w_i^{(l+1)}$: the i -th pre-activation at layer $l + 1$

Question: How should we set σ so that $\text{Var}[z_i^{(l+1)}] = 1$?

Answer: $\sigma = \sqrt{2/K}$

Derivation: Refer to the exercise for the derivation



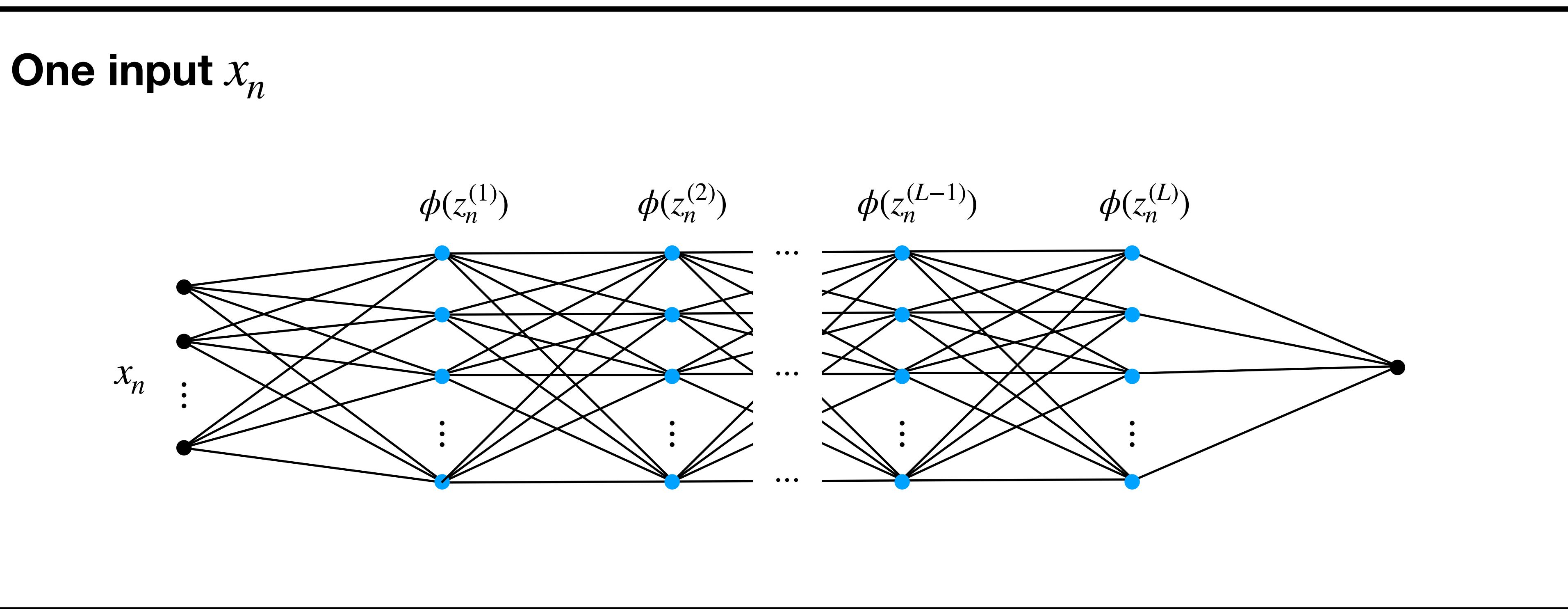
Normalization Layers

Batch Normalization

Consider a batch $B = (x_1, \dots, x_M)$ and denote by $z_n^{(l)}$ the layer's pre-activation input corresponding to the observation x_n

