When AI meets life sciences
当人工智能遇上生命科学

2025 清华大学 AI创新实践课程

张强锋，清华大学

Sep 29, 2025

Course outline

- Foundation Module: Fundamentals of Artificial Intelligence and Life Sciences
  - Artificial intelligence introduction (week 1-4, team up)
  - Review major fields in life sciences (week 5)
  - Intersection logic of AI and biological (week 6)

- Application Module: The AI-Driven Revolution in Life Sciences
  - Gene and Genome Scale (week 7-9, midterm)
  - Structural Biology and Drug Design (week 10-12)
  - Cellular and Medical Applications (week 13-14)

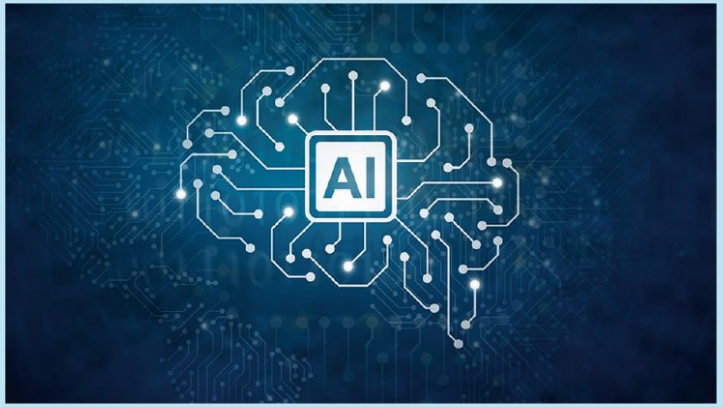
- Practical Module: From Data to Solutions (week 15-16, final)

Outline

- **Deep learning vs shallow learning**
- **Neural networks**
- **Basic concepts of deep learning training**
  - ➢ Forward propagation
  - ➢ Backward propagation
  - ➢ Gradient descent algorithm and parameters update
- **Overfitting and regularization**
- **Practice**

Revolution of AI

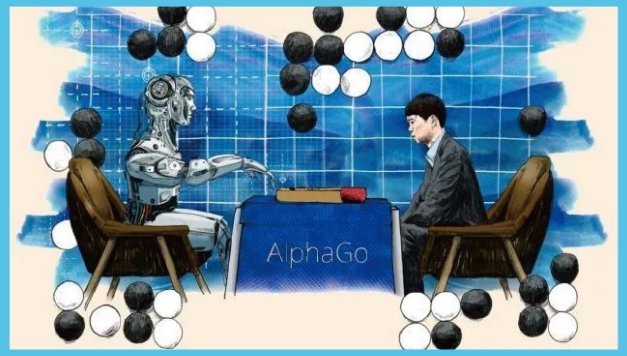
Artificial Intelligence



Machine Learning

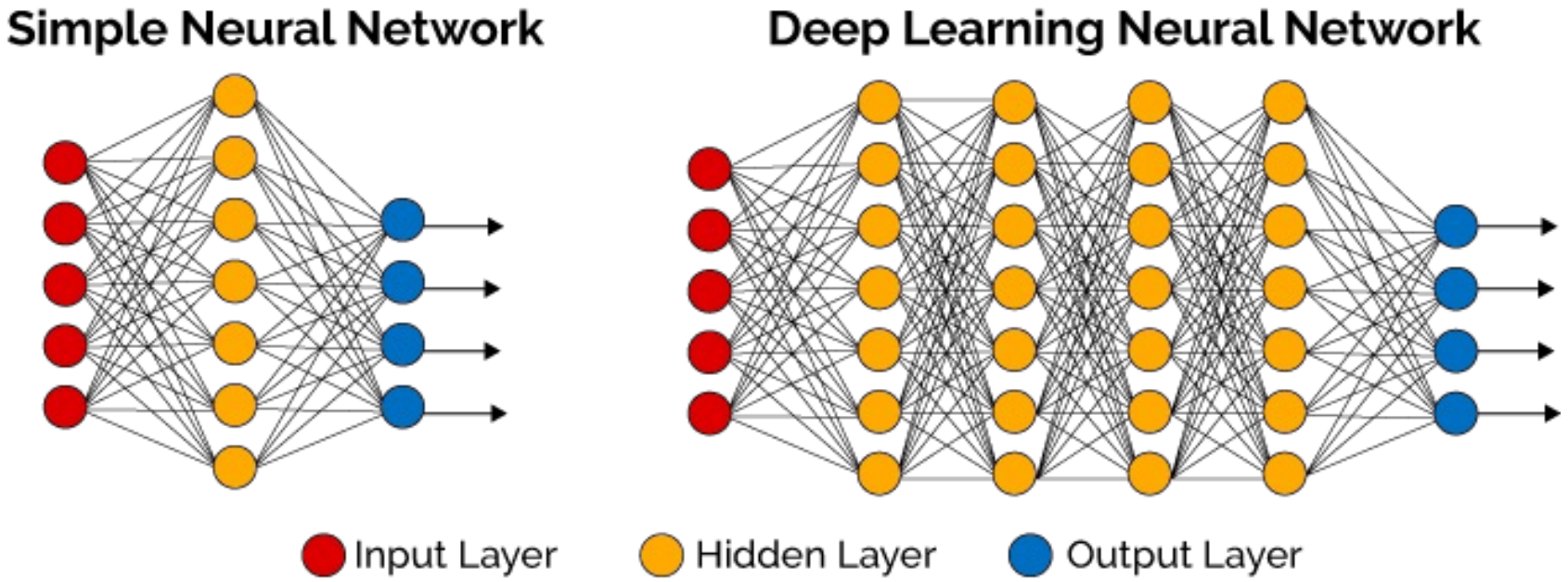


Deep Learning



1950's 1960's 1970's 1980's 1990's 2000's 2010's 2020's

Deep learning – complex models

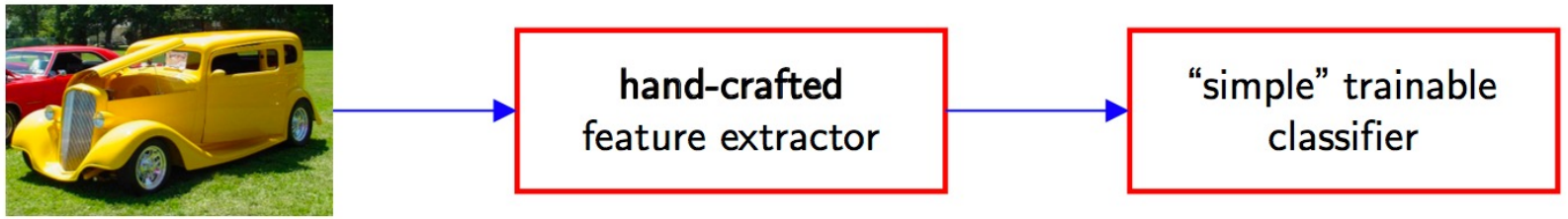


Universality: For any arbitrary function f(x), there exists a neural network that closely approximate it for any input x

Deep learning VS. shallow learning

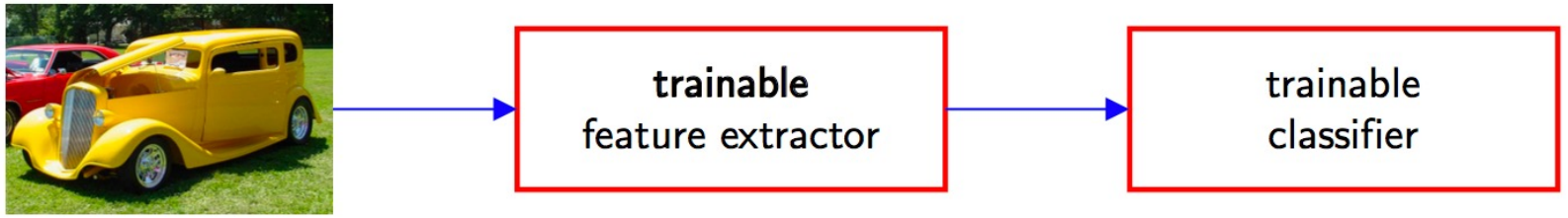
traditional model of pattern recognition (since the late 50's)

- ▷ fixed/engineered features (or kernel) + trainable classifier



end-to-end learning/feature learning/deep learning

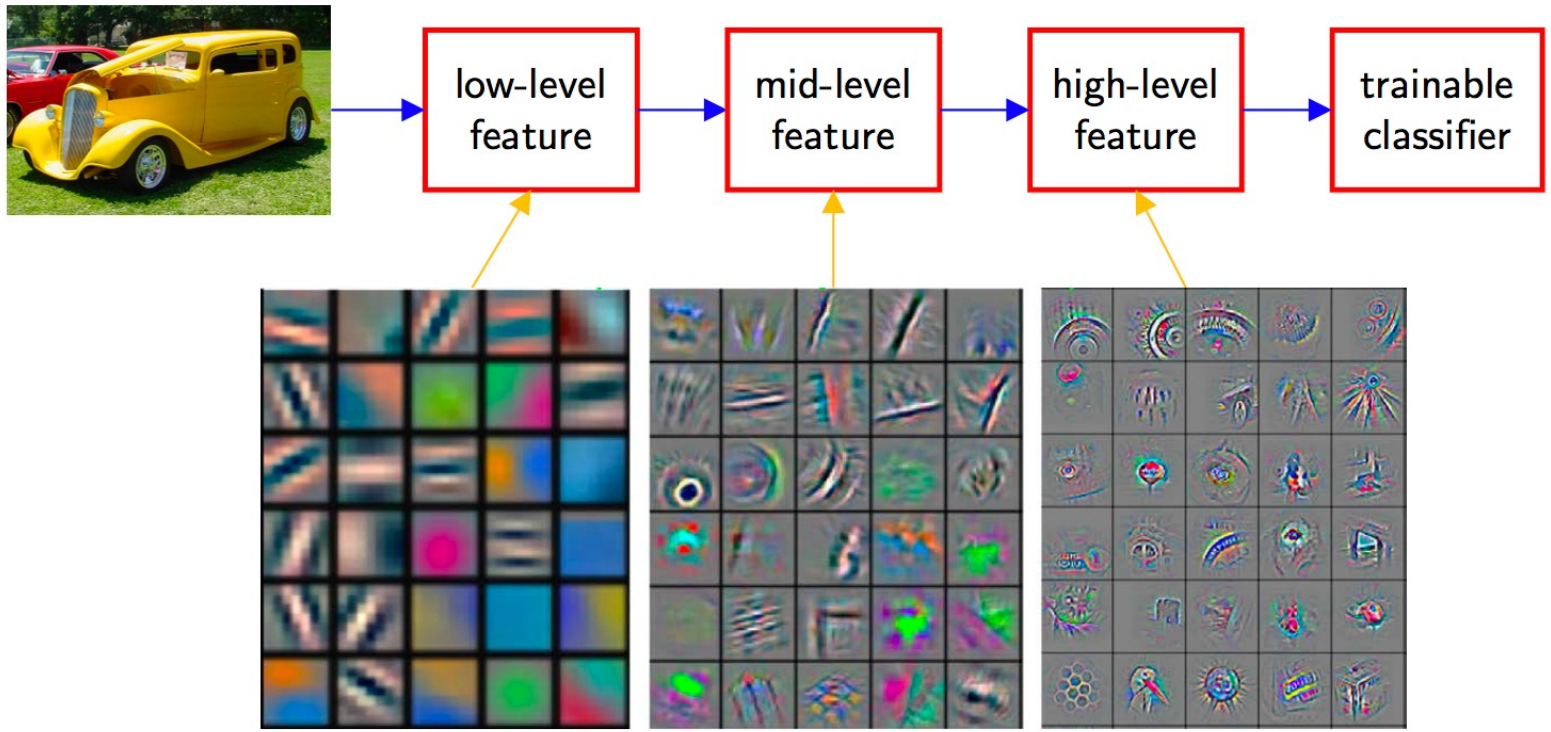
- ▷ trainable features (or kernel) + trainable classifier



Deep learning VS. shallow learning

DL = learning hierarchical representations

- ▷ deep if more than one stage of nonlinear feature transform

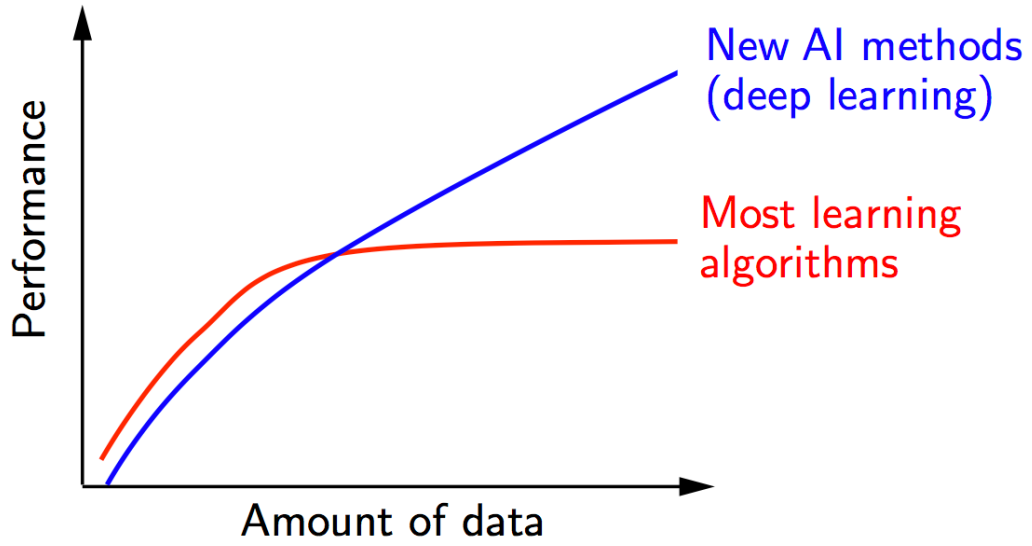


feature visualization of CNN trained on ImageNet (Zeiler and Fergus 2013)

It is time for deep learning

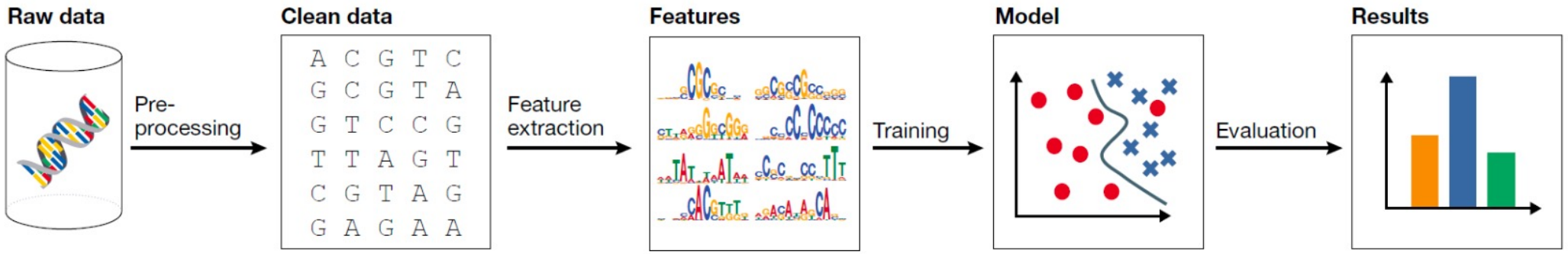
three key driving forces

1. big data
2. power of parallel/distributed computing
3. sophisticated algorithms



source: Andrew Ng

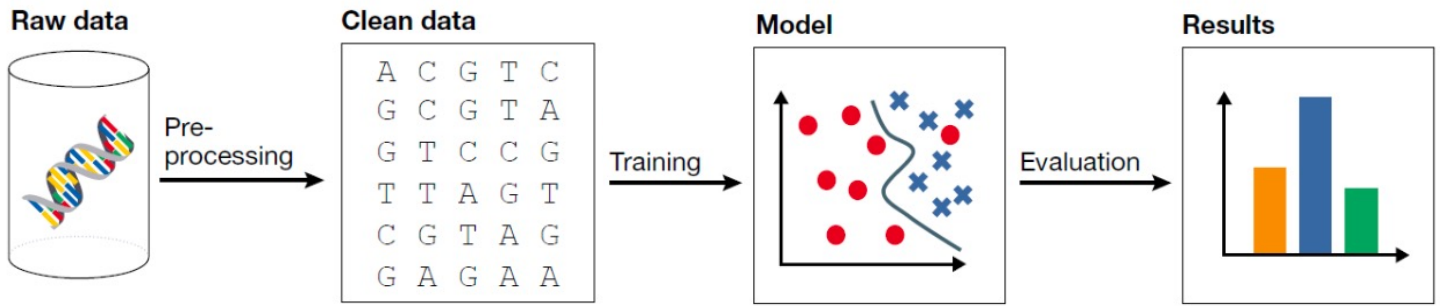
Classical machine learning



source: [Angermueller et al., 2016]

