# Introduction to Deep Learning: Part II
# How to Train a Deep Neural Network

Dr. Yaohua Zang & Vincent Scholz

April 30th, 2025

# Recap (I)

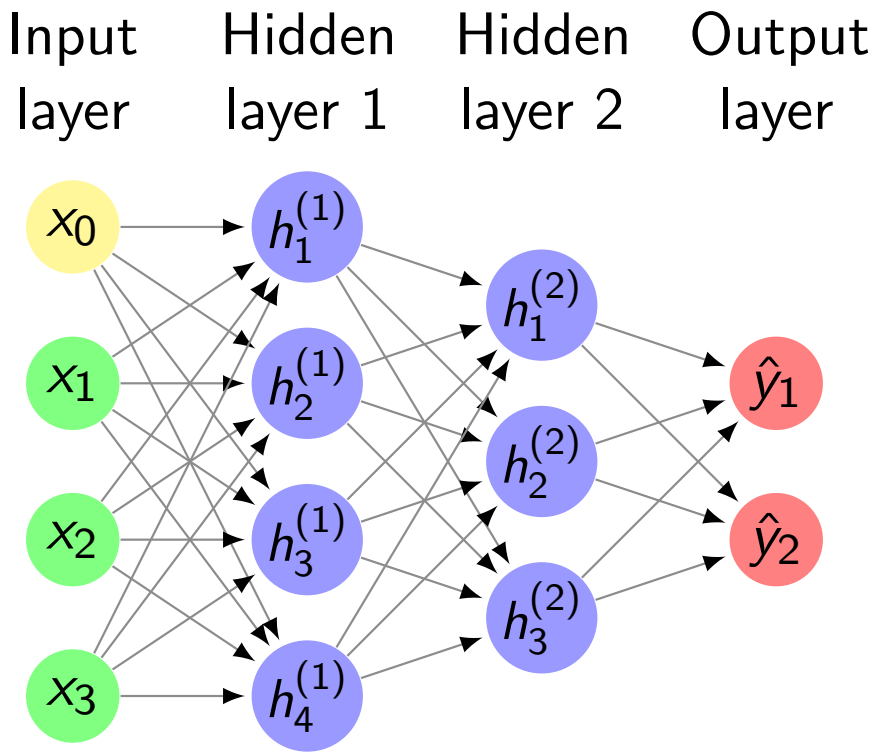
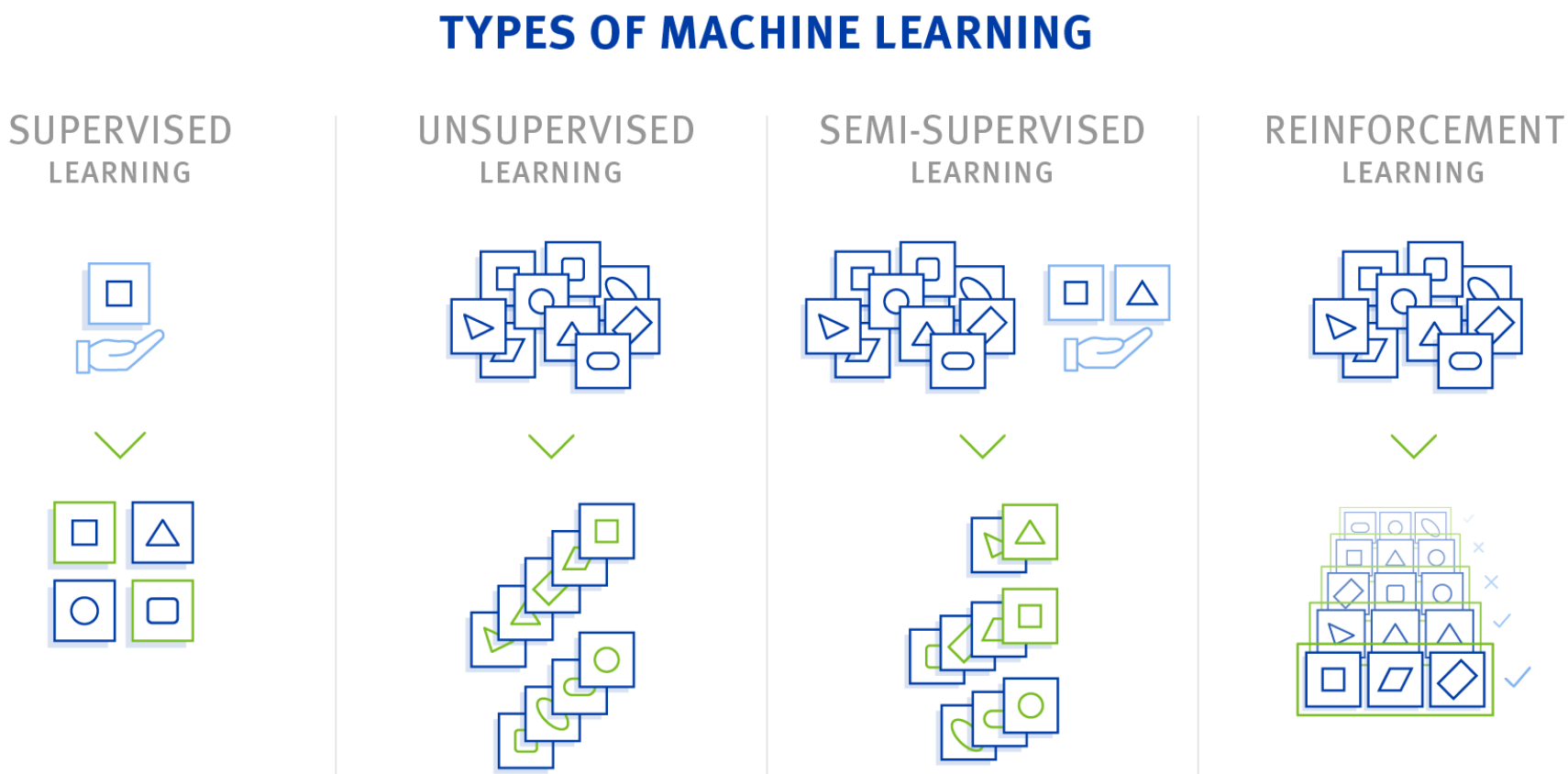
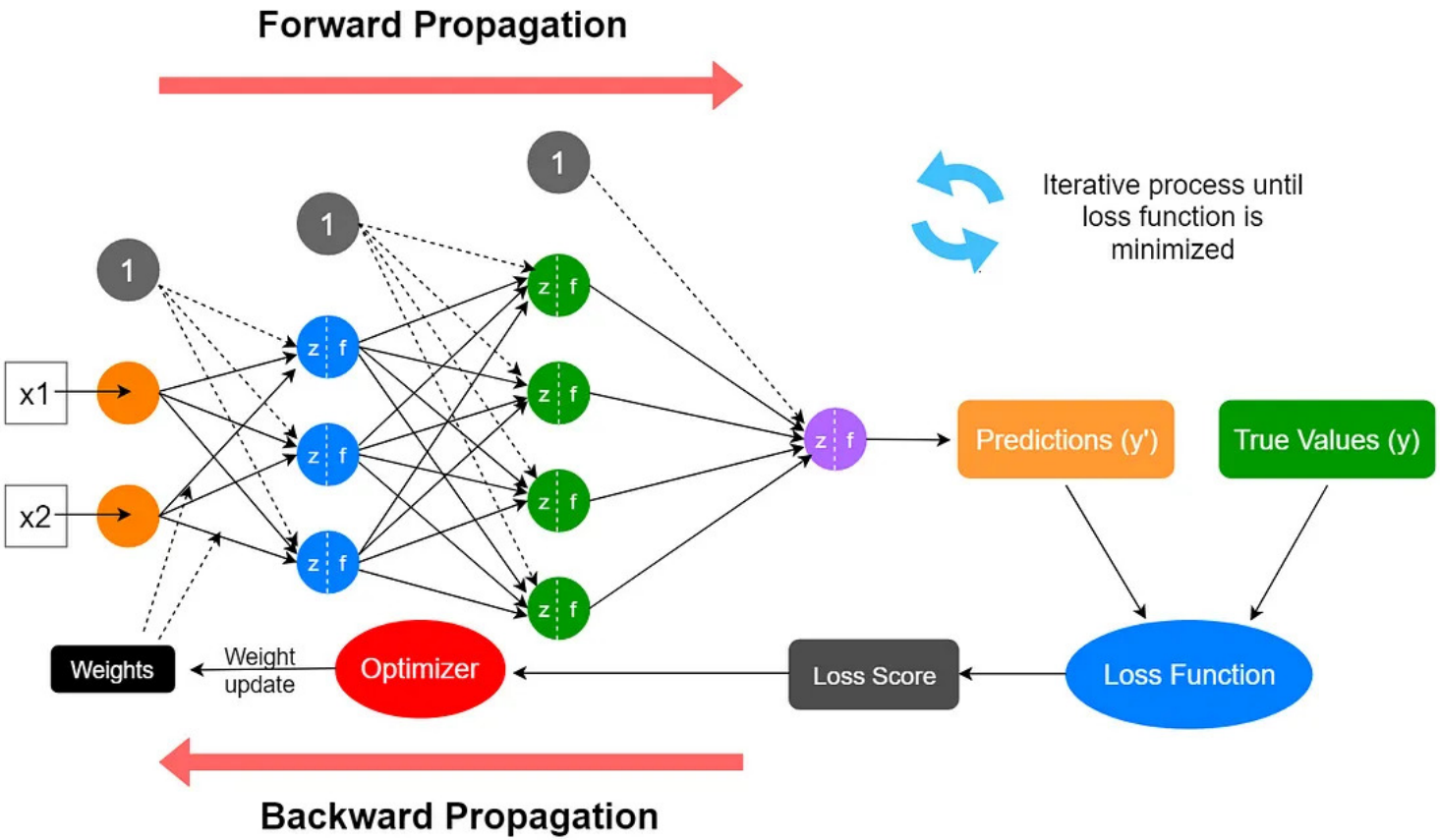