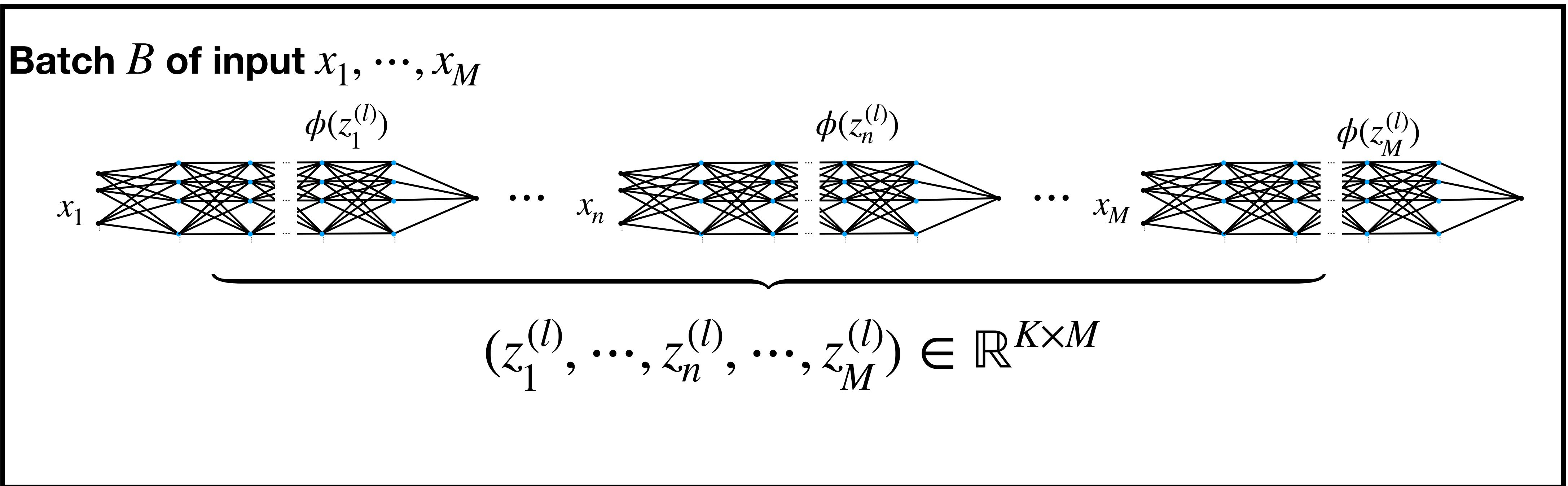
Batch Normalization

Consider a batch $B = (x_1, \dots, x_M)$ and denote by $z_n^{(l)}$ the layer's pre-activation input corresponding to the observation x_n



Batch Normalization

Consider a batch $B = (x_1, \dots, x_M)$ and denote by $z_n^{(l)}$ the layer's pre-activation input corresponding to the observation x_n



Batch Normalization

Consider a batch $B = (x_1, \dots, x_M)$ and denote by $z_n^{(l)}$ the layer's pre-activation input corresponding to the observation x_n

Step 1: Normalize each layer's input using its mean and its variance over the batch:

$$\bar{z}_n^{(l)} = \frac{z_n^{(l)} - \mu_B^{(l)}}{\sqrt{(\sigma_B^{(l)})^2 + \varepsilon}}$$

Component-wise

where $\mu_B^{(l)} = \frac{1}{M} \sum_{n=1}^M z_n^{(l)}$ and $(\sigma_B^{(l)})^2 = \frac{1}{M} \sum_{n=1}^M (z_n^{(l)} - \mu_B^{(l)})^2$, and $\varepsilon \in \mathbb{R}_{\geq 0}$ is a small value added for numerical stability

Step 2: Introduce learnable parameters $\gamma^{(l)} \in \mathbb{R}^K$ (scale) and $\beta^{(l)} \in \mathbb{R}^K$ (shift) to be able to recover the original activations if needed:

$$\hat{z}_n^{(l)} = \gamma^{(l)} \odot \bar{z}_n^{(l)} + \beta^{(l)}$$

Batch Normalization

Scale-invariance: For $\varepsilon \approx 0$, the output is invariant to activation-wise affine scaling of $z_n^{(l)}$

$$\text{BN}(a \odot z_n^{(l)} + b) = \text{BN}(z_n^{(l)}) \text{ for } a \in \mathbb{R}_{>0}^K \text{ and } b \in \mathbb{R}^K$$

Thus, for example, there is no need to include a bias *before* BatchNorm.