- Preprocessing
- feature engineering
- model selection & learning
- evaluation

Deep learning framework

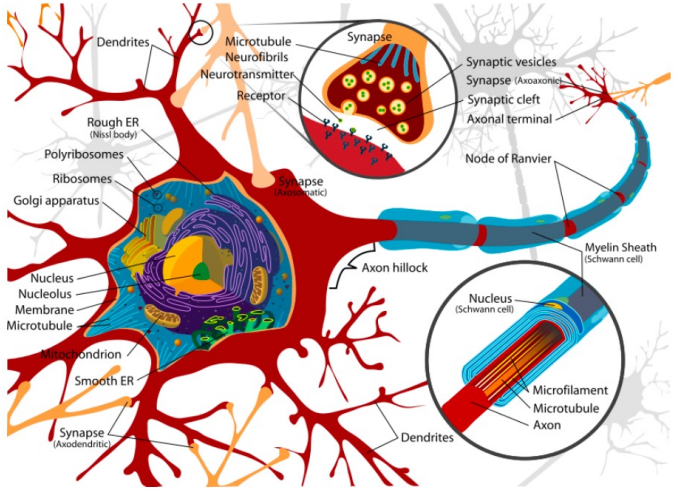


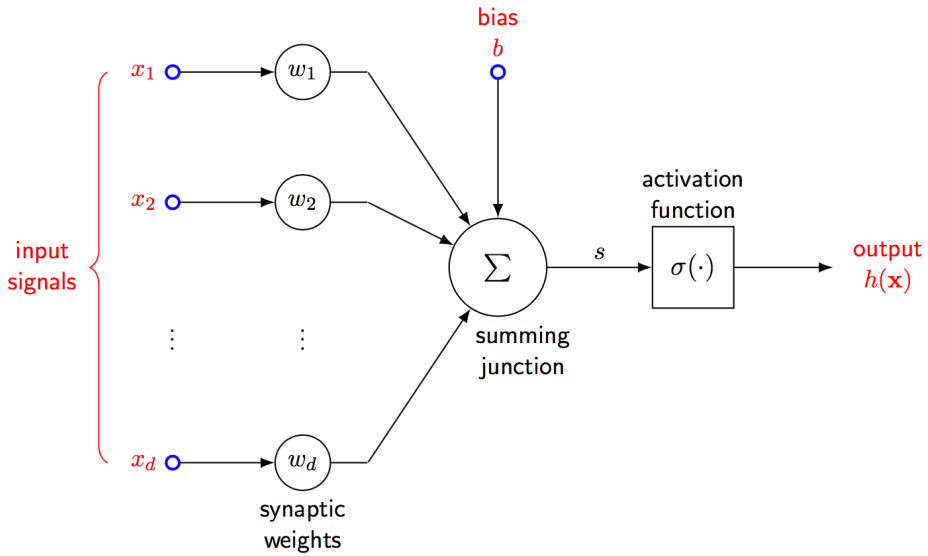
source: [Angermueller et al., 2016]

- Preprocessing
- feature engineering
- model selection & learning
- evaluation

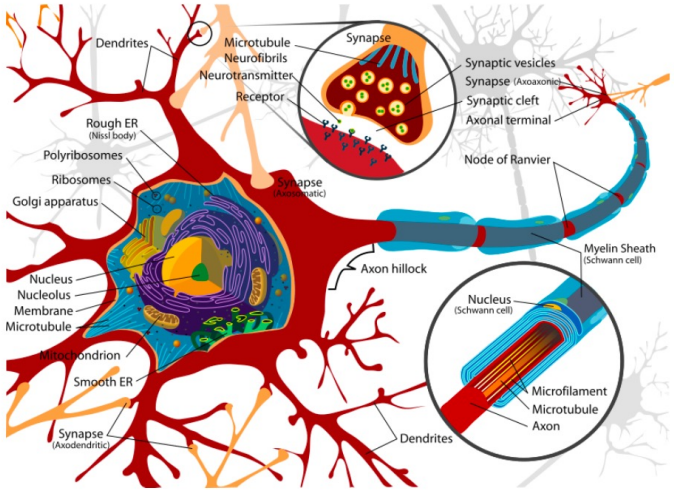
The concept of a neuron

- ⋆ electrically excitable cell that processes and transmits information through electrical and chemical signals

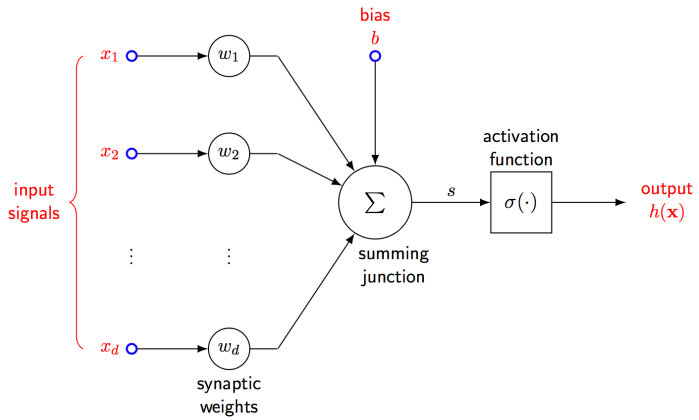


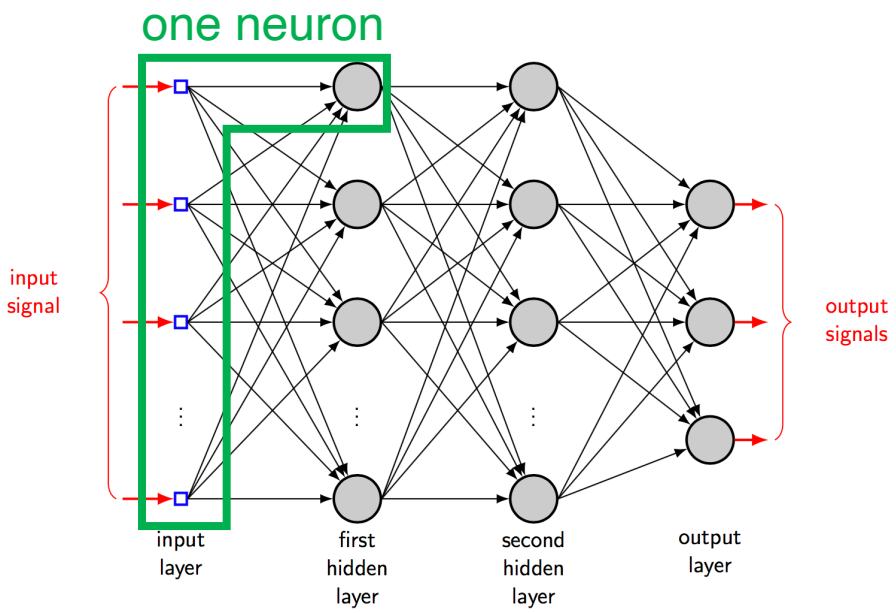


Networks of neurons: layered architecture

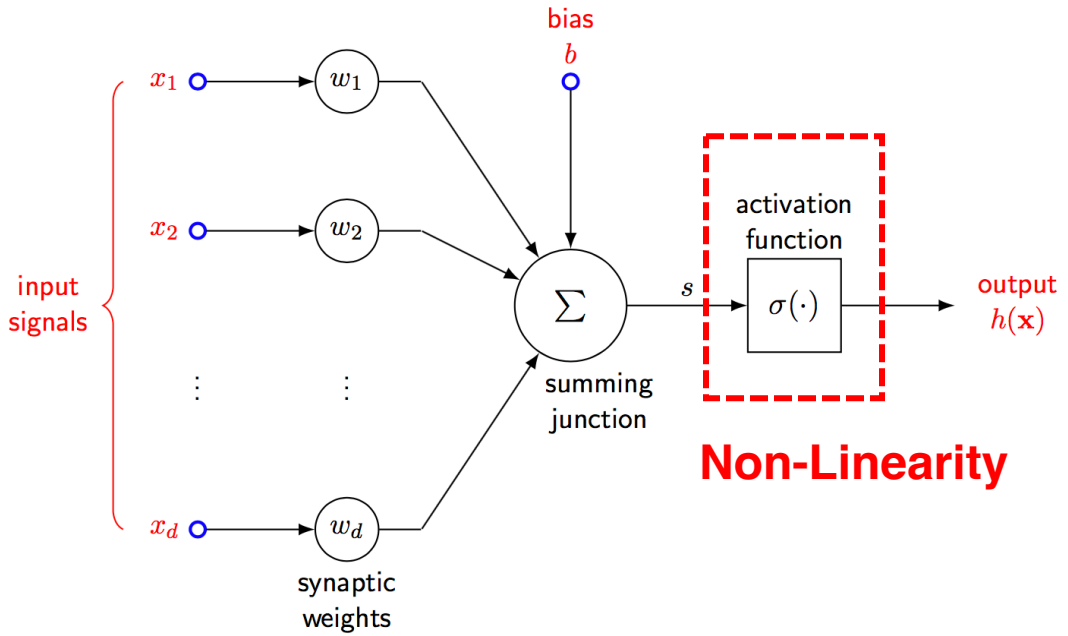








Activation function



$h(x) = \sigma(\boldsymbol{X}^l \boldsymbol{W}^l + \boldsymbol{b})$, l represents the layer

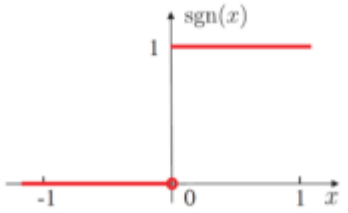
The characteristics of activation functions (σ):

- Non-linear
- Differentiable

Activation functions

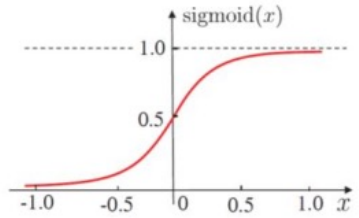
Differentiable

$$\sigma(z) = \mathrm{sgn}(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$



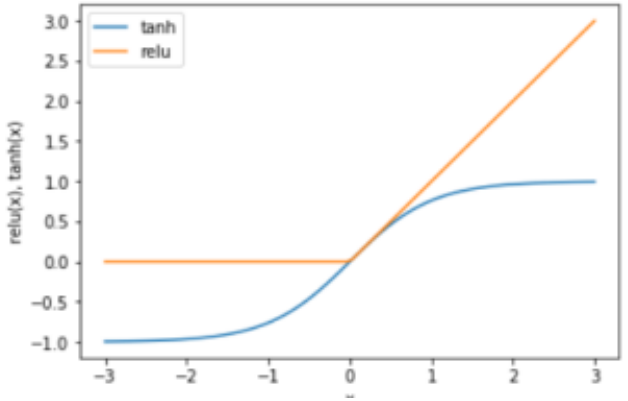
$$\sigma(z) = Sigmoid(z) = \frac{1}{1+e^{-z}}$$

frequently used in classification models



$$\frac{0-(-e^{-x})}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \times \left(1 - \frac{1}{1+e^{-x}}\right)$$

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



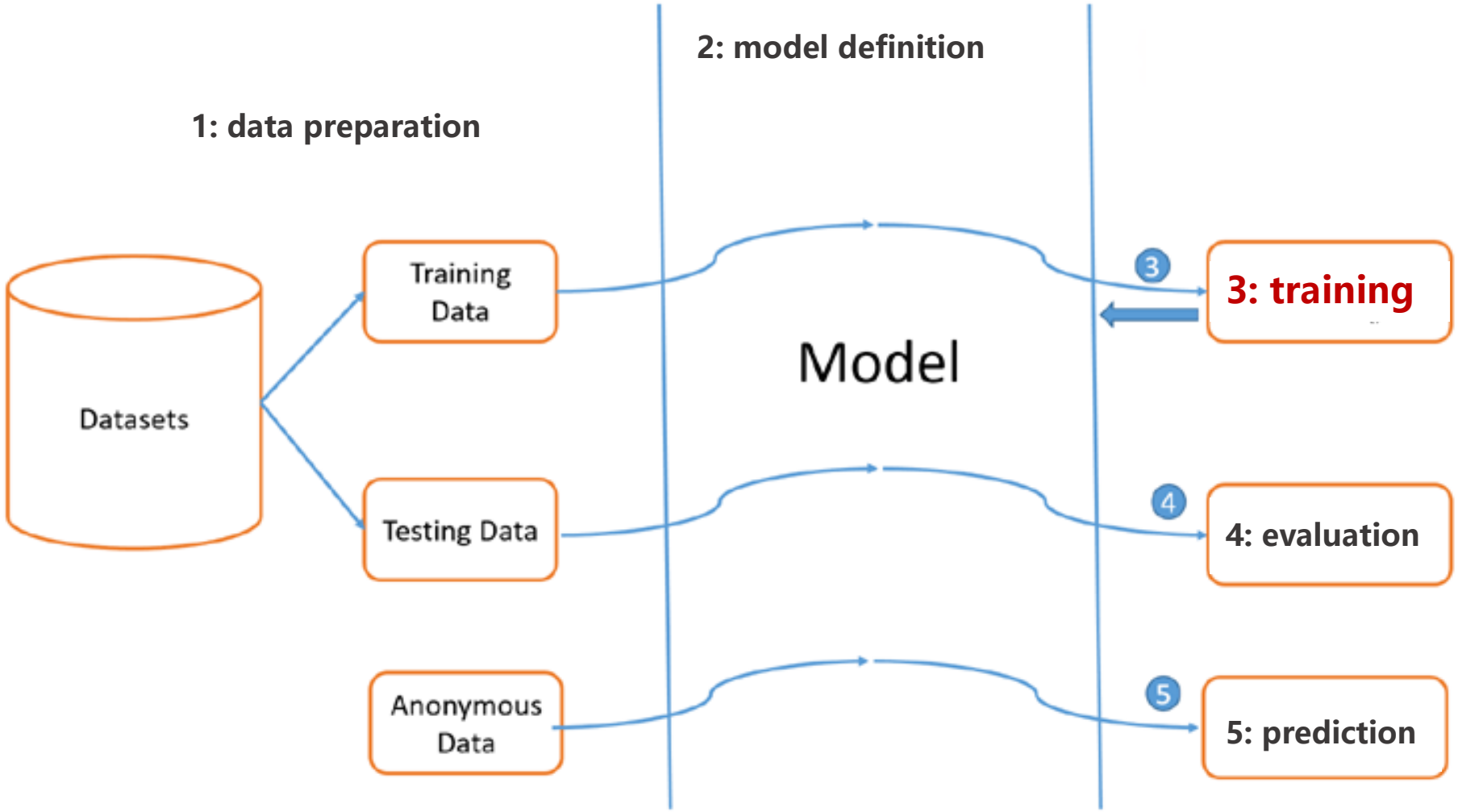
$$\frac{(e^{x}+e^{-x})^2-(e^{x}-e^{-x})^2}{(e^{x}+e^{-x})^2}$$

$$\sigma(z) = Relu(z) = \max(0, z)$$

$$\begin{cases} 0, x <= 0 \\ 1, x > 0 \end{cases}$$

frequently used in regression models and hidden layers

Developing a deep learning model



Break it down

- **Phase 1: data preparation**
  - Step 1: data preparation: data collection & cleaning, data preprocessing (standardization/normalization, encoding), data splitting (training set/validation set/test set)
- **Phase 2: model definition**
  - Step 2: model definition: choose an architecture, define layers, activation functions
  - Step 3: model configuration (hyperparameters): loss functions, optimizer, learning rate, metrics
- **Phase 3: model training**
- **Phase 4: model evaluation**
- **Phase 5: model application**

```
graph TD
    subgraph "Phase 1: Preparation"
        A[1. Prepare Data: Clean/Preprocess/Split] --> B(2.
Define Model: Design Architecture)
        B --> C(3. Compile Model: Set Loss/Optimizer/Metrics)
        C --> D(4. Initialize Parameters)
    end

    subgraph "Phase 2: Core Training Loop (Repeated for N
Epochs)"
        D --> E{Start an Epoch}
        E --> F[Get a Batch of Data]
        F --> G[5. Forward Propagation --> Prediction Ŷ]
        G --> H[6. Calculate Loss: Loss(Ŷ, Y)]
        H --> I[7. Backward Propagation --> Calculate Gradients]
        I --> J[8. Update Parameters via Optimizer]
        J --> K{All Batches Processed?}
        K -- No --> F
        K -- Yes --> L[Epoch Complete]
    end

    subgraph "Phase 3: Evaluation & Deployment"
        L --> M[9. Evaluate on Validation Set]
        M --> N{Overfitting or Underfitting Detected?}
        N -- Yes, Tune Hyperparameters --> C
        N -- No --> O{All Epochs Complete?}
        O -- No --> E
        O -- Yes --> P[11. Final Test on Test Set]
        P --> Q[12. Save and Deploy Model for Inference]
    end
```