Figure: Neural Network Architecture

# Recap (II)


TYPES OF MACHINE LEARNING
SUPERVISED LEARNING
UNSUPERVISED LEARNING
SEMI-SUPERVISED LEARNING
REINFORCEMENT LEARNING

# Overview training process



# Overview

1. Preparation
   - Selecting Network Architecture
   - Initialization of Network Parameters
2. Forward Propagation
3. Training
   - Loss Function
   - Backward Propagation (Optimization)
4. Summary

# Table of Contents

# Selecting Network Architecture

The most critical aspect of deep learning that primarily influences training outcomes:

**Key considerations:**

- Data type and quantity available
- Learning task objective (supervised, unsupervised, reinforcement)

**Architecture selection process:**

1. Look for pre-trained networks that (only) require fine-tuning
2. Research literature for similar tasks or applications
3. Research wider literature (e.g., from informatics)
4. Only then consider making a NN yourself

# Guess the number of neurons: GPT-4

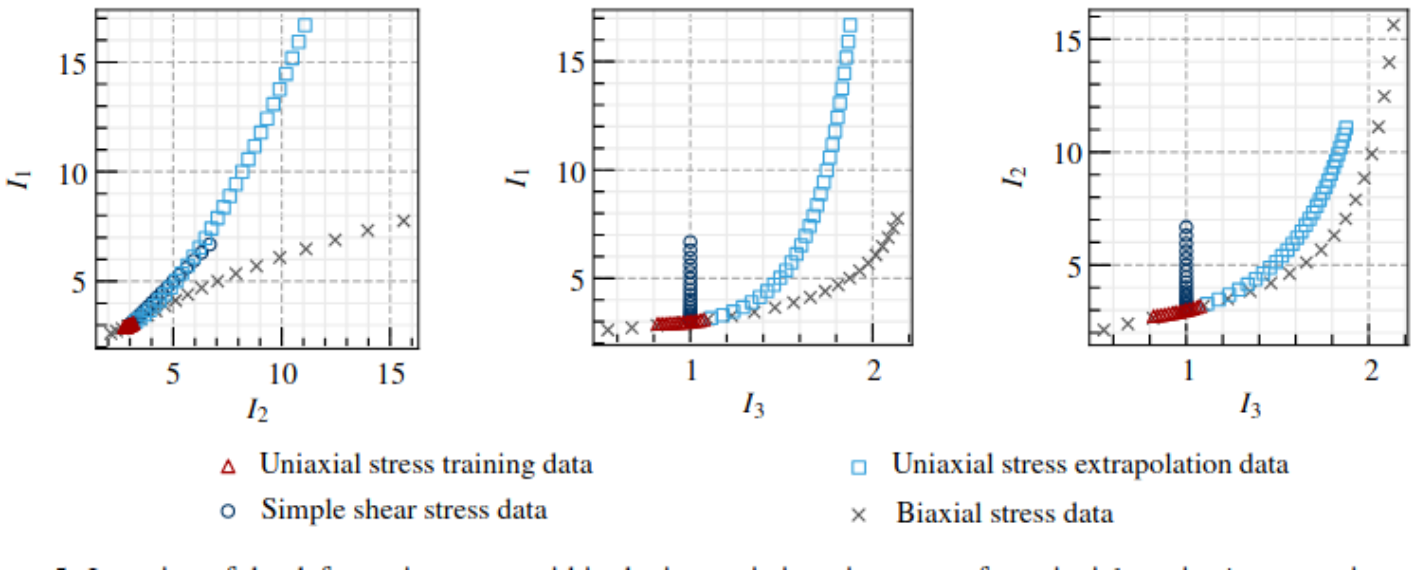
Possible range: $10^0$ to $10^{12}$

# Guess the number of neurons: GPT-4

Possible range: $10^0$ to $10^{12}$
Answer: $10^{11} = 100$ Billion

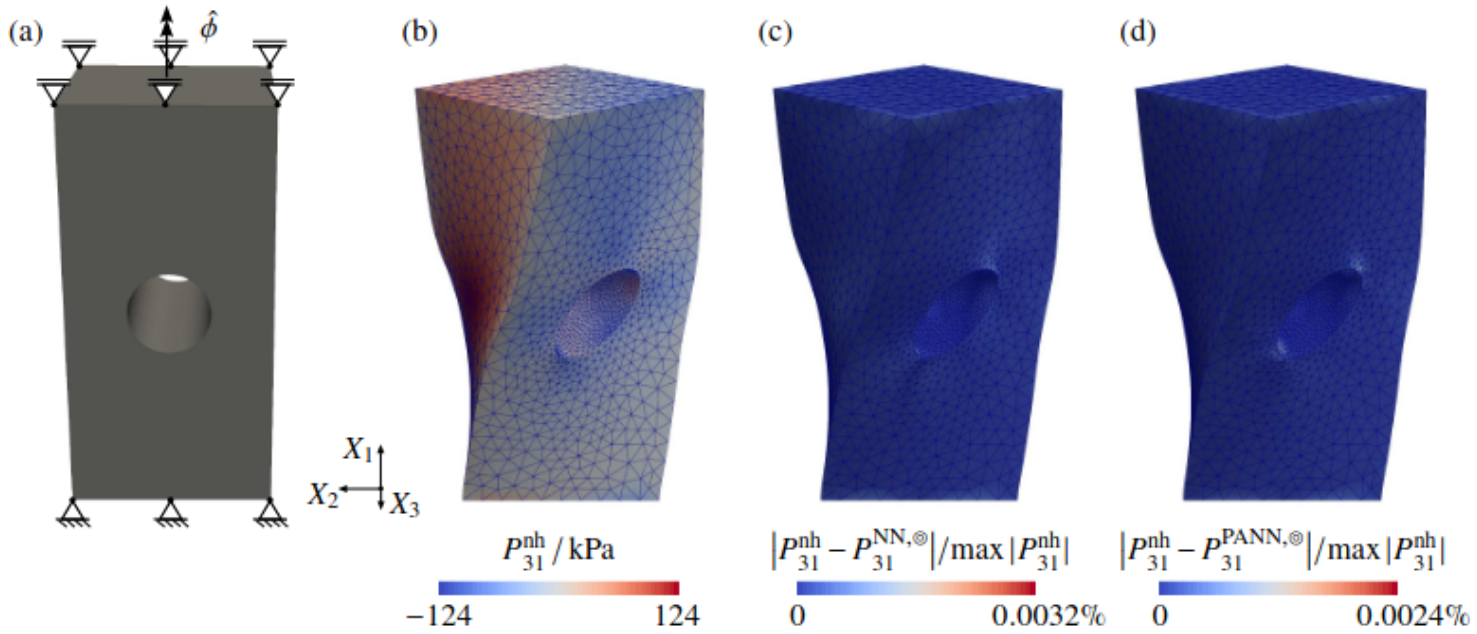
# Guess the number of neurons: Learning a material law

Possible range: $10^0$ to $10^{12}$



# Guess the number of neurons: Learning a material law

Possible range: $10^0$ to $10^{12}$



# Guess the number of neurons: Learning a material law

Possible range: $10^0$ to $10^{12}$
Answer: 4

# Architecture Considerations

**When analyzing your data:**

- Determine input/output dimensions