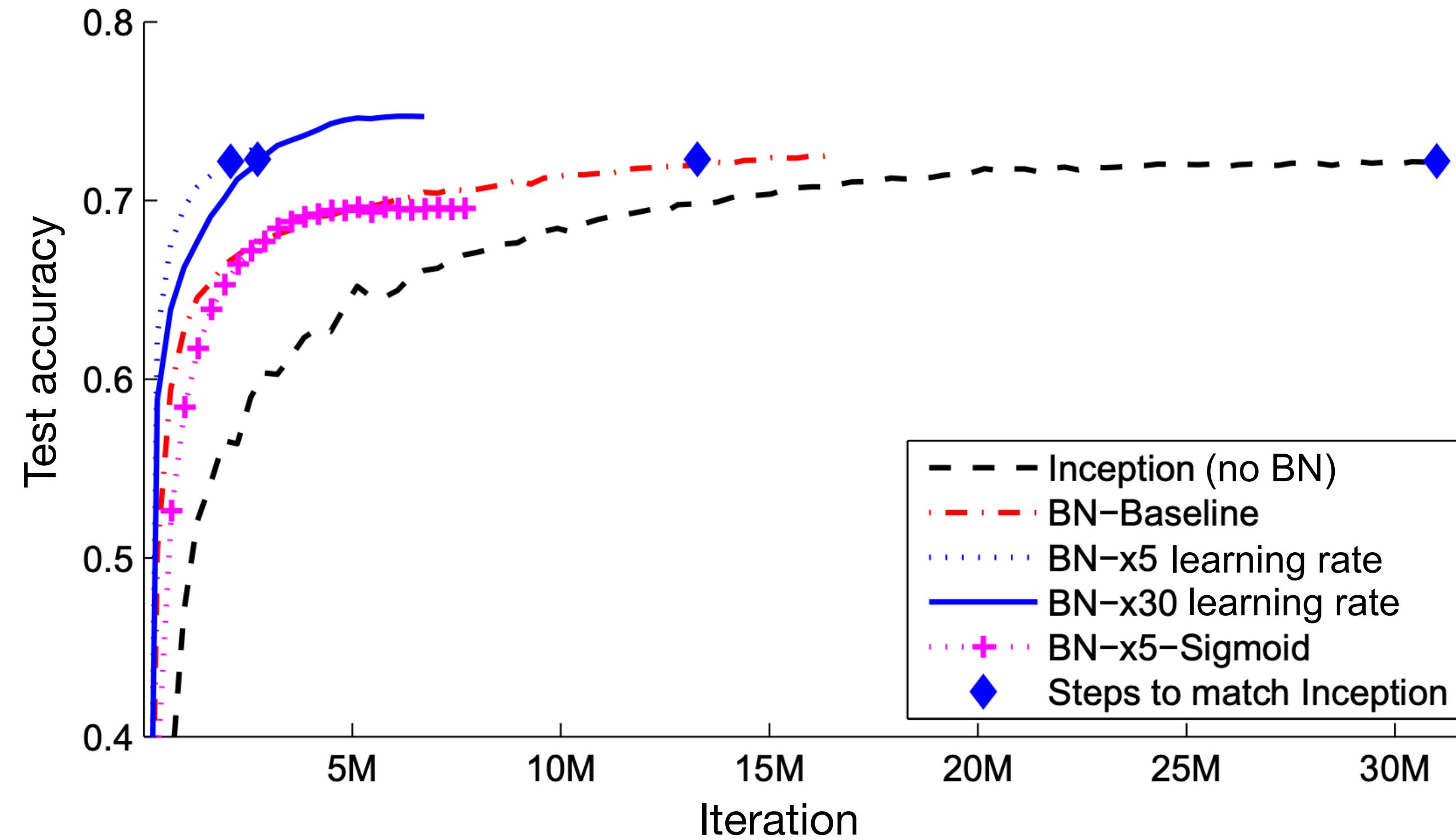
Inference: Use *fixed* mean and variance for normalization, as samples may arrive one at a time

- Estimate $\hat{\mu}^{(l)} = \mathbf{E}_{B \sim \mathcal{D}^{Train}}[\mu_B^{(l)}]$ and $\hat{\sigma}^{(l)} = \mathbf{E}_{B \sim \mathcal{D}^{Train}}[\sigma_B^{(l)}]$ during training, use these for inference
- Exponential moving averages are commonly used in practice

Implementation:

- Requires sufficiently large batches to get good estimates of $\mu_B^{(l)}, \sigma_B^{(l)}$
- BatchNorm is applied a bit differently for non-fully-connected nets (see the pytorch docs for CNNs)
- In PyTorch, switch modes by using `model.train()` for training and `model.eval()` for inference

Batch Normalization - Results



Source: [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift \(ICML 2015\)](#)

- BatchNorm leads to **much faster convergence**
- BatchNorm allows to use **much larger learning rates** (up to $30 \times$)

Layer Normalization

Step 1: Normalize each layer's input using its mean and its variance over *the features* (instead of over *the inputs*):

$$\bar{z}_n^{(l)} = \frac{z_n^{(l)} - \mu_n^{(l)} \cdot \mathbf{1}_K}{\sqrt{(\sigma_n^{(l)})^2 + \varepsilon}}$$

where $\mu_n^{(l)} = \frac{1}{K} \sum_{k=1}^K z_n^{(l)}(k)$ and $(\sigma_n^{(l)})^2 = \frac{1}{K} \sum_{k=1}^K (z_n^{(l)}(k) - \mu_n^{(l)})^2$, and $\varepsilon \in \mathbb{R}_{\geq 0}$

Step 2: Introduce learnable parameters $\gamma^{(l)}, \beta^{(l)} \in \mathbb{R}^K$:

$$\hat{z}_n^{(l)} = \gamma^{(l)} \odot \bar{z}_n^{(l)} + \beta^{(l)}$$

Remarks:

- Normalize across features, independently for each observation
- Very common alternative, widely used for transformers and text data
- No batch dependency, use the same for training and inference

Normalization - conclusion

Benefits of normalization layers:

- Stabilizes activation magnitudes / reduces initialization impact
- Stabilizes and speeds up training, allows larger learning rates
- Additional regularization effect from noisy batch statistics $\mu_B^{(l)}, \sigma_B^{(l)}$

Used in almost all modern deep learning architectures

- Often inserted after every convolutional layer, before non-linearity

Recap

- Neural networks are trained with gradient-based methods such as **SGD**
- To compute the gradients, we use **backpropagation**, which involves the chain rule to efficiently calculate the gradients based on the network's intermediate outputs $z^{(l)}$ and $\delta^{(l)}$
- Proper **parameter initialization** should avoid exploding and vanishing gradients by carefully controlling the layerwise variance
- **Batch and Layer normalization** dynamically stabilize the training process, allowing for faster convergence and the use of larger learning rates