Break it down

- Phase 1: data preparation
- Phase 2: model definition
- **Phase 3: model training**
  - Step 4: parameter Initialization: Break Symmetry, Xavier/He initialization, Pre-trained Weights
  - Step 5: forward propagation
  - Step 6: calculate loss
  - Step 7: backward propagation
  - Step 8: update parameters
- Phase 4: model evaluation
- Phase 5: model application

```
graph TD
    subgraph "Phase 1: Preparation"
        A[1. Prepare Data: Clean/Preprocess/Split] --> B(2.
Define Model: Design Architecture)
        B --> C(3. Compile Model: Set Loss/Optimizer/Metrics)
        C --> D(4. Initialize Parameters)
    end

    subgraph "Phase 2: Core Training Loop (Repeated for N
Epochs)"
        D --> E{Start an Epoch}
        E --> F[Get a Batch of Data]
        F --> G[5. Forward Propagation --> Prediction Ŷ]
        G --> H[6. Calculate Loss: Loss(Ŷ, Y)]
        H --> I[7. Backward Propagation --> Calculate Gradients]
        I --> J[8. Update Parameters via Optimizer]
        J --> K{All Batches Processed?}
        K -- No --> F
        K -- Yes --> L[Epoch Complete]
    end

    subgraph "Phase 3: Evaluation & Deployment"
        L --> M[9. Evaluate on Validation Set]
        M --> N{Overfitting or Underfitting Detected?}
        N -- Yes, Tune Hyperparameters --> C
        N -- No --> O{All Epochs Complete?}
        O -- No --> E
        O -- Yes --> P[11. Final Test on Test Set]
        P --> Q[12. Save and Deploy Model for Inference]
    end
```

Break it down

- **Phase 1: data preparation**
- **Phase 2: model definition**
- **Phase 3: model training**
- **Phase 4: model evaluation**
  - Step 9: epoc validation: validation set (underfitting and overfitting)
  - Step 10: model tuning and final selection (reset hyperparameters)
- **Phase 5: model application**
  - Step 11: final testing
  - Step 12: deployment and inference

```
graph TD
    subgraph "Phase 1: Preparation"
        A[1. Prepare Data: Clean/Preprocess/Split] --> B(2.
Define Model: Design Architecture)
        B --> C(3. Compile Model: Set Loss/Optimizer/Metrics)
        C --> D(4. Initialize Parameters)
    end

    subgraph "Phase 2: Core Training Loop (Repeated for N
Epochs)"
        D --> E{Start an Epoch}
        E --> F[Get a Batch of Data]
        F --> G[5. Forward Propagation --> Prediction Ŷ]
        G --> H[6. Calculate Loss: Loss(Ŷ, Y)]
        H --> I[7. Backward Propagation --> Calculate Gradients]
        I --> J[8. Update Parameters via Optimizer]
        J --> K{All Batches Processed?}
        K -- No --> F
        K -- Yes --> L[Epoch Complete]
    end

    subgraph "Phase 3: Evaluation & Deployment"
        L --> M[9. Evaluate on Validation Set]
        M --> N{Overfitting or Underfitting Detected?}
        N -- Yes, Tune Hyperparameters --> C
        N -- No --> O{All Epochs Complete?}
        O -- No --> E
        O -- Yes --> P[11. Final Test on Test Set]
        P --> Q[12. Save and Deploy Model for Inference]
    end
```

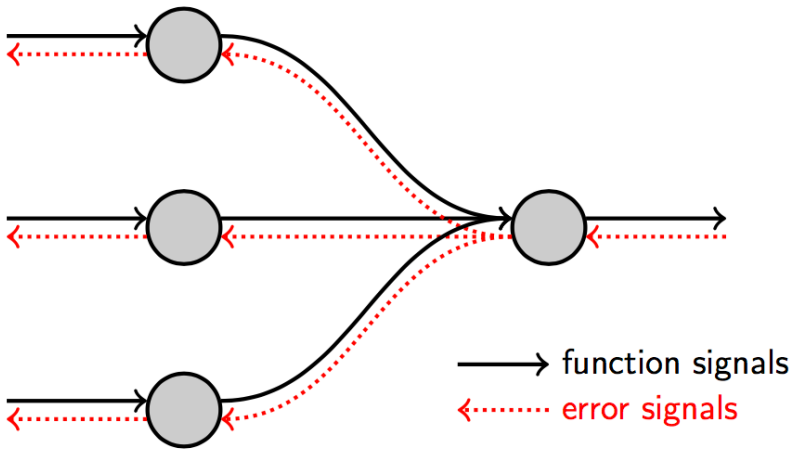
Training of a network

function signals: forward propagation

- input signal comes in at input
- propagates forward through network, and
- emerges at output

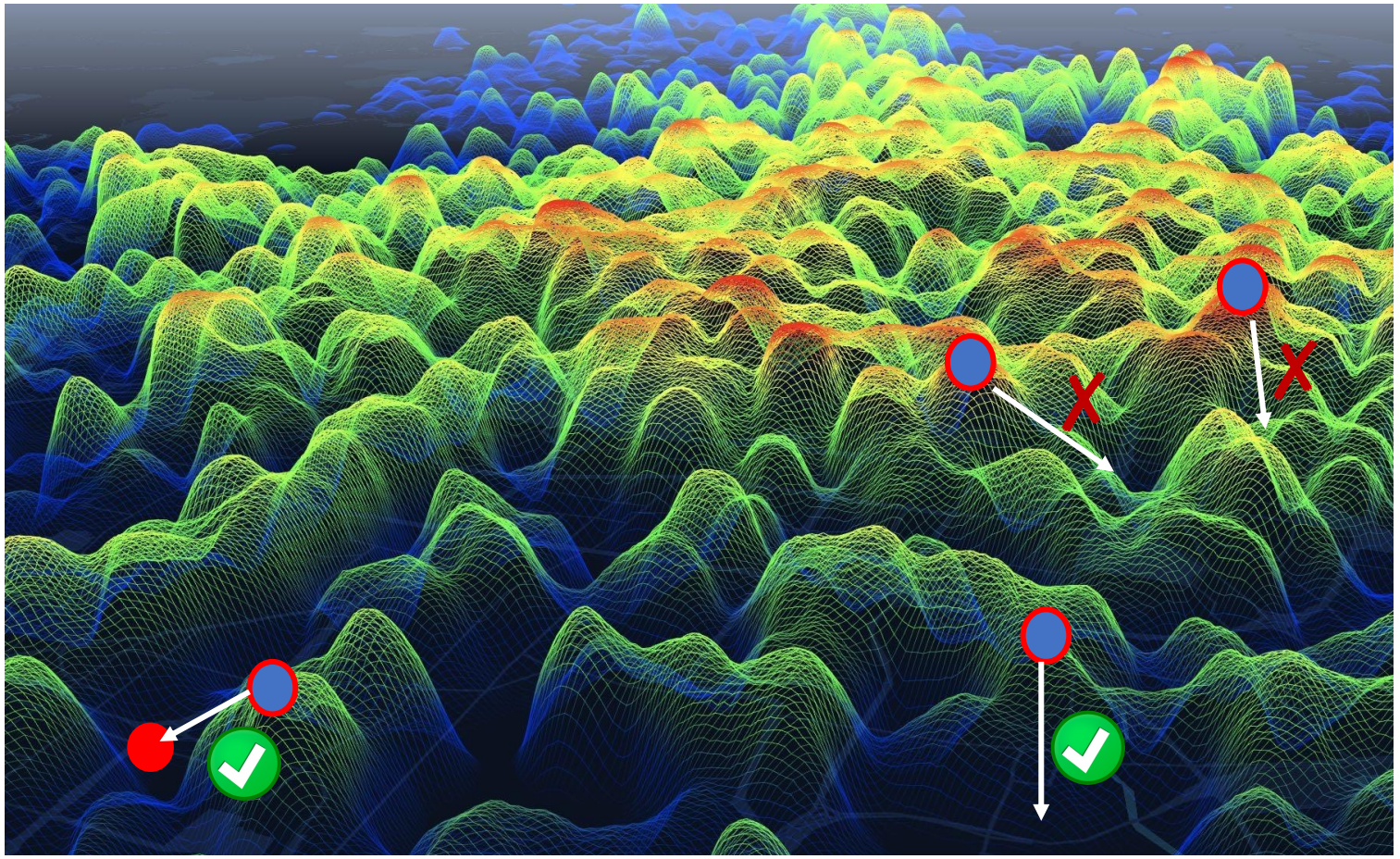
error signals: back(ward) propagation

- originates at an output neuron, and
- propagates backward



source: [Haykin, 2009]

Weight initialization



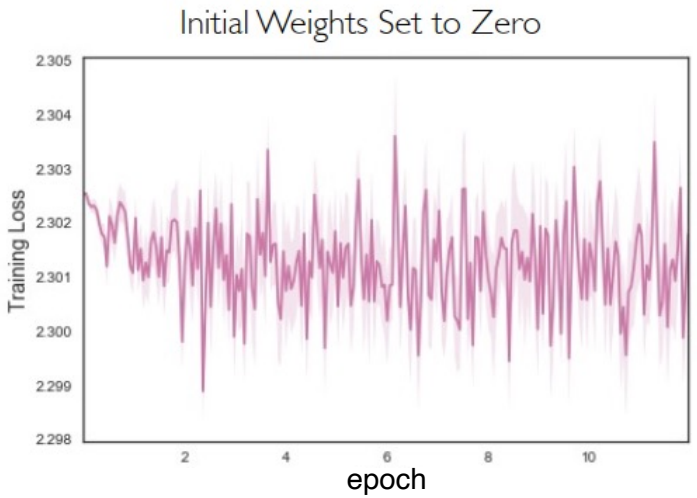
Local minimum

Global minimum

The initialization point determines the convergence speed and influences the final outcome.

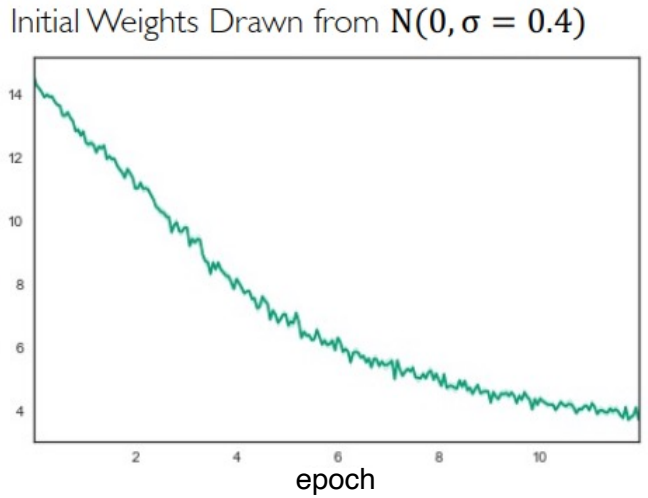
Weight initialization

Zero Initialization



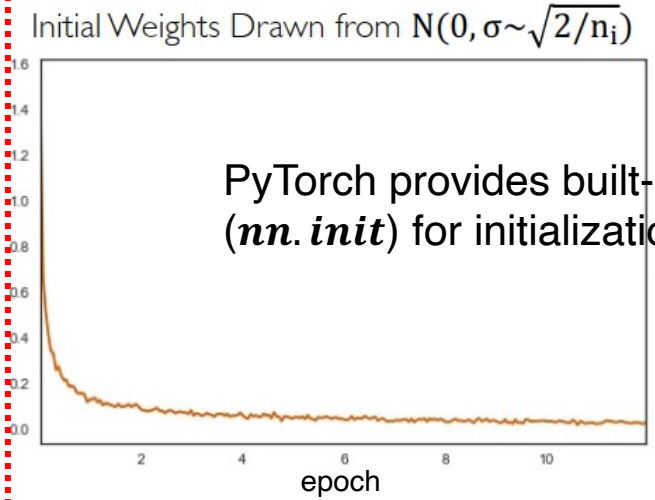
- The loss is oscillating
- The network cannot learn at all

Random Initialization



- Rate of convergence is very low
- The network barely achieves a validation accuracy of about 88%

He Initialization

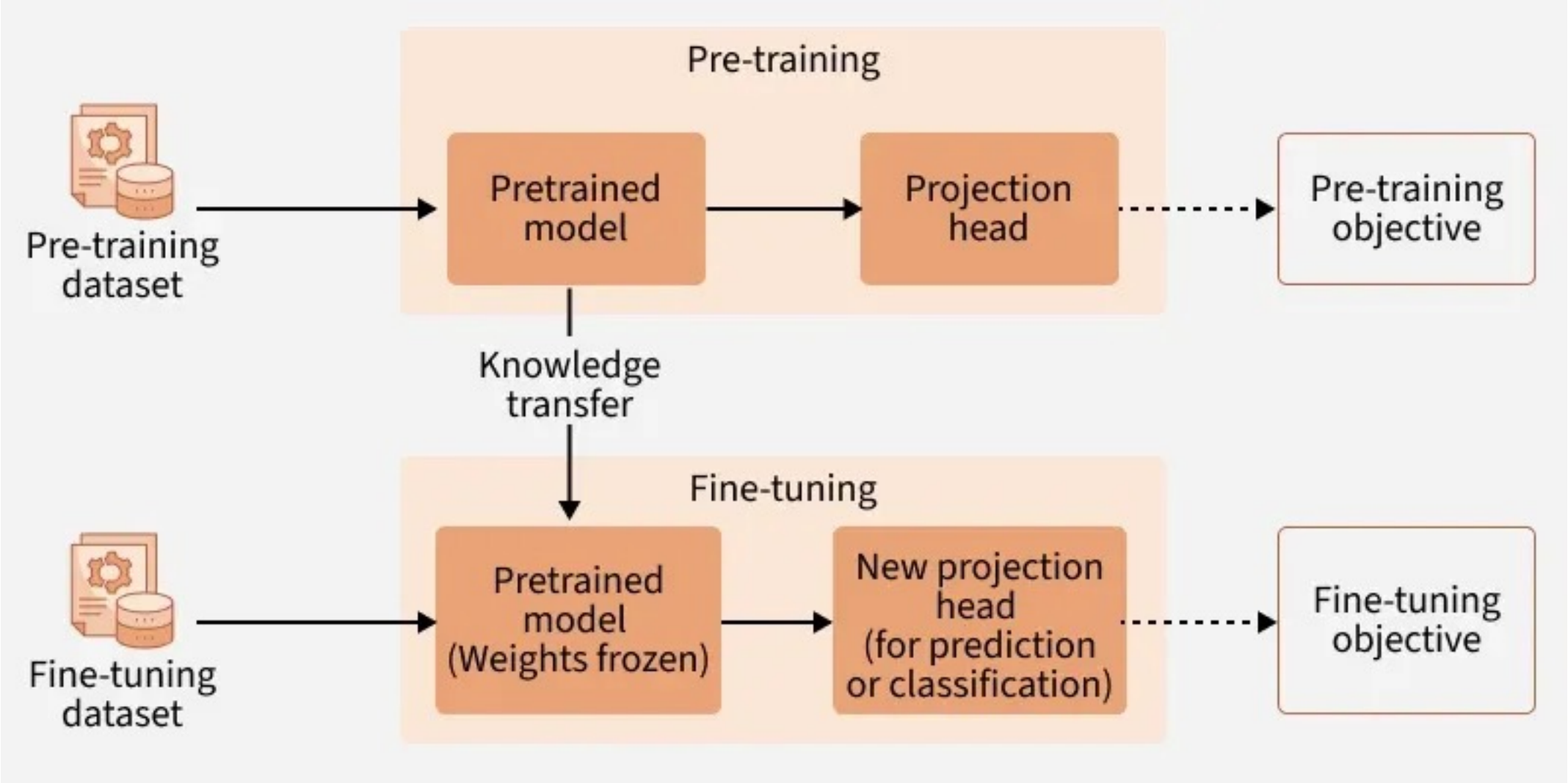


- The loss is smaller two orders of magnitude than others
- A validation accuracy greater than 99%

PyTorch provides built-in methods ($nn.init$) for initialization.

Weight initialization

Large language models/ Foundation models!