- Consider methods to integrate physical information
- Start with generous number of layers and neurons
  - Include at least one hidden layer
  - When number of parameters equals number of data pairs, exact learning becomes possible
- Select appropriate activation function (ReLU often suitable)
- Fine-tune hyperparameters iteratively

# Parameter Initialization

Proper initialization ensures:

- Efficient learning
- Prevention of vanishing/exploding gradients
- Improved convergence speed

**Common initialization methods:**

- Zero Initialization (Not recommended)
- Constant Initialization (Rarely used)
- Uniform Initialization (Less common)
- Normal Initialization (Most common)

# Advanced Initialization Techniques

- **Xavier (Glorot) Initialization**
  - Weights from normal distribution: mean 0, variance $\frac{1}{n}$
  - Best for sigmoid and tanh activation functions
  - Maintains variance of activations across layers
- **He Initialization**
  - Weights from normal distribution: mean 0, variance $\frac{2}{n}$
  - Recommended for ReLU and Leaky ReLU
  - Prevents vanishing gradient problem
- **Orthogonal Initialization**
  - Uses orthogonal matrix
  - Preserves gradient norms during backpropagation
  - Useful in deep architectures for numerical stability

# Table of Contents

# Forward Propagation

The process of passing input data through network layers to compute output:

$$\hat{y} = NN(x;\theta) = W^{(l)} f^{(l)} \odot \left( W^{(l-1)} \odot f^{(l-1)}(\cdots) + b_{l-1} \right) + b_l$$

Where:

- $\hat{y}$ - predicted output
- $\theta$ - all trainable parameters: $\{W^{(l)}, b^{(l)}, \ldots, W^{(1)}, b^{(1)}\}$
- $W^{(l)}$, $b^{(l)}$ - weights and biases of layer $l$
- $f^{(l)}$ - activation function at layer $l$
- $\odot$ - element-wise multiplication

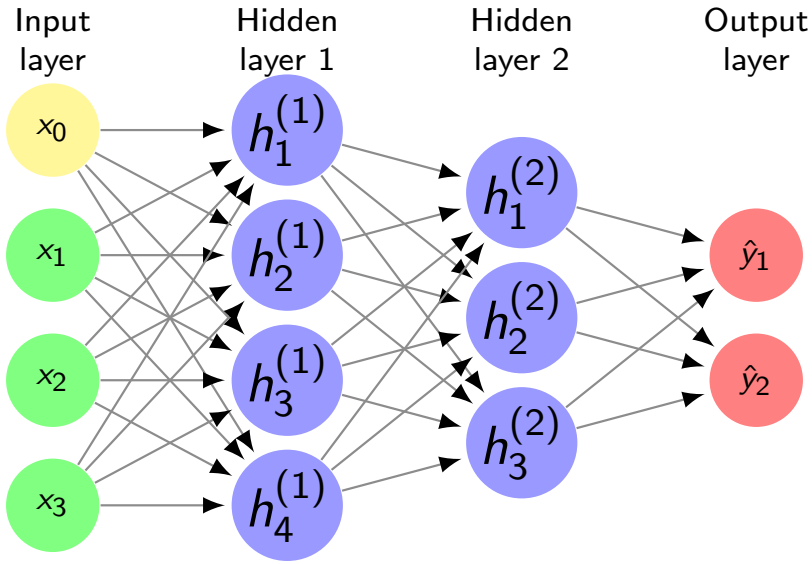



Figure: Neural Network Architecture

# Forward Propagation Steps

1. **Input Layer**: Feed input data $x$ into network
2. **Hidden Layers**: Each hidden layer applies:

$$z^{(l)} = W^{(l)} h^{(l-1)} + b^{(l)}$$
$$h^{(l)} = f^{(l)}(z^{(l)})$$

   where $h^{(l)}$ is the activation at layer $l$
3. **Output Layer**: Final layer produces predicted output $\hat{y}$
4. **Loss Calculation**: Compare prediction to actual target using loss function

# Table of Contents

# Loss Function (Cost Function)

Quantifies how well the neural network performs:

1. Pass each training example $(x_i, y_i)$ through network to get prediction $\hat{y}_i = NN(x_i; \theta)$
2. Compare $\hat{y}_i$ with actual label $y_i$ using loss function
3. Compute average loss over all training examples, e.g.:

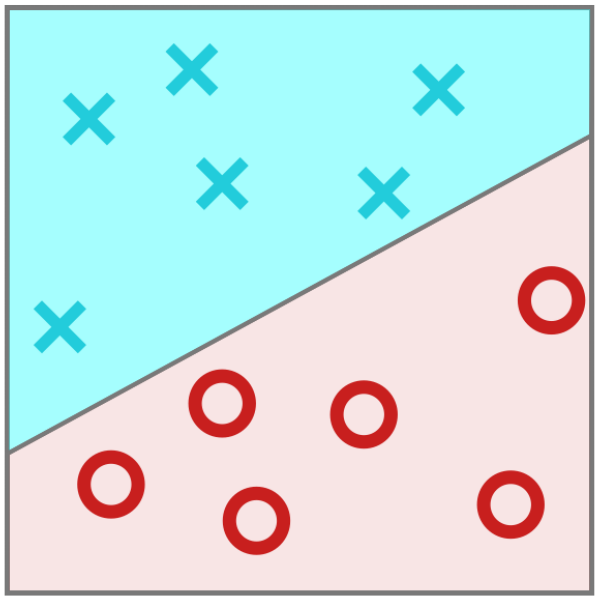
$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

# Regression vs Classification

The choice of loss function depends on the problem type:

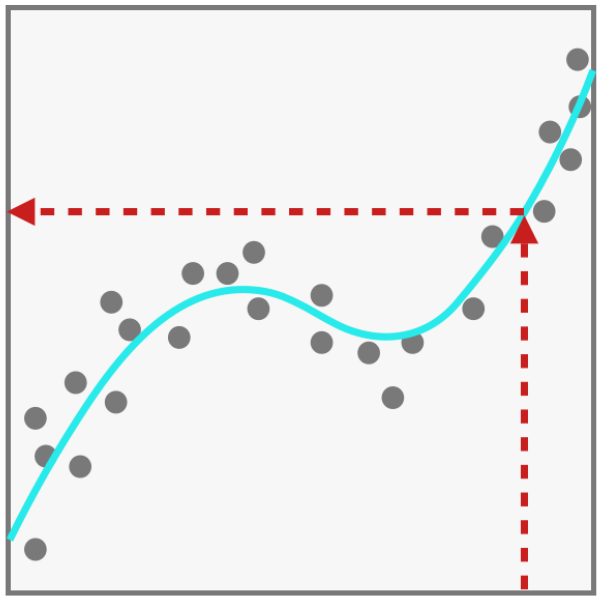
- **Regression**: Predict continuous values
- **Classification**: Predict discrete labels