Forward propagation

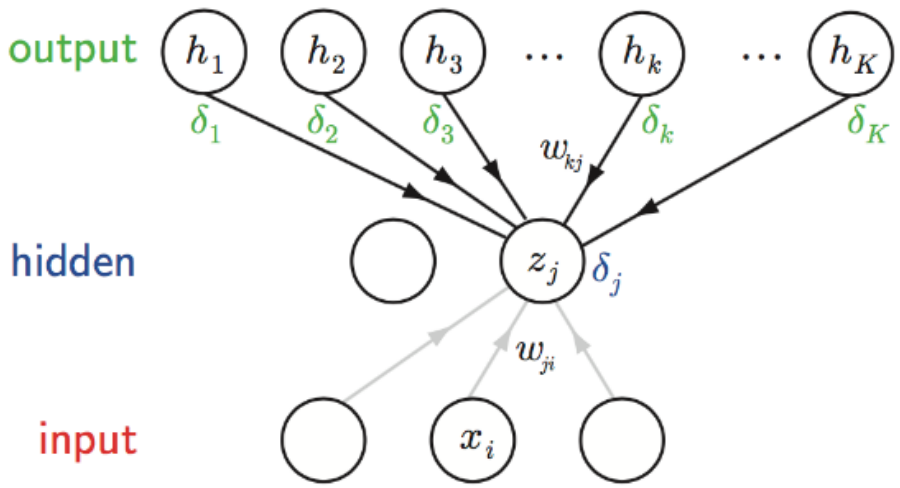
⋆ forward propagation: send input $\mathbf{x}$ via hidden $\mathbf{z}$ to output $\mathbf{h}$

- activation function:

$$\sigma(s) = 1/(1+e^{-s})$$

- hidden layer:

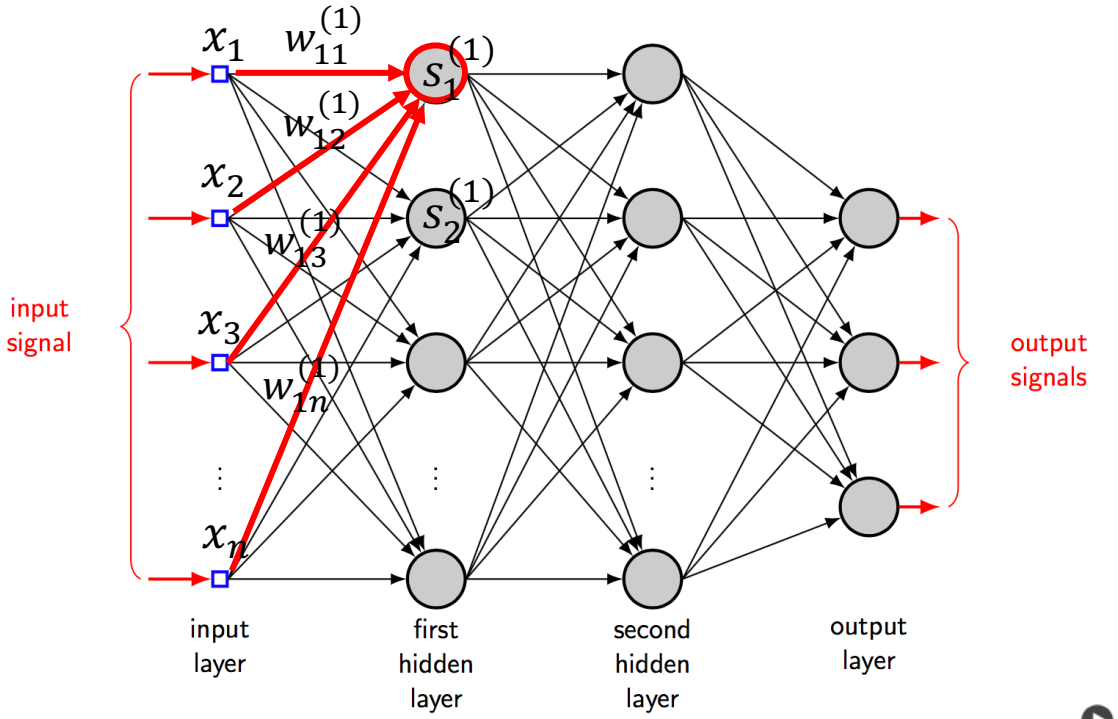
$$z_j(\mathbf{x}) = \sigma\left(\sum_i w_{ji} x_i\right)$$



- output layer:

$$h_k(\mathbf{x}) = \sigma\left(\sum_j w_{kj} z_j\right) = \sigma\left(\sum_j w_{kj}\ \sigma\left(\sum_i w_{ji} x_i\right)\right)$$

Multi-layer forward propagation



How to calculate the 1st neuron of 1st hidden layer:

$$s_1^{(1)} = w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + \cdots + w_{1n}^{(1)}x_n + b_1^{(1)}$$

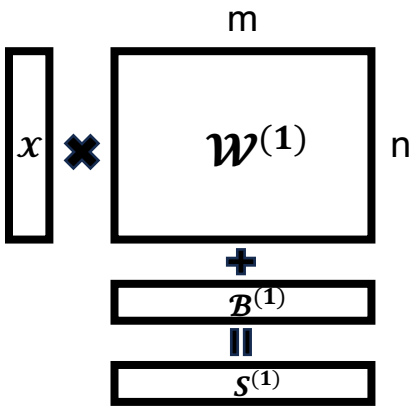
How to calculate all neurons of 1st hidden layer:

$$\boldsymbol{S}^{(1)} = \boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{B}^{(1)}$$

$$\boldsymbol{S}^{(1)} = \left(s_1^{(1)} s_2^{(1)} s_3^{(1)} \cdots s_m^{(1)}\right)$$

$$\boldsymbol{X} = (x_1\ x_2\ x_3 \cdots\ x_n) \quad \boldsymbol{B}^{(1)} = \left(b_1^{(1)} b_2^{(1)} b_3^{(1)} \cdots b_m^{(1)}\right)$$

$$\boldsymbol{W}^{(1)} = \begin{pmatrix} w_{11}^{(1)} & \cdots & w_{m1}^{(1)} \\ \vdots & \ddots & \vdots \\ w_{1n}^{(1)} & \cdots & w_{mn}^{(1)} \end{pmatrix}$$




```
import numpy as np

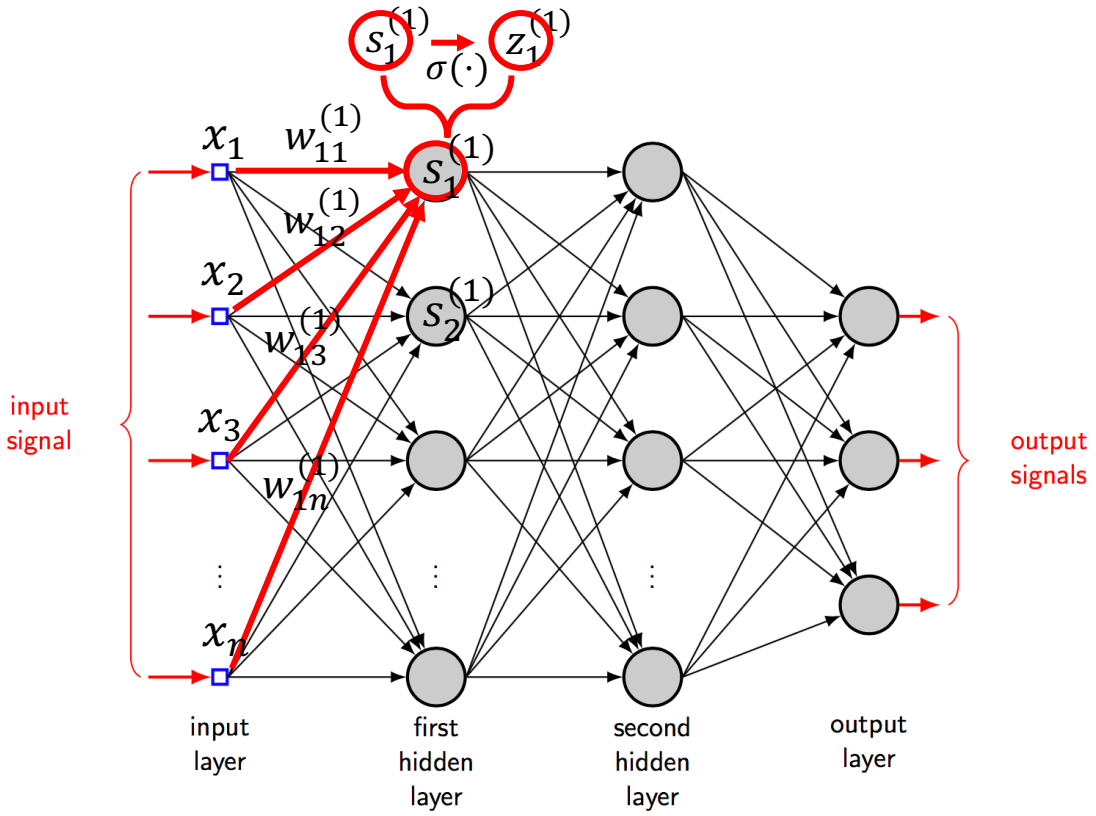
X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])
print(X.shape)
print(W1.shape)
print(B1.shape)
```

```
(2,)
(2, 3)
(3,)
```

```
S1 = np.dot(X, W1) + B1
print(S1.shape)
print(S1)
```

```
(3,)
[0.3 0.7 1.1]
```

Multi-layer forward propagation



$$\boldsymbol{Z}^{(1)} = \boldsymbol{\sigma}\left(\boldsymbol{S}^{(1)}\right) = \boldsymbol{\sigma}(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{B}^{(1)})$$

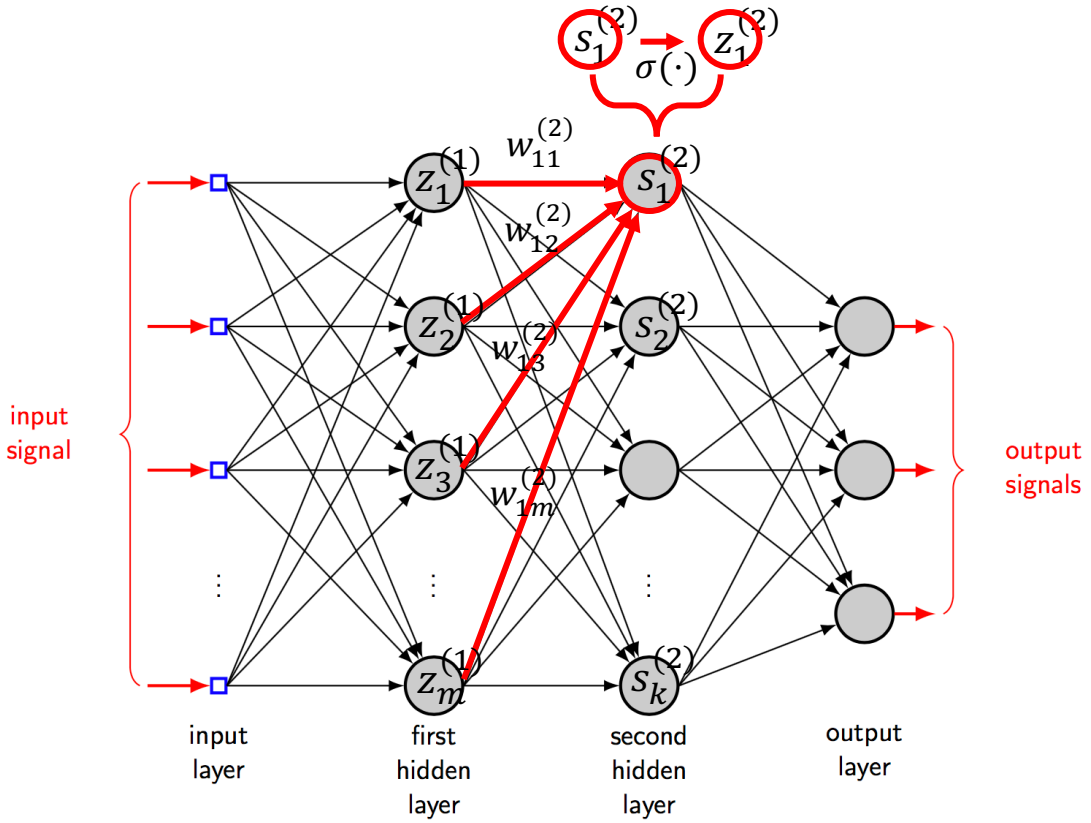
```
def  sigmoid(x):
    return  1/(1+np.exp(-x))

Z1  =  sigmoid(S1)

print(S1)
print(Z1)
```

```
[0.3  0.7  1.1]
[0.57444252  0.66818777  0.75026011]
```

Output of forward propagation



$$\boldsymbol{Z}^{(2)} = \boldsymbol{\sigma}\left(\boldsymbol{S}^{(2)}\right) = \boldsymbol{\sigma}(\boldsymbol{Z}^{(1)}\boldsymbol{W}^{(2)} + \boldsymbol{B}^{(2)})$$

```
W2 = np.array([[0.1, 0.3, 0.5, 0.7, 0.9],
               [0.2, 0.4, 0.6, 0.8, 1.0],
               [-0.1, 0.2, -0.3, 0.4, -0.5]])
B2 = np.array([0.1, 0.2, 0.3, 0.2, 0.1])

S2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(S2)

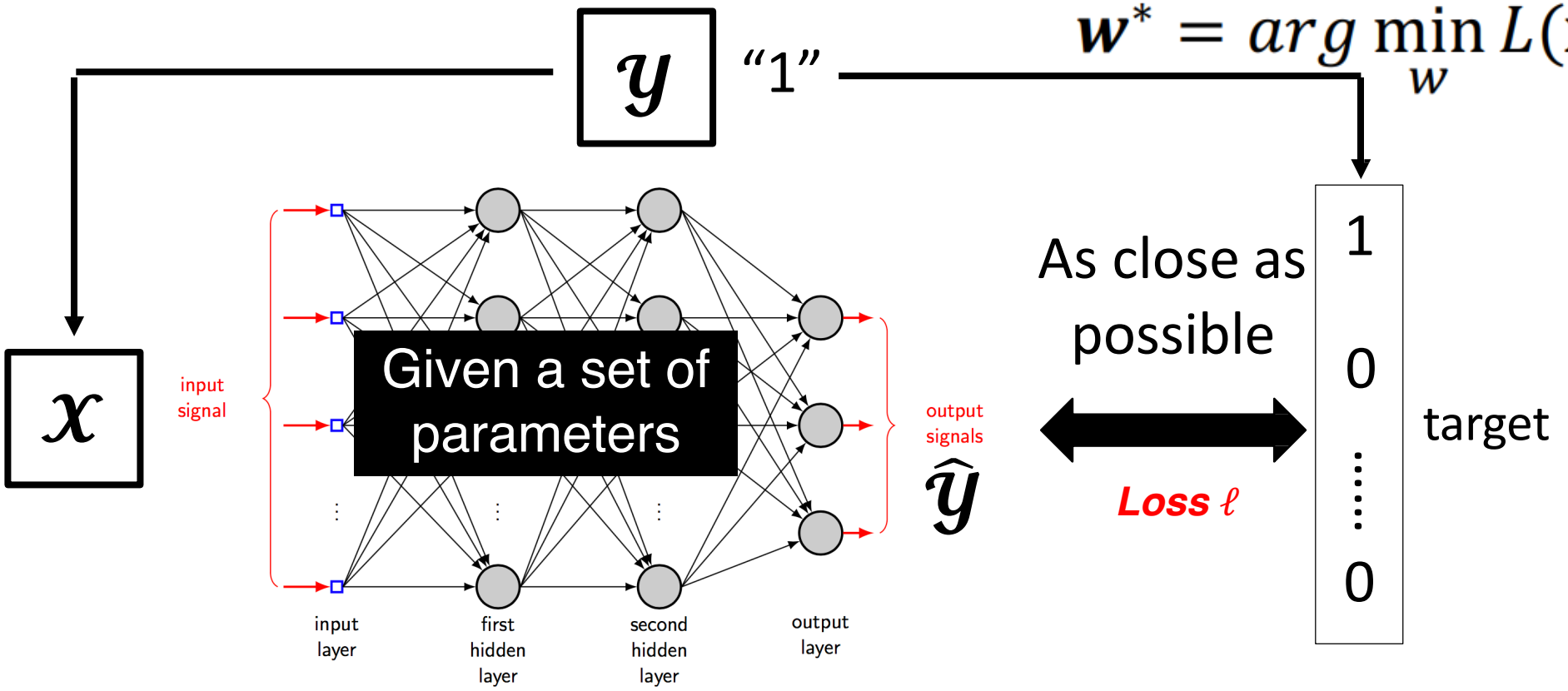
print(S2)
print(Z2)
```

```
[0.2160558  0.78965989 0.76305589 1.43676402 0.91005598]
[0.55380481 0.6877583  0.68201683 0.80795304 0.71301162]
```

Loss functions

A good function should make the loss of all examples as small as possible.

$$\boldsymbol{w}^* = arg \min_{w} L(\mathbf{x}, \boldsymbol{w})$$



Loss can be the distance between the network output ($\hat{y}$) and target (y), it should be as small as possible.