**Classification** Groups observations into "classes"



Here, the line classifies the observations into X's and O's

**Regression** predicts a numeric value



Here, the fitted line provides a predicted output, if we give it an input

# Regression Loss Functions

1. **Mean Squared Error (MSE or L2 Loss)**:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

   - Amplifies larger deviations (sensitive to outliers)
   - Differentiable for gradient-based optimization
   - Ideal for normal (Gaussian) distribution targets

2. **Mean Absolute Error (MAE or L1 Loss)**:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

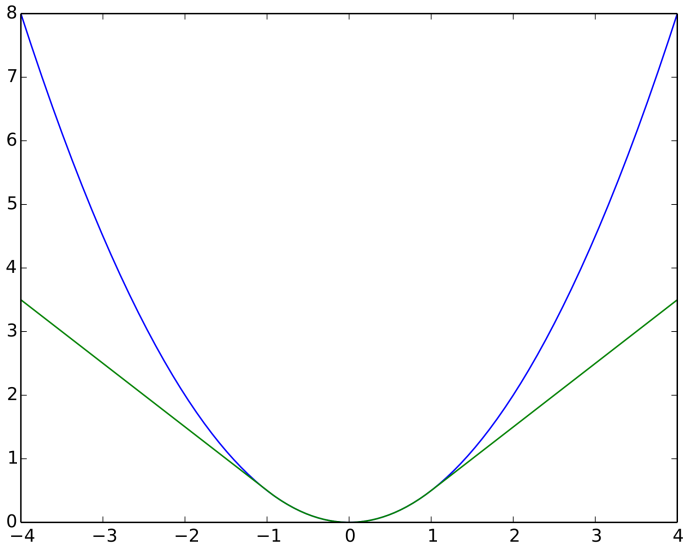
   - Less sensitive to outliers than MSE
   - Not differentiable at $y_i = \hat{y}_i$

# More Regression Loss Functions

3. **Huber Loss (Smooth L1 Loss)**:

$$\mathcal{L}_\delta(y_i, \hat{y}_i) = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2, & |y_i - \hat{y}_i| \leq \delta \\ \delta \cdot (|y_i - \hat{y}_i| - \frac{1}{2}\delta), & \text{otherwise} \end{cases}$$

- Combines advantages of MSE and MAE
- Quadratic for small errors, linear for large errors
- Robust to outliers while maintaining differentiability
- Hyperparameter $\delta$ determines transition point



# Classification Loss Functions

1. **Binary Cross-Entropy (Log Loss)**:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right)$$

- For binary classification tasks
- Penalizes incorrect predictions, encourages high confidence
- Works well for binary labels (e.g., spam vs. not spam)
- Logarithmic penalty heavily punishes confident incorrect predictions

| | $y = 0$ | $y = 1$ |
|---|---|---|
| $\hat{y} = 0.01$ | 0.01 | 4.6 |
| $\hat{y} = 0.99$ | 4.6 | 0.01 |

# Multi-Class Classification Loss

Table: One hot encoding for Multi-Class Classification

| Feature name | # Feature | One-hot encoding |
|---|---|---|
| Cat | 1 | $[1, 0, 0]$ |
| Bee | 2 | $[0, 1, 0]$ |
| Dog | 3 | $[0, 0, 1]$ |

# Multi-Class Classification Loss

**Categorical Cross-Entropy (Softmax Loss)**:

$$\mathcal{L}(y_i, \hat{y}_i) = -\sum_{j}^{c} t_j \log(\frac{e^{s_p}}{\sum_j^c e^{s_j}}) = -\log(\frac{e^{s_p}}{\sum_j^c e^{s_j}})$$

Where:

- $y_i = (t_1, \cdots, t_c)$, $\hat{y}_i = (s_1, \cdots, s_c)$
- $C =$ number of classes
- $s_j =$ predicted score for class $j$
- $s_p =$ score of correct class
- One-hot encoded targets: $t_i = 0, \forall i \neq p$ and $t_p = 1$

# Summary of Loss Functions

| Task Type | Loss Functions | Characteristics |
|---|---|---|
| **Regression** | MSE (L2) | - Smooth,<br>- differentiable,<br>- sensitive to outliers |
| | MAE (L1) | - Robust to outliers,<br>- non-differentiable at zero |
| | Huber | - Balanced approach between MSE and MAE |
| **Binary Classification** | Binary Cross-Entropy | - Encourages confident predictions<br>- Penalizes incorrect predictions |
| **Multi-Class Classification** | Categorial Cross-Entropy | - softmax for probability distribution<br>- Not suitable for multi-label |

# Backward Propagation

**Backward Propagation of Errors**:

- Fundamental algorithm for training neural networks
- Minimizes discrepancy between predicted and actual outputs
- Adjusts weights and biases using gradient of loss function

Training objective:

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y_i - NN(x_i; \theta))^2$$

# Understanding Backpropagation

**Key aspects:**

- **Iterative process** minimizing cost function
- Updates model parameters in direction of negative gradient
- Uses **chain rule** to propagate error backward through network
- Computes partial derivatives efficiently layer by layer
- Updates parameters using gradient descent or variants

# Picture Backpropagation

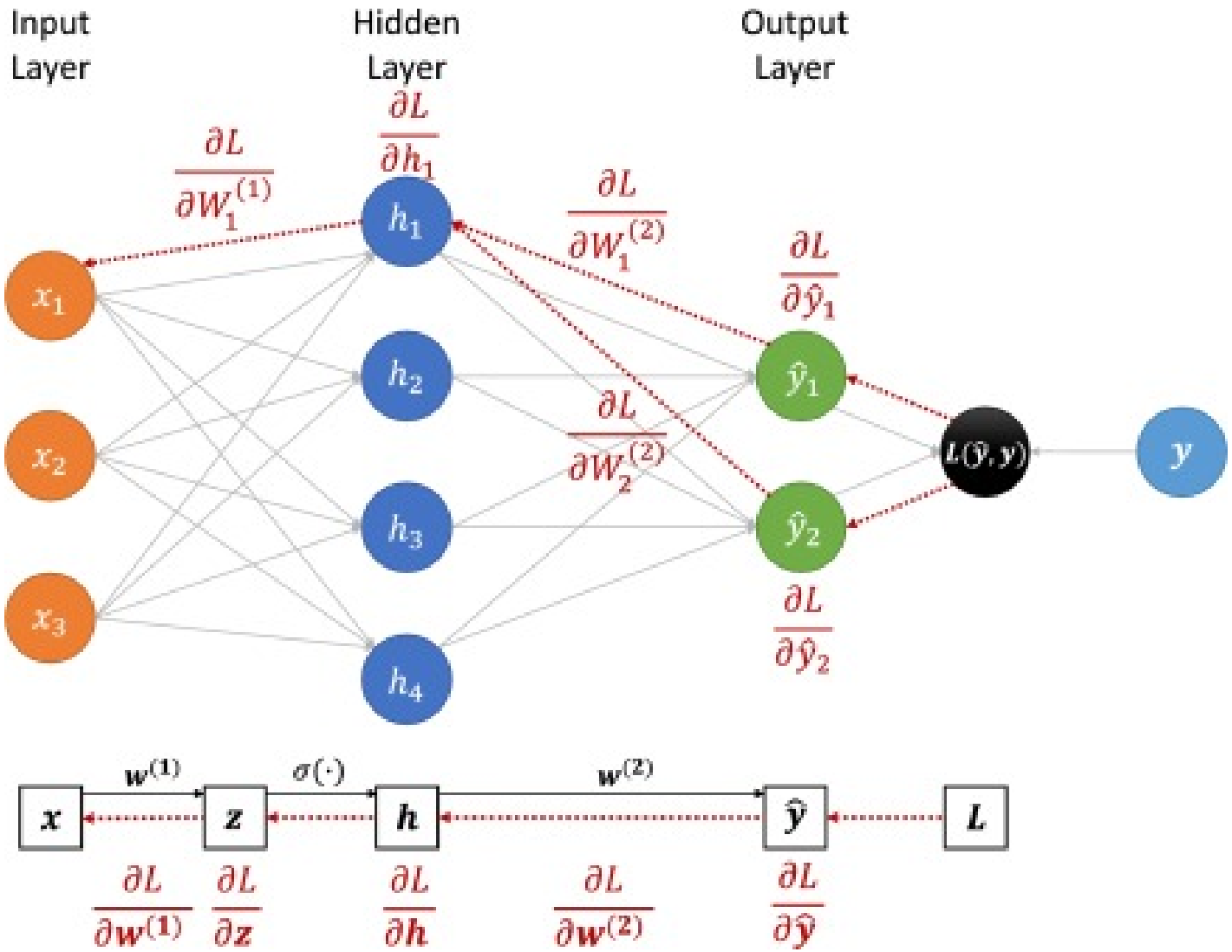



Figure: Backpropagation

# Mathematical Formulation

For a single neuron:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$
$$a = f(z)$$

Using the chain rule to compute gradients:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b}$$

Update weights and biases using gradient descent:

$$w_i \leftarrow w_i - \eta \frac{\partial \mathcal{L}}{\partial w_i}$$
$$b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b}$$

where $\eta$ is the learning rate.