Loss functions

Mean Squared Error (MSE)

$$\varepsilon = \frac{1}{2}\sum_{k}(y_k - \hat{y}_k)^2$$

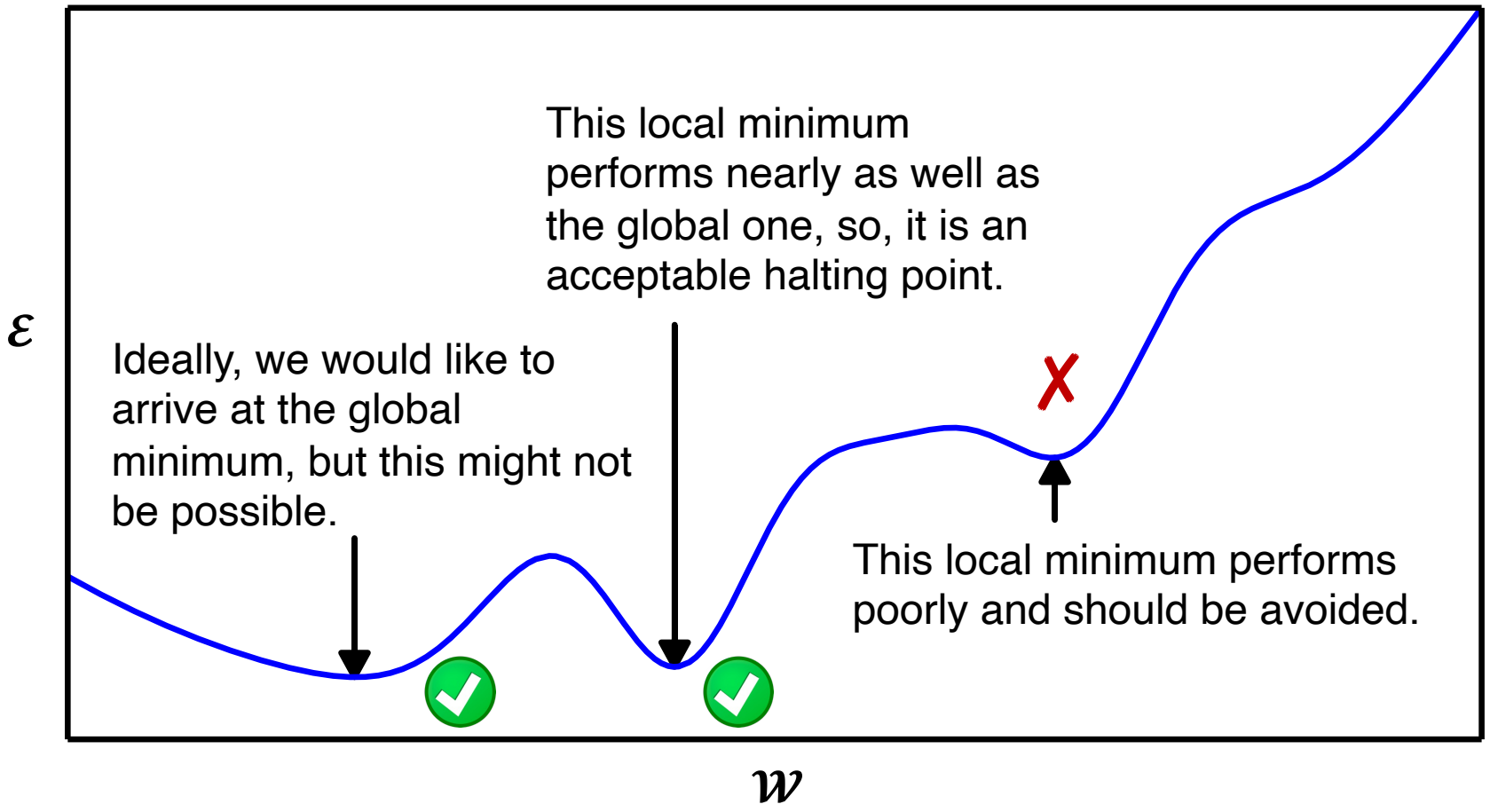
- Regression
- It's used when predicting a continuous value
- e.g., house prices, temperature, age.

Cross-Entropy Error

$$\varepsilon = -\sum_{k}\hat{y}_k \log y_k$$

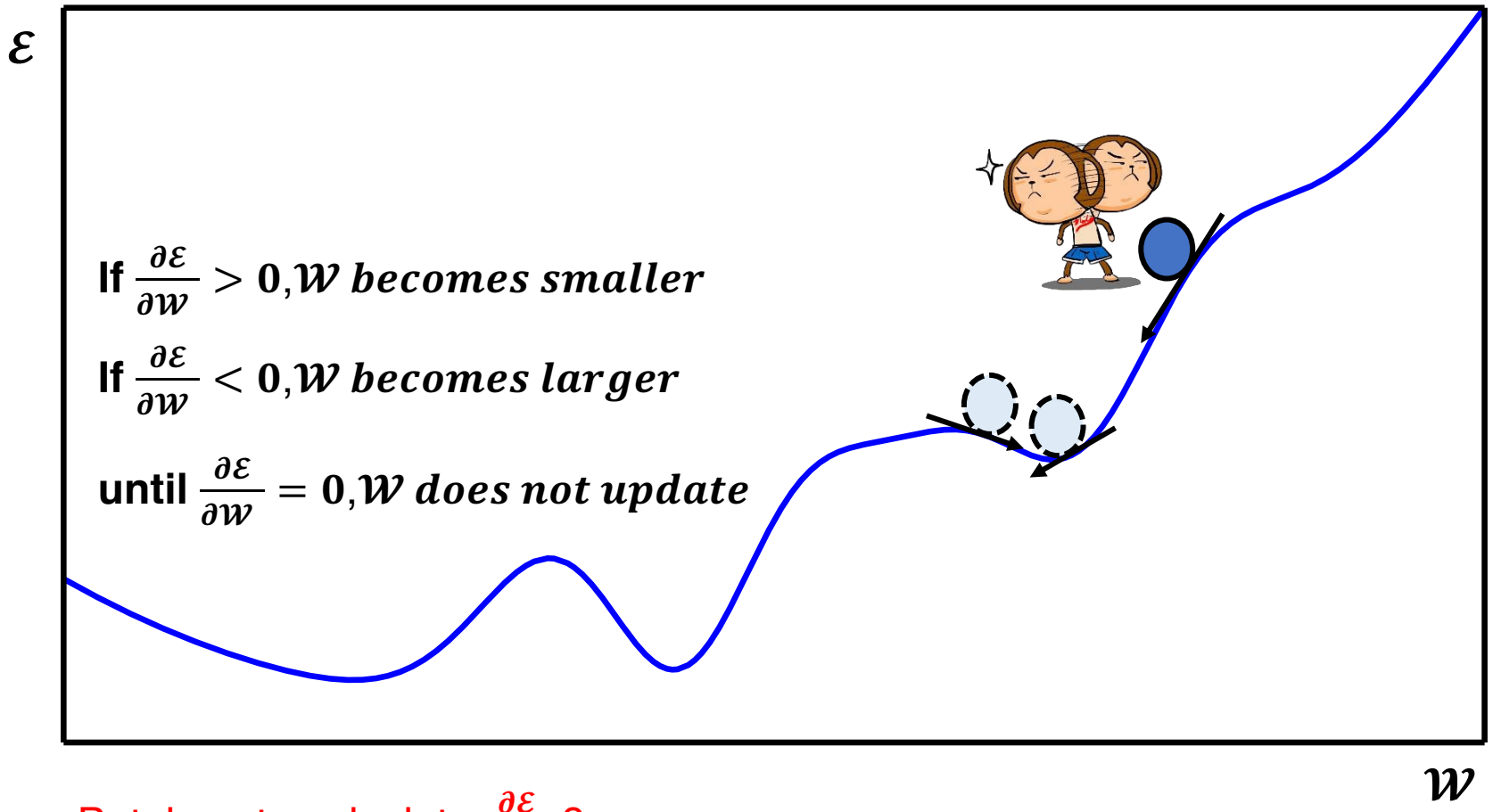
- Classification
- It's the standard loss function for models predicting class probabilities
- e.g., image recognition, spam detection.

How to find an optimal w to minimize the loss



Gradient descent

We aim to find the parameter $\boldsymbol{w}$ that minimizes the total loss $\boldsymbol{\varepsilon}$.



But, how to calculate $\frac{\partial \varepsilon}{\partial w}$?

$$\boldsymbol{w}_\text{new} = \boldsymbol{w}_\text{old} - \eta \frac{\partial \varepsilon}{\partial w}$$

Back propagation

★ back propagation: enables us to train hidden weights

- input-to-hidden weights w_{ji}

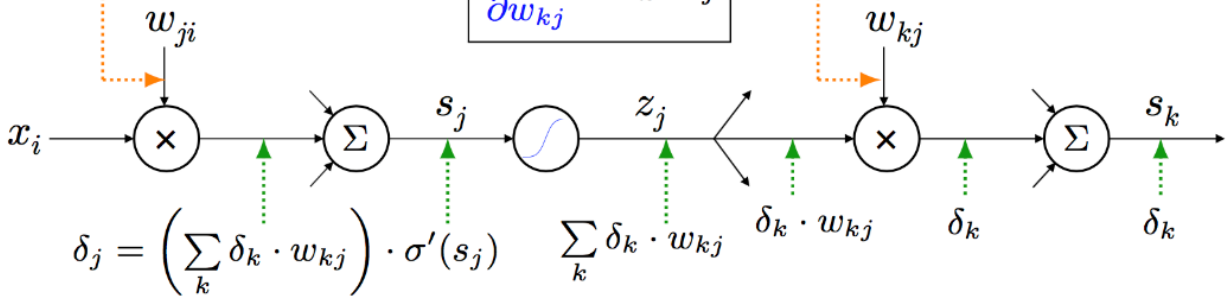
$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_{ji}} &= \delta_j \cdot x_i \\ \delta_j &= \sigma'(s_j) \cdot \sum_k w_{kj}\delta_k \\ \Delta w_{ji} &= -\eta \cdot \delta_j \cdot x_i\end{aligned}$$

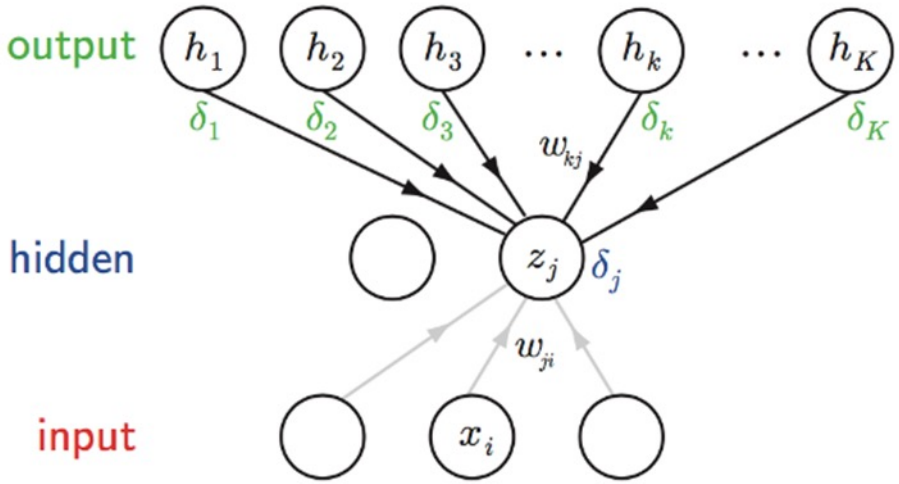
- hidden-to-output weights w_{kj}

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_{kj}} &= \delta_k \cdot z_j \\ \delta_k &= \sigma'(s_k) \cdot e_k \\ \Delta w_{kj} &= -\eta \cdot \delta_k \cdot z_j\end{aligned}$$

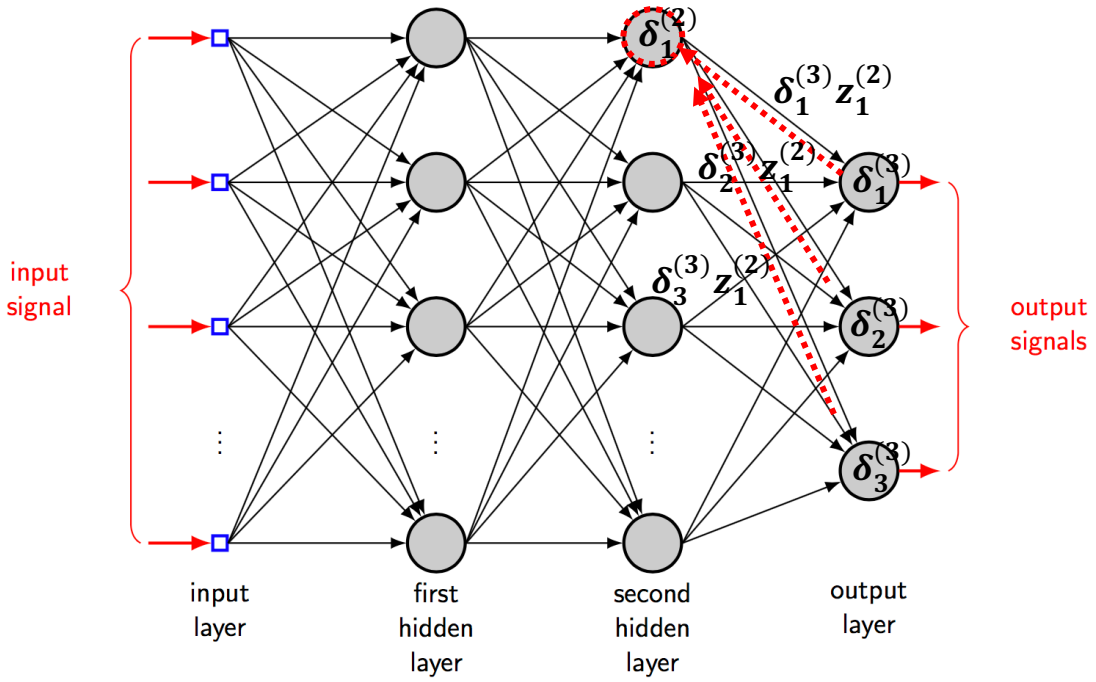
$$\left(\sum_k \delta_k \cdot w_{kj}\right) \cdot \sigma'(s_j) \cdot x_i \implies \boxed{\frac{\partial \mathcal{E}}{\partial w_{ji}} = \delta_j \cdot x_i = \left(\sum_k \delta_k \cdot w_{kj}\right) \cdot \sigma'(s_j) \cdot x_i}$$

$$\boxed{\frac{\partial \mathcal{E}}{\partial w_{kj}} = \delta_k \cdot z_j}$$





Back propagation



From the 2nd layer output $\boldsymbol{z}^{(2)}$ to the output layer $\boldsymbol{h}^{(3)}$:

$$\hat{y} = \boldsymbol{h}^{(3)} = \boldsymbol{z}^{(2)}\boldsymbol{w}^{(3)} + \boldsymbol{b}^{(3)}$$

The activate function of output layer is $\sigma(\cdot) = \cdot$

How to calculate the gradients of the output layer:

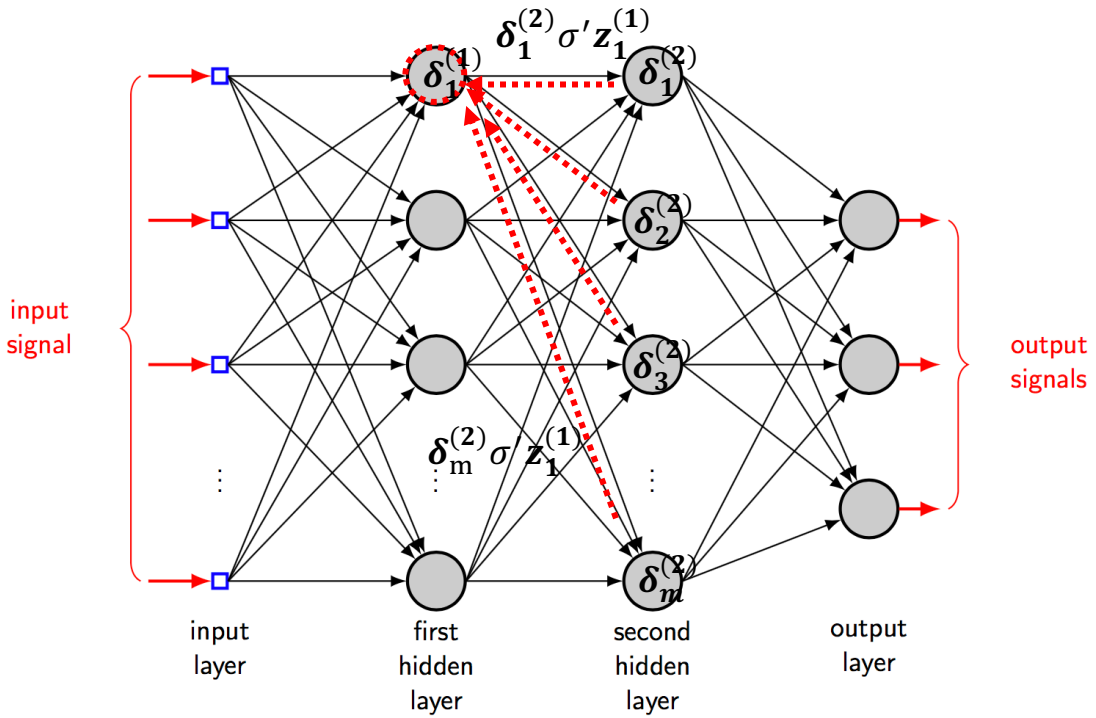
$$\delta_i^{(3)} = \frac{\partial \mathcal{E}}{\partial h_i^{(3)}} = |y_i - \hat{y}_i|$$

How to calculate the gradients of the 2nd layer:

$$\frac{\partial \mathcal{E}}{\partial w_{i1}^{(3)}} = \frac{\partial \mathcal{E}}{\partial h_i^{(3)}} \frac{\partial h_i^{(3)}}{\partial w_{i1}^{(3)}} = \delta_i^{(3)} z_1^{(2)}$$

$$\delta_1^{(2)} = \frac{\partial \mathcal{E}}{\partial z_1^{(2)}} = \sum_i \frac{\partial \mathcal{E}}{\partial h_i^{(3)}} \frac{\partial h_i^{(3)}}{\partial z_1^{(2)}} = \sum_i \delta_i^{(3)} w_{i1}^{(3)}$$

Back propagation



From the 1st layer output $\mathcal{Z}^{(1)}$ to the 2nd layer $\mathcal{Z}^{(2)}$:

$$\begin{aligned}\hat{y} &= \boldsymbol{h}^{(3)} \\ &= \mathcal{Z}^{(2)}\mathcal{W}^{(3)} + \mathcal{B}^{(3)} \\ &= \boldsymbol{\sigma}\left(\mathcal{S}^{(2)}\right)\mathcal{W}^{(3)} + \mathcal{B}^{(3)} \\ &= \boldsymbol{\sigma}\left(\mathcal{Z}^{(1)}\mathcal{W}^{(2)} + \mathcal{B}^{(2)}\right)\mathcal{W}^{(3)} + \mathcal{B}^{(3)}\end{aligned}$$

How to calculate the gradients of the 1st layer:

$$\frac{\partial \mathcal{E}}{\partial \mathcal{W}_{i1}^{(2)}} = \frac{\partial \mathcal{E}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial \mathcal{W}_{i1}^{(2)}} = \delta_i^{(2)} \sigma' z_1^{(1)}$$

$$= \sigma' z_1^{(1)} \sum_k \delta_k^{(3)} \mathcal{W}_{ki}^{(3)}$$

$$\delta_1^{(1)} = \frac{\partial \mathcal{E}}{\partial z_1^{(1)}} = \sum_i \frac{\partial \mathcal{E}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial z_1^{(1)}} = \sum_i \delta_i^{(2)} \sigma' \mathcal{W}_{i1}^{(2)}$$

$$= \sum_i \sum_k \delta_k^{(3)} \mathcal{W}_{ki}^{(3)} \sigma' \mathcal{W}_{i1}^{(2)}$$

Gradient descent algorithm to update the parameters

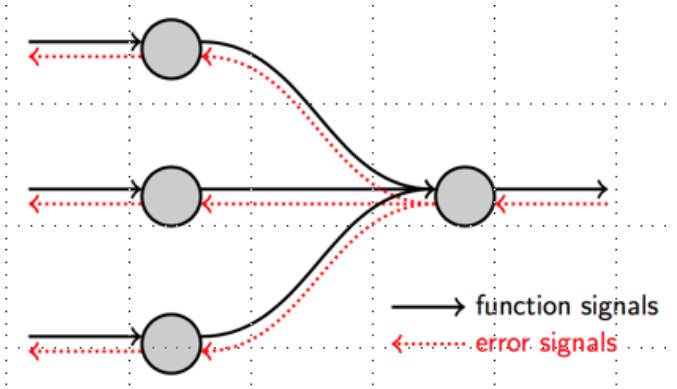
Gradient based training algorithm - Training()

for (t = 1 to T) {

1. initialized $\boldsymbol{w}$
2. ForwardPropagation()
3. BackwardPropagation()
4. $\boldsymbol{w} = \boldsymbol{w} - \eta \frac{\partial \varepsilon}{\partial \boldsymbol{w}}$

}

Parameter Updates



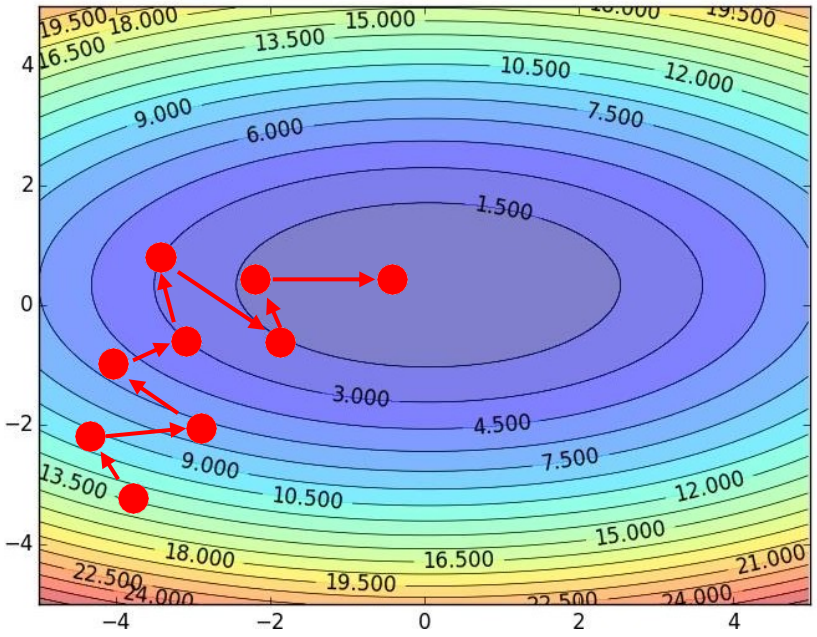
Continue updating the parameters $\boldsymbol{w}$ until the loss ε is minimized or the parameters can no longer be updated ($\frac{\partial \varepsilon}{\partial \boldsymbol{w}} = 0$)

How can we prevent vanishing gradients before the loss converges to a global minimum?

Optimized gradient descent algorithm

- ✓ Optimize the neural network with multiple sets of different initial parameters, and select the solution with the smallest error as the final parameters.
- ✓ Simulated annealing [Aarts and Korst, 1989]. At each step, it accepts results worse than the current solution with a certain probability, thereby helping to escape local minima.
- ✓ Stochastic gradient descent. Unlike standard gradient descent, which computes the gradient precisely, stochastic gradient descent incorporates random elements when calculating the gradient.
- ✓ Genetic algorithm [Goldberg, 1989]. Genetic algorithms are also commonly used to train neural networks to better approximate the global minimum.

Stochastic gradient descent algorithm



When m is large, b becomes small, and this method is referred to as mini-batch gradient descent.

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{W}} = \sum_{k=1}^{m} \frac{1}{b} \sum_{i=1}^{b} \frac{\partial \mathcal{E}(y_{ki}, \hat{y}_{ki}; \boldsymbol{W}, \boldsymbol{B})}{\partial \boldsymbol{W}}$$

When b = 1, it is called fully stochastic gradient descent (SGD).

$$\boldsymbol{W} = \boldsymbol{W} - \eta \frac{1}{b} \sum_{i=1}^{b} \frac{\partial \mathcal{E}(y_{ki}, \hat{y}_{ki}; \boldsymbol{W}, \boldsymbol{B})}{\partial \boldsymbol{W}}$$ repeat from 1 to m

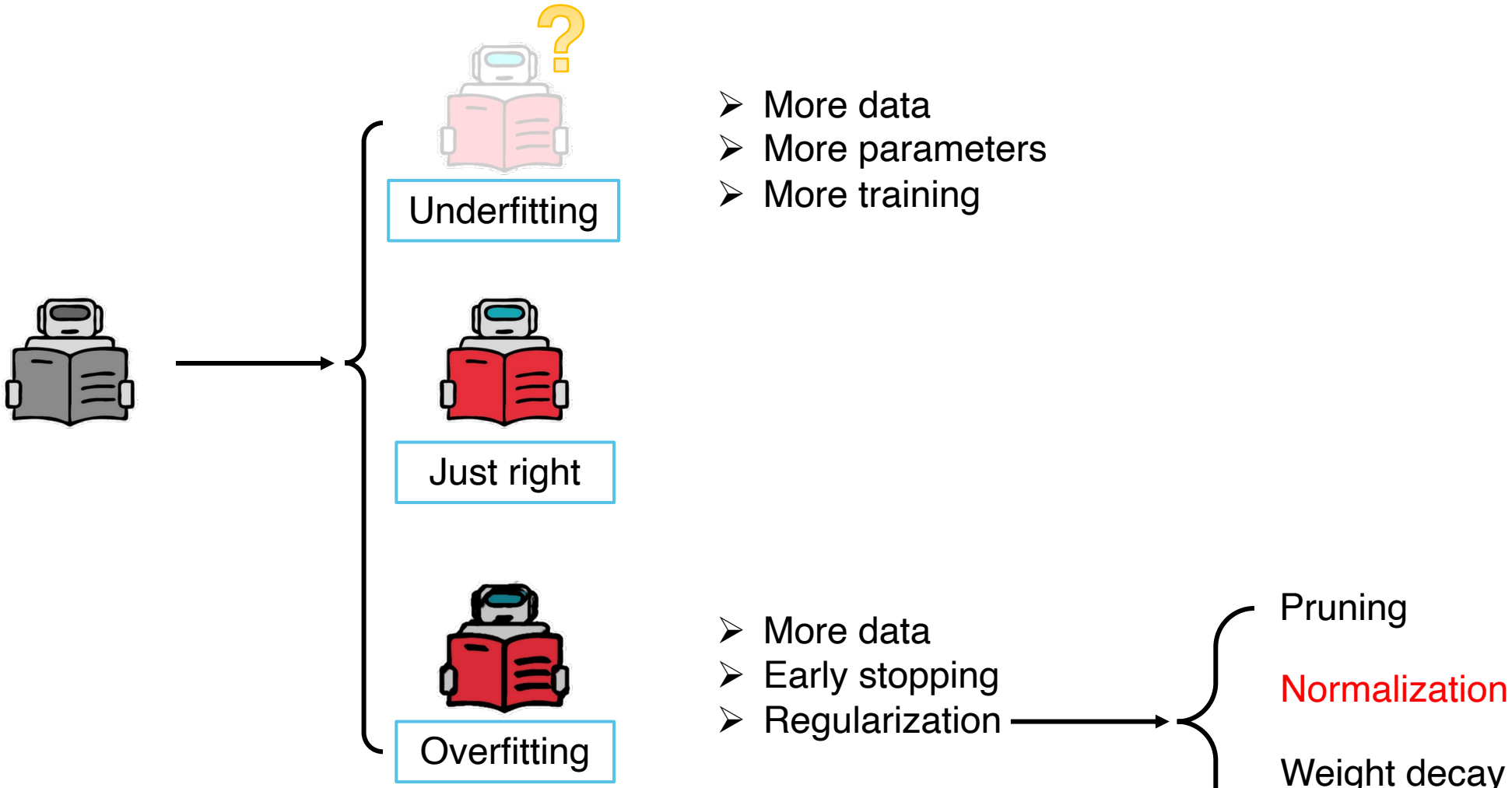
I. We partition n samples of data into m subsets of size b ($n = mb + \lambda$).

II. A full update to the parameters $\boldsymbol{W}$ constitutes one iteration and requires processing all m mini-batches.

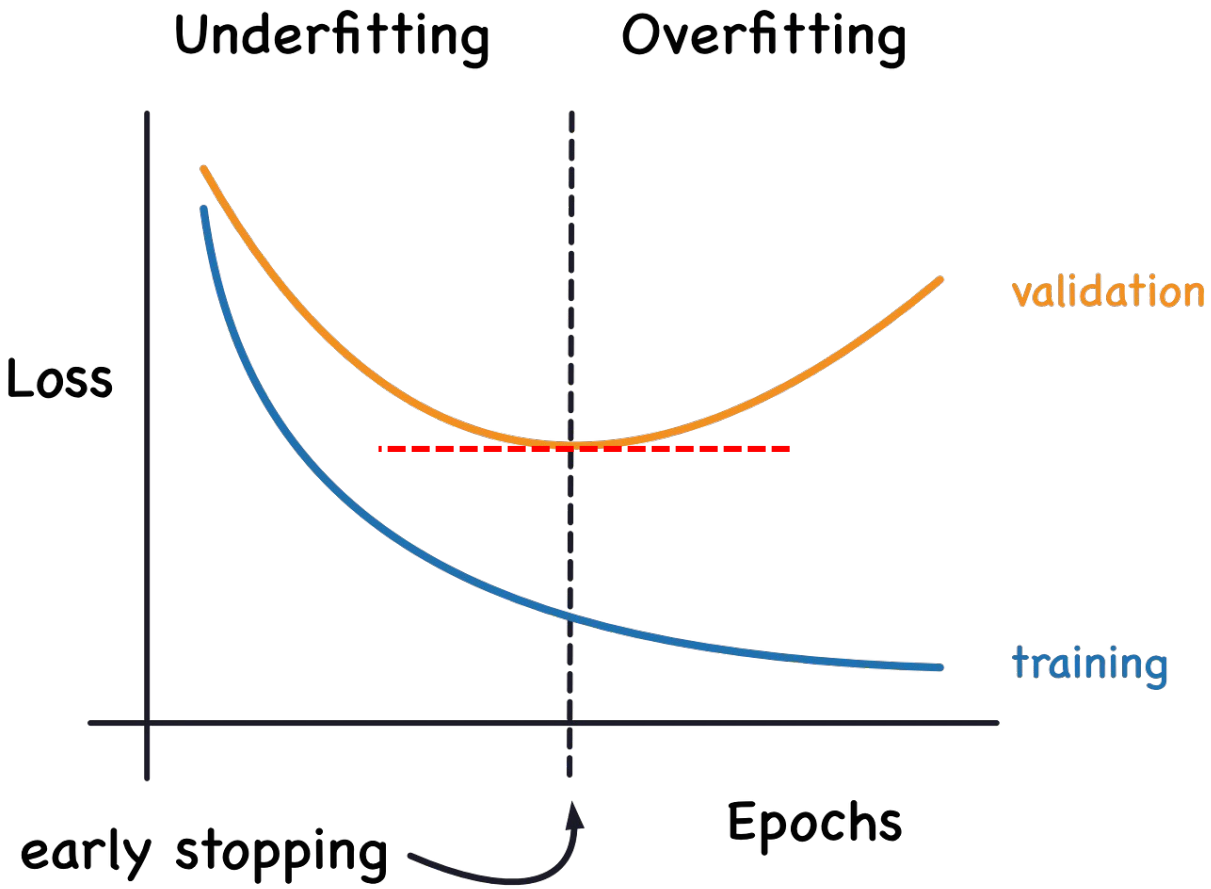
III. Throughout training, we perform T such iterations. Before each iteration, the data is reshuffled to randomize mini-batch composition, introducing stochasticity.

✅ Stochasticity helps escaping from local minima.