# Gradient Descent

**Gradient Descent** is a fundamental optimization algorithm that:

- Minimizes cost function by iteratively adjusting parameters
- Moves in direction of negative gradient to find minimum

Parameter update rule:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \mathcal{L}(\theta_t)}{\partial \theta_t}$$

Where:

- $\theta_t$ = model parameters at iteration $t$
- $\eta$ = learning rate (controls step size)
- $\frac{\partial \mathcal{L}(\theta_t)}{\partial \theta_t}$ = gradient of loss function

# Gradient Descent Picture

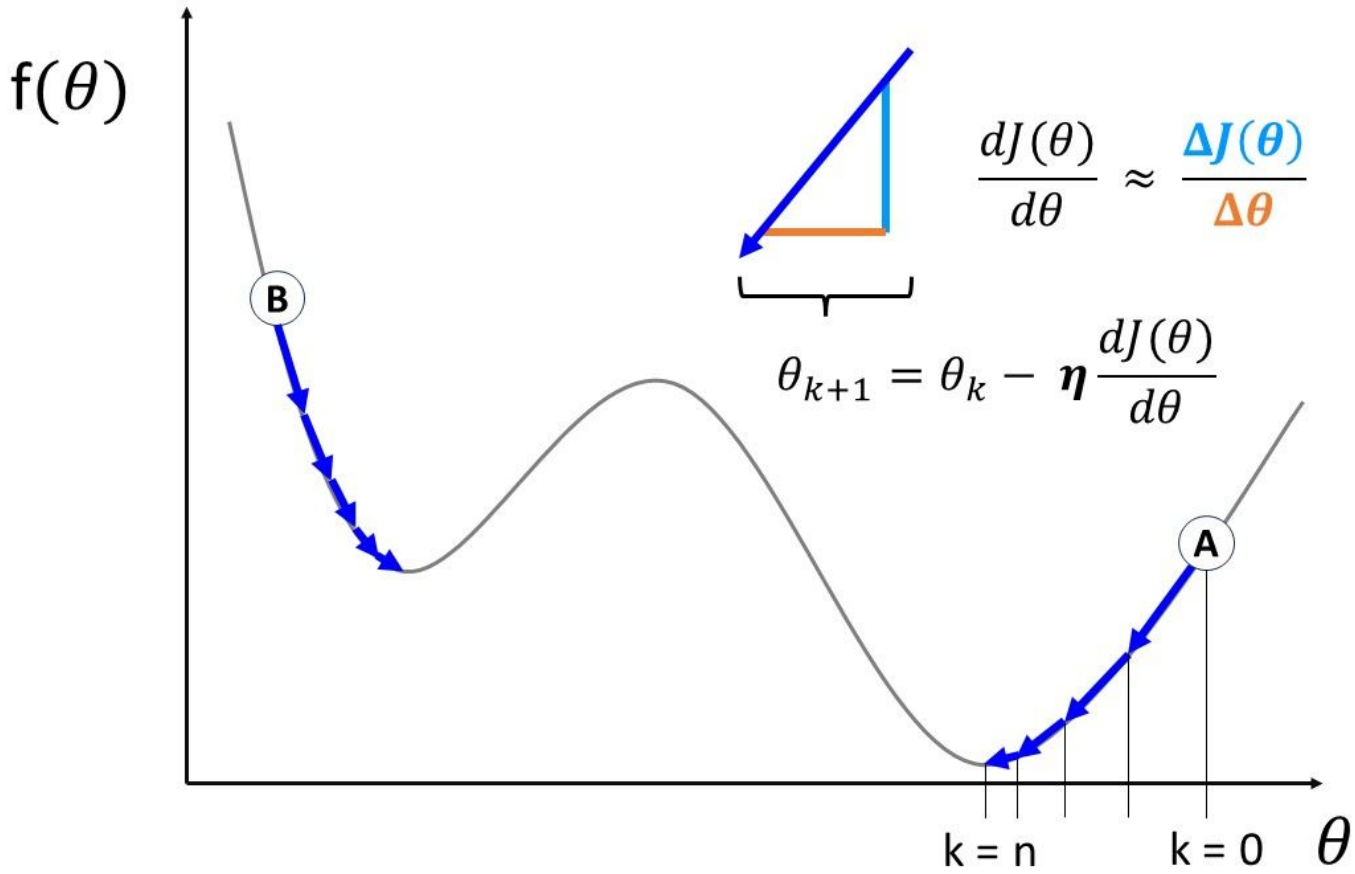



Figure: Gradient Descent.

# Importance of Learning Rate