Overfitting and regularization



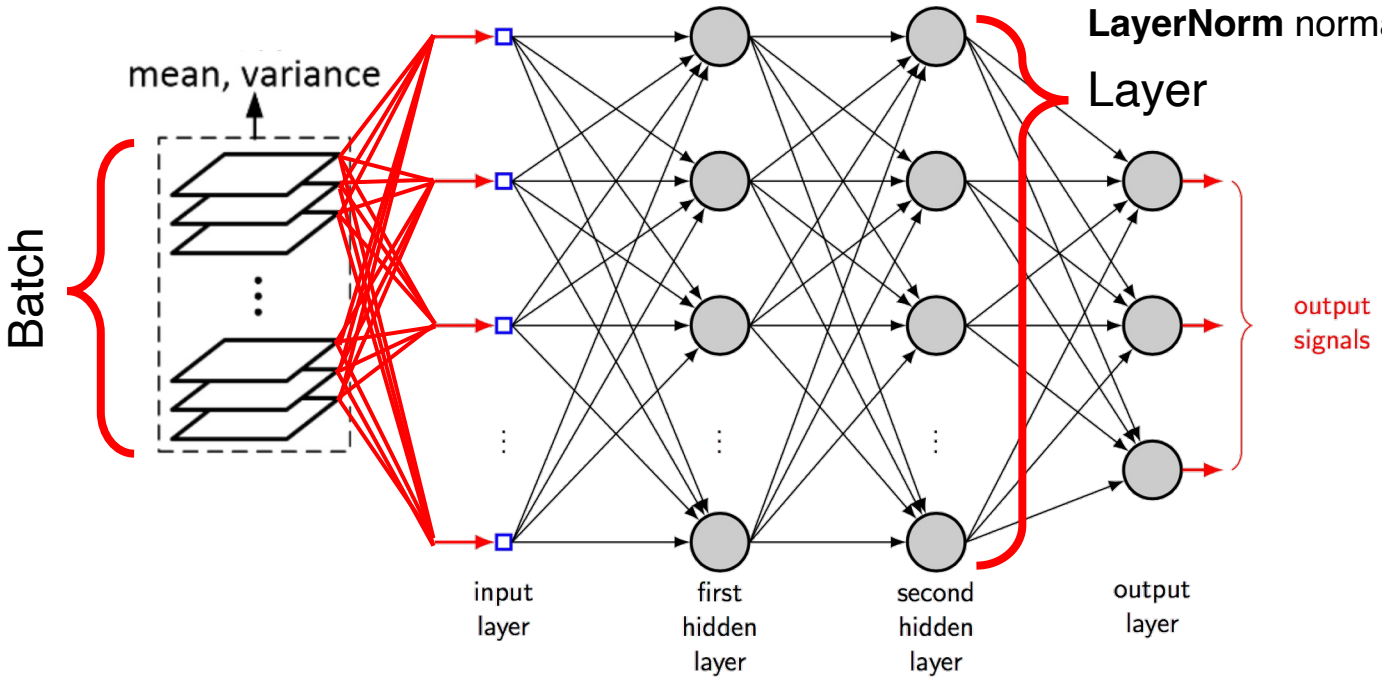
Early stopping



TrainingWithEarlyStopping()

for (t = 1 to T) {
1. initialized $\boldsymbol{\mathcal{W}}$
2. ForwardPropagation()
3. BackwardPropagation()
4. $\boldsymbol{\mathcal{W}} = \boldsymbol{\mathcal{W}} - \eta \frac{\partial \mathcal{E}_{training}}{\partial \boldsymbol{\mathcal{W}}}$
5. calculate the $\mathcal{E}$ using validation set.
6. if $\mathcal{E}_{validation}$ plateaus {
end;
}
}

Batch normalization & layer normalization

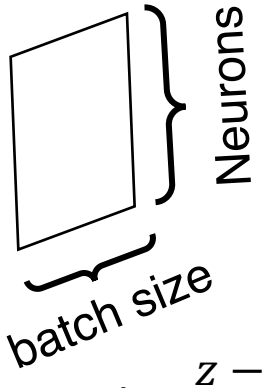


BatchNorm normalizes each feature across all samples in a mini-batch

LayerNorm normalizes all features within each sample

Layer

$$\mathcal{Z}^{(2)} = \boldsymbol{\sigma}\left(\mathcal{S}^{(2)}\right) = \boldsymbol{\sigma}(\widehat{\mathcal{Z}^{(1)}}\mathcal{W}^{(2)} + \mathcal{B}^{(2)})$$

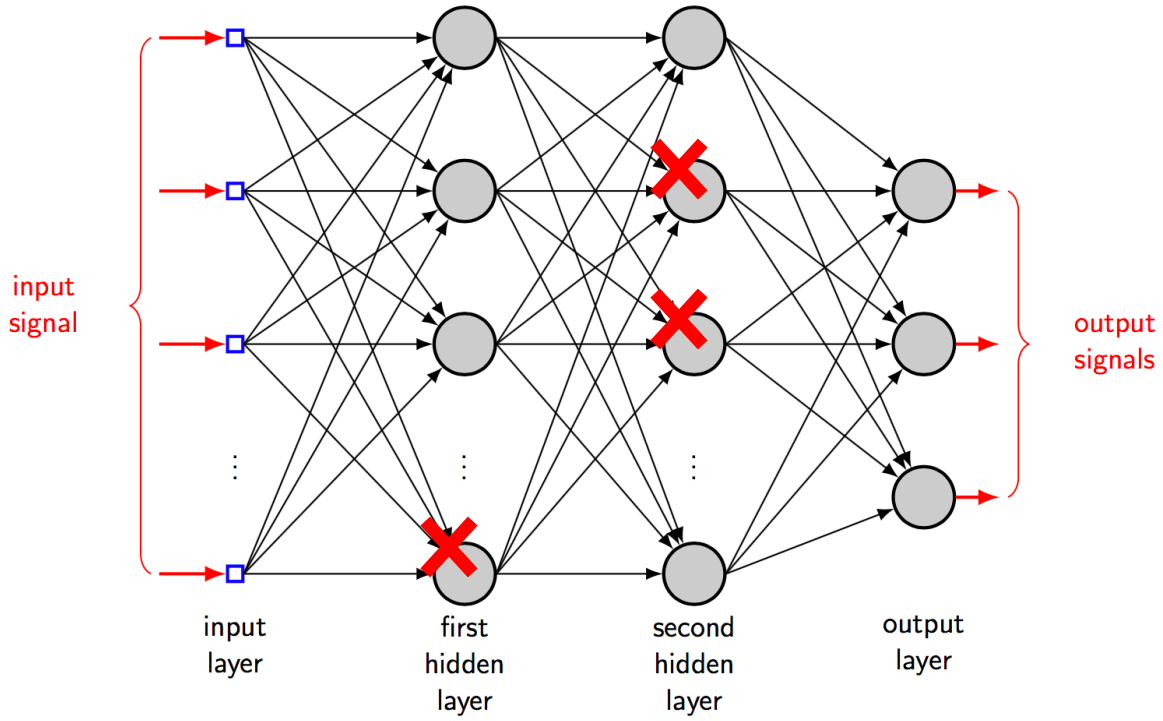


$$\hat{z} = \frac{z - u_{neuron}}{\sigma_{neuron}}$$

$$\hat{z} = \frac{z - u_{batch}}{\sigma_{batch}}$$

- They can effectively mitigate gradient explosion and vanishing gradient problems, and help prevent model overfitting.
- BN is effective with large batch sizes, whereas LN is preferred for small batches and is particularly well-suited for sequential models like RNNs and Transformers.

Dropout



training:

- Before the forward propagation of each mini-batch, each neuron has a p% probability of being dropped out.
- Resample dropout neurons each iteration.

test:

- No dropout
- If the dropout rate at training is p%, all the weights times (1-p)%

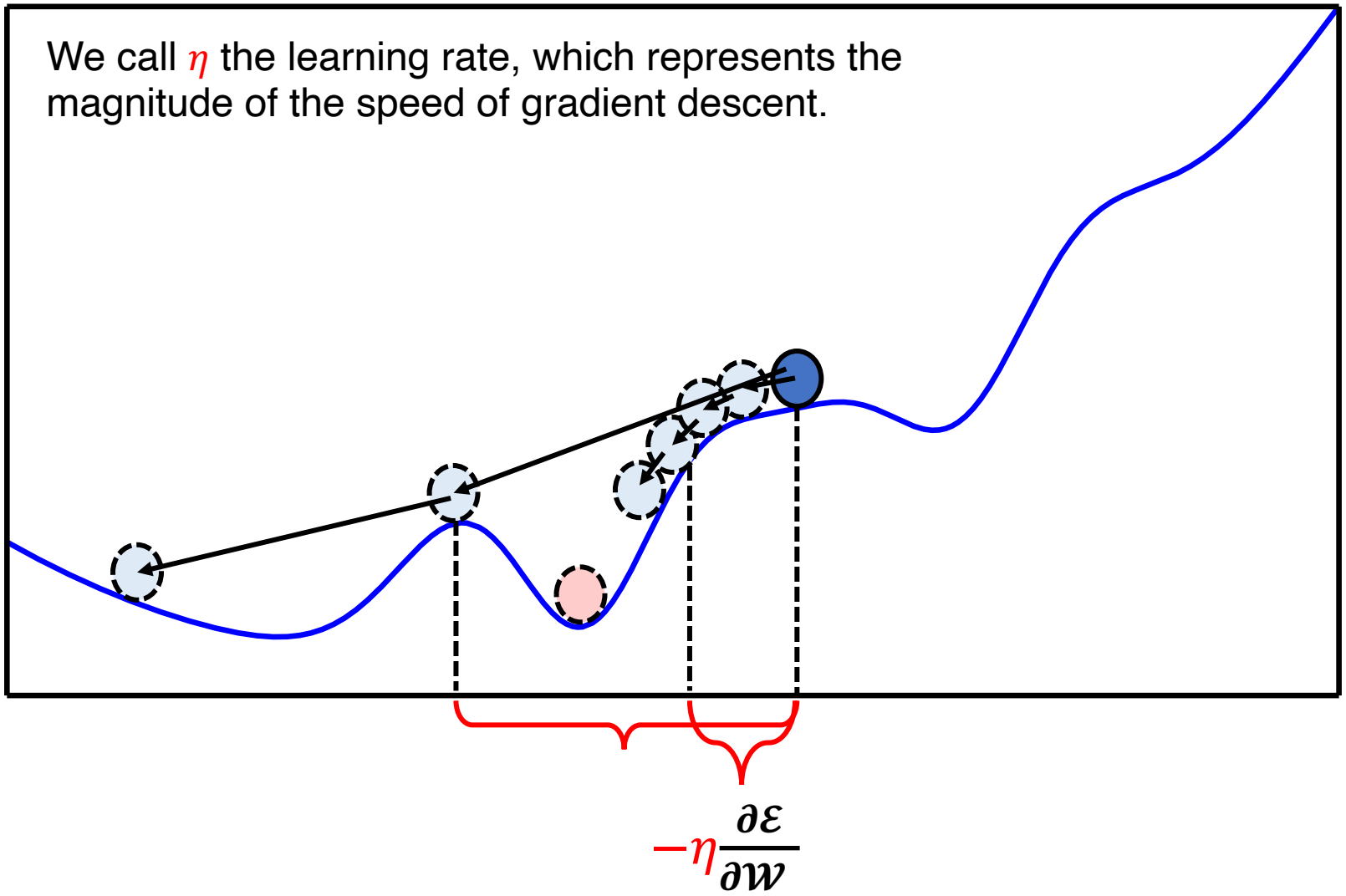
Hyperparameters

Parameters that are not obtained through model training but are used for presetting or tuning selection

- Optimizer
  - Learning rate
  - momentum
- Training hyperparameters
  - Batch size
  - epoch
  - Early stop steps
  - ……
- Weight decay rate

- Model architecture Hyperparameters
  - Number of hidden layers
  - Number of hidden layer neurons
  - Number of output size
  - Embedding size
  - ……

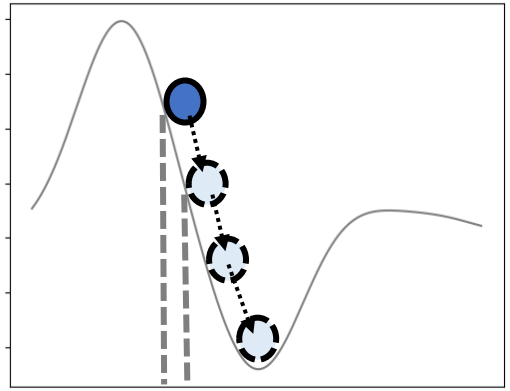
Learning rate



- If the learning rate is too high, it is very likely that the algorithm will not decrease or even skip the optimal point during each update.
- If the learning rate is too small, the update speed will be very slow each time, and it will take many iterations to reach the optimal point. Moreover, it is difficult to escape from the local minimum.

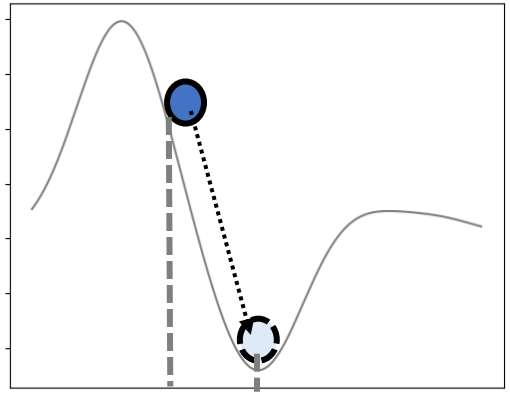
Learning rate

$$w = w - \eta \frac{\partial \varepsilon}{\partial w}$$



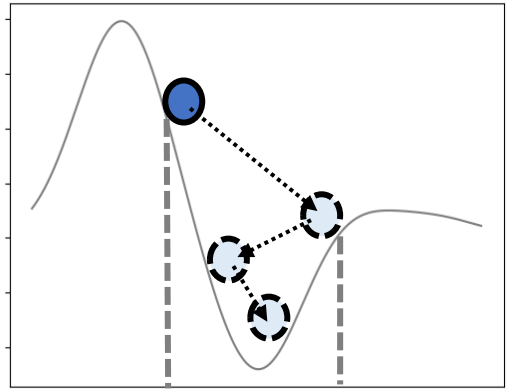
$\eta \ll \eta_{opt}$

Too slow



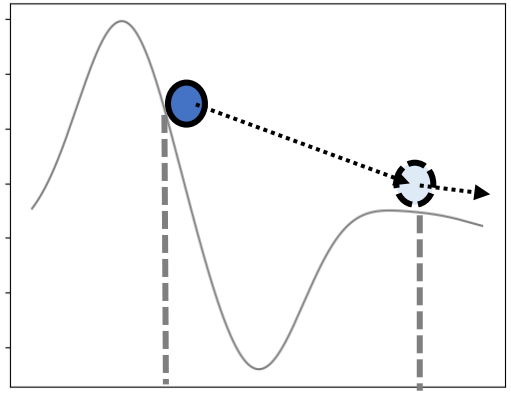
$\eta = \eta_{opt}$

Optimal



$\eta > \eta_{opt}$

Oscillate



$\eta > 2\eta_{opt}$

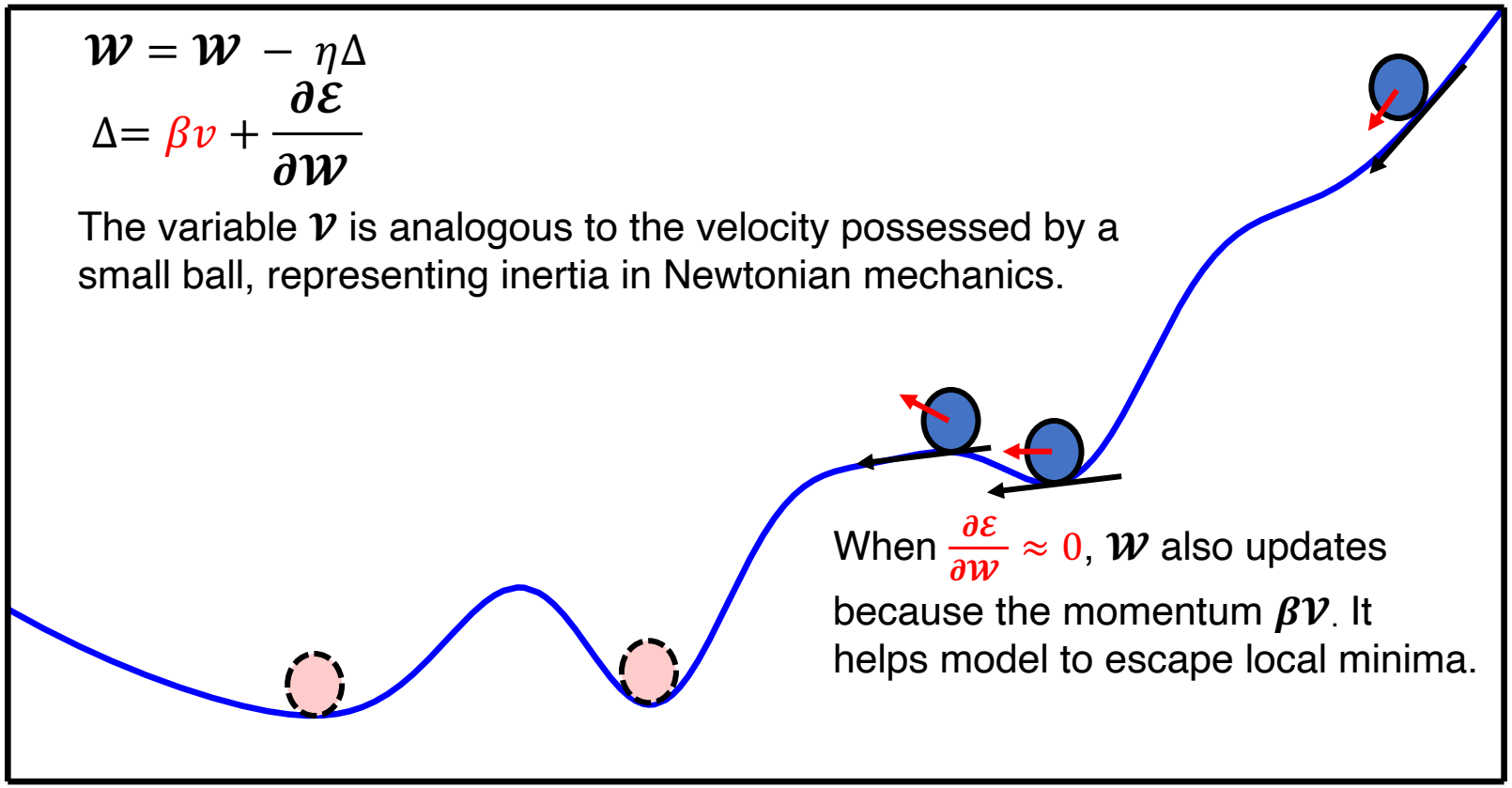
Diverge

Momentum

$$\boldsymbol{w} = \boldsymbol{w} - \eta\Delta$$

$$\Delta = \beta v + \frac{\partial \mathcal{E}}{\partial \boldsymbol{w}}$$

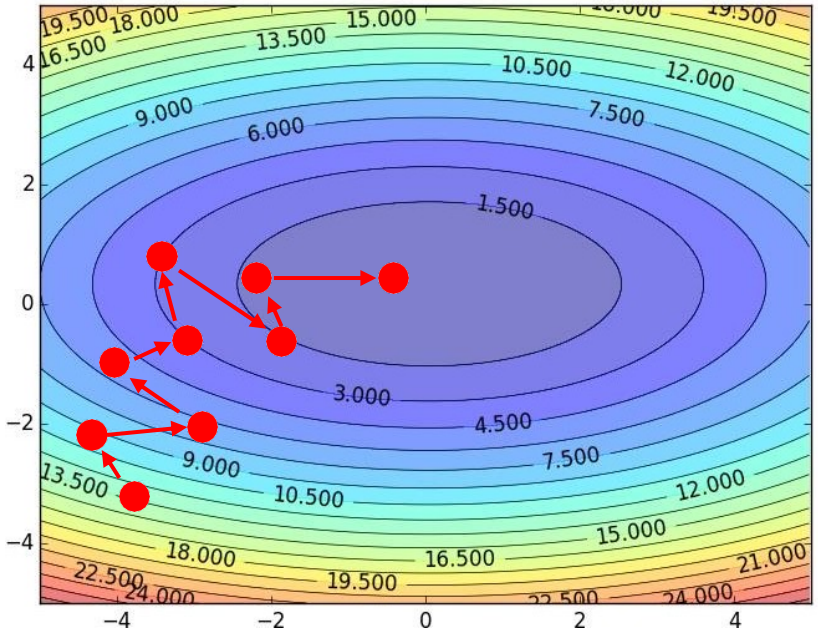
The variable $\boldsymbol{v}$ is analogous to the velocity possessed by a small ball, representing inertia in Newtonian mechanics.



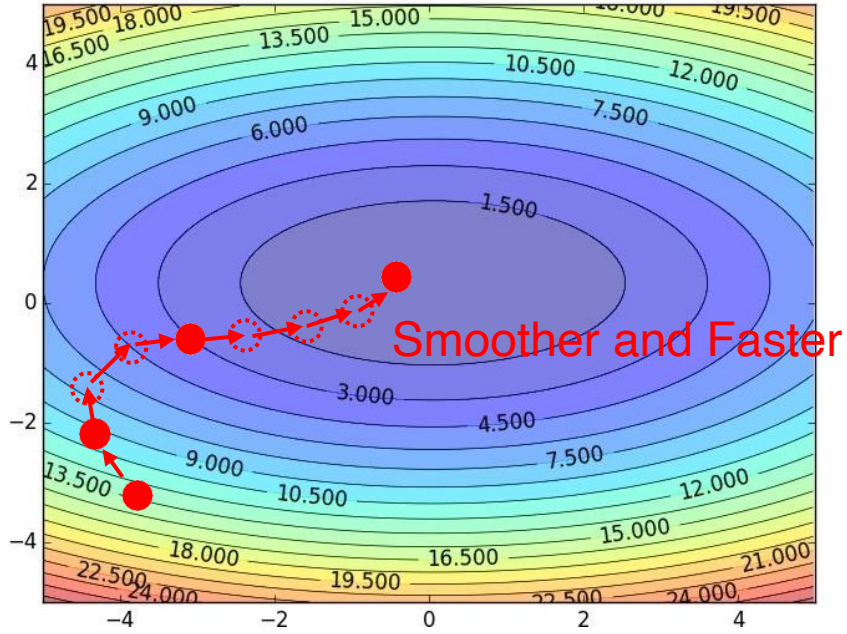
When $\frac{\partial \mathcal{E}}{\partial w} \approx 0$, $\boldsymbol{w}$ also updates because the momentum $\boldsymbol{\beta v}$. It helps model to escape local minima.

With momentum, the ball's movement is determined by both its current velocity and the gradient, instead of just the gradient alone.

Momentum



SGD



Smoother and Faster

SGD with momentum

Adaptive momentum: Adam

$$\boldsymbol{w} = \boldsymbol{w} - \Delta^{t}$$

$$\Delta^{t} = \eta \frac{\partial \mathcal{E}}{\partial \boldsymbol{w}}$$

$$\Delta^{t} = p^{t} \odot \frac{\partial \mathcal{E}}{\partial \boldsymbol{w}}$$

Adaptive learning rate

$$\Delta^{t} = \beta \Delta^{t-1} + p^{t} \odot \frac{\partial \mathcal{E}}{\partial \boldsymbol{w}}$$

Adaptive learning rate & adaptive momentum

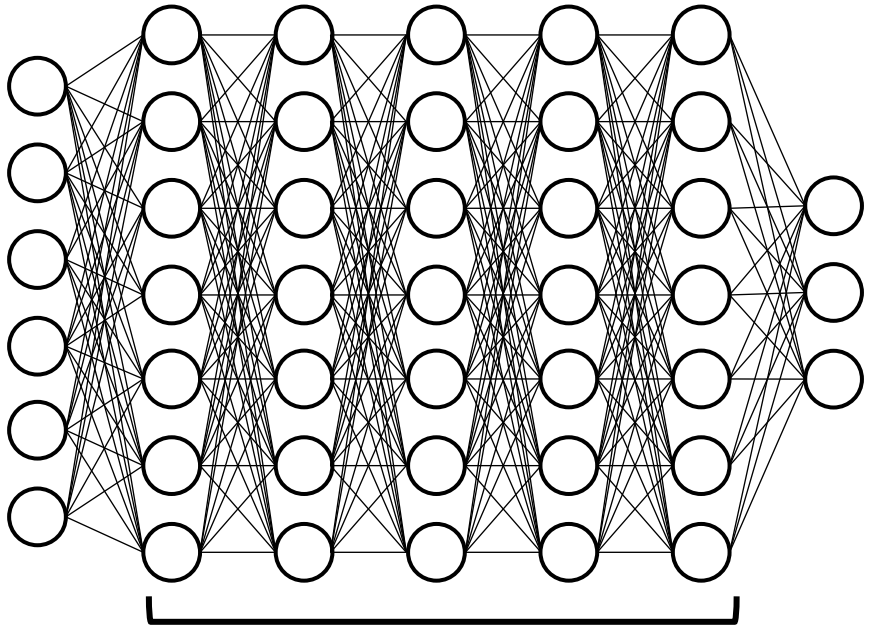
$$\Delta^{t} = p^{t} \odot s^{t}$$

$$s^{t} = \beta s^{t-1} + (1-\beta) \frac{\partial \mathcal{E}}{\partial \boldsymbol{w}}$$

SGD → RMSProp → RMSProp with momentum → Adam

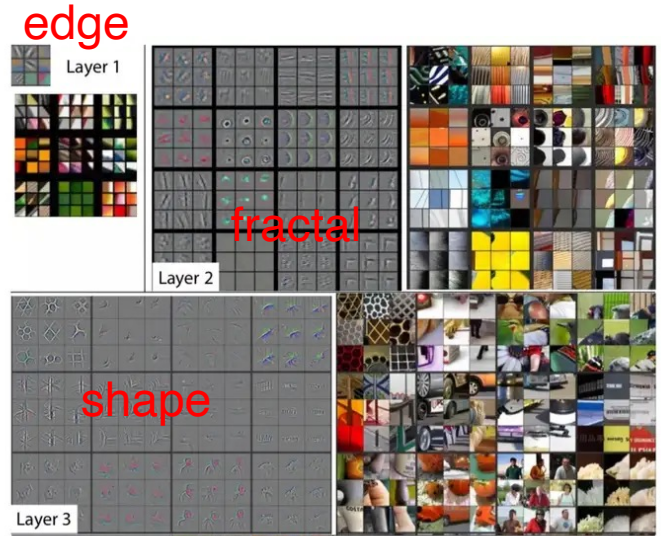
Most commonly used

Expressiveness of DL: role of depth



Depth: layers of neural network

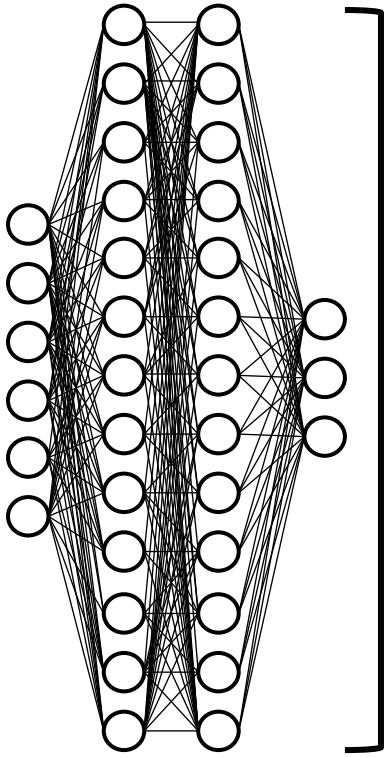
Layer X Size	Word Error Rate (%)
1 X 2k	24.2
2 X 2k	20.4
3 X 2k	18.4
4 X 2k	17.8
5 X 2k	17.2
7 X 2k	17.1
……	……



- Enhanced non-linear modeling capability, enabling the network to fit more complex features.
- Hierarchical layer-by-layer feature learning.

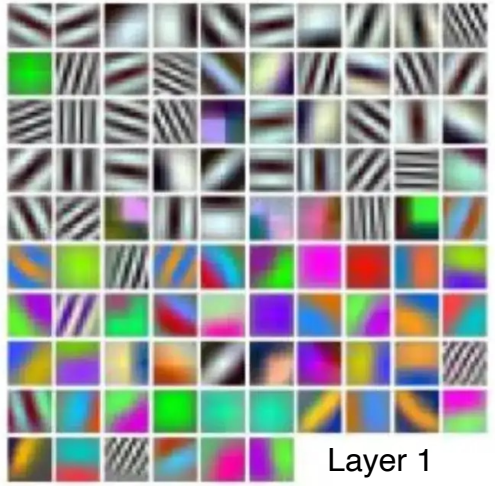
- Unstable gradients, difficult backpropagation, which may cause network degradation.
- Performance saturate.

Expressiveness of DL: role of width



Width: neurons of each layer

Layer X Size	Word Error Rate (%)	Layer X Size	Word Error Rate (%)
1 X 2k	24.2		
2 X 2k	20.4		
3 X 2k	18.4		
4 X 2k	17.8		
5 X 2k	17.2	1 X 3772	22.5
7 X 2k	17.1	1 X 4634	22.6
		1 X 16k	22.1

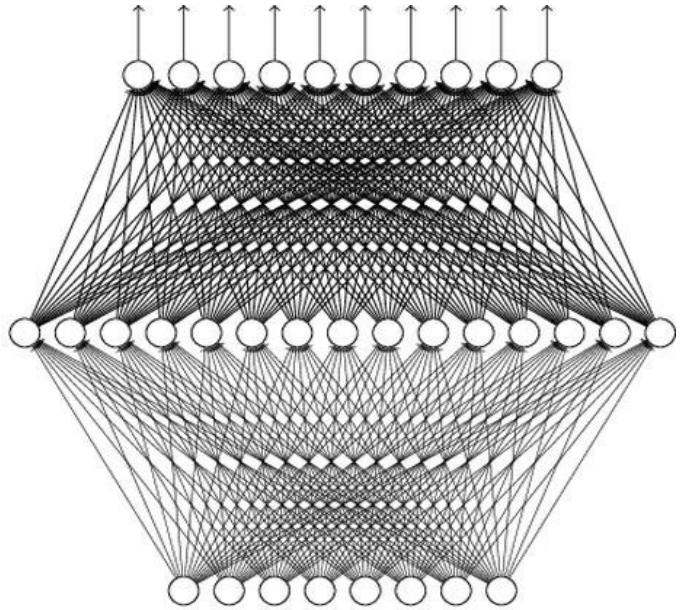


Layer 1

- Increases the feature dimension, enhancing the representational capacity within a single layer.
- smooth the loss landscape.
- Highly parallelizable and thus very GPU-friendly.

- Computational cost grows quadratically.
- Widening is exponentially more expensive than deepening for the same improvement.

Universality theorem



Any continuous function f:

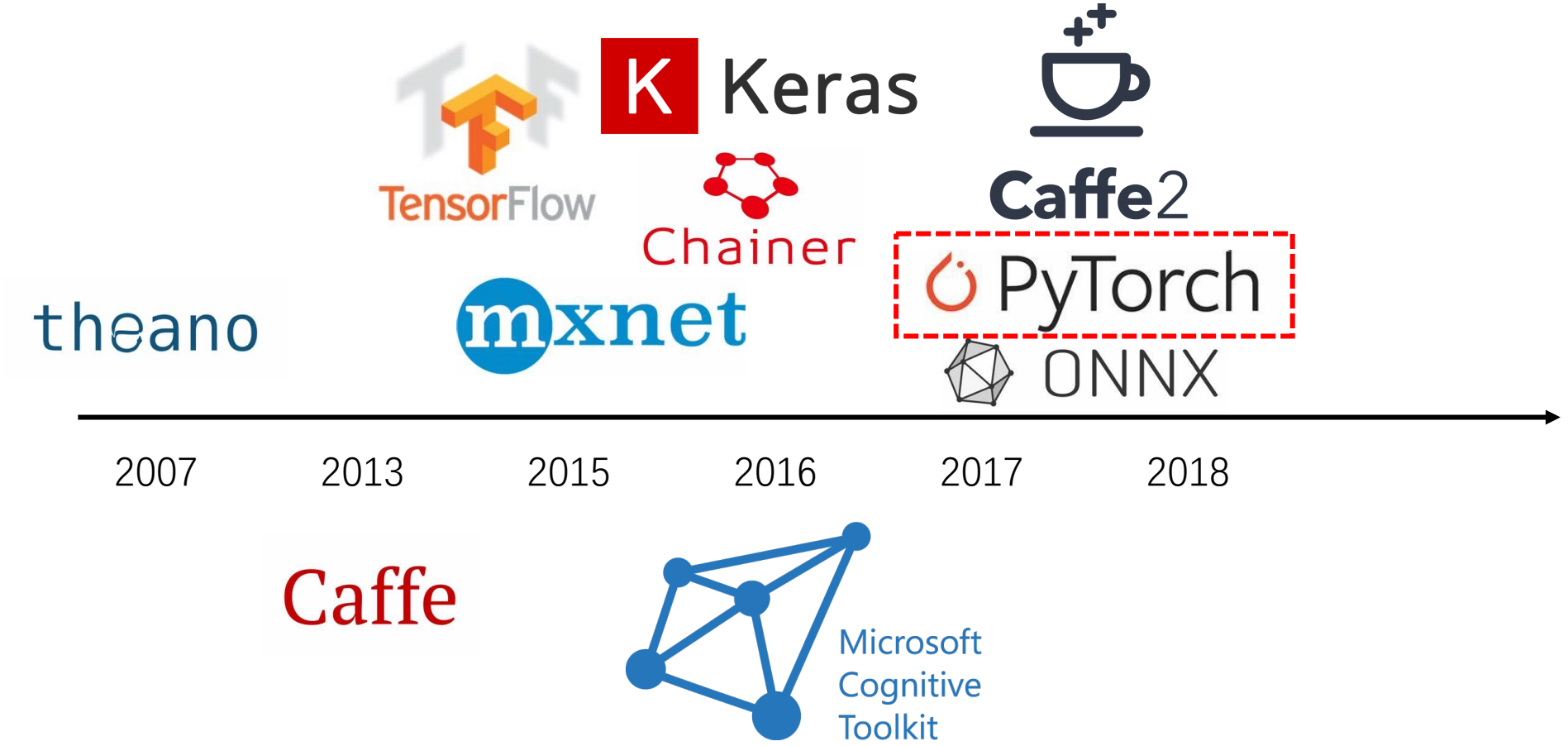
$$f\colon R^N \rightarrow R^M$$

Can be realized by a network with at least one hidden layer.

Universal approximation theorem (Hornik et al., Cybenko, 1989)

Both deeper and fatter are important, but depth is typically the primary consideration.

Deep learning community in python



Q&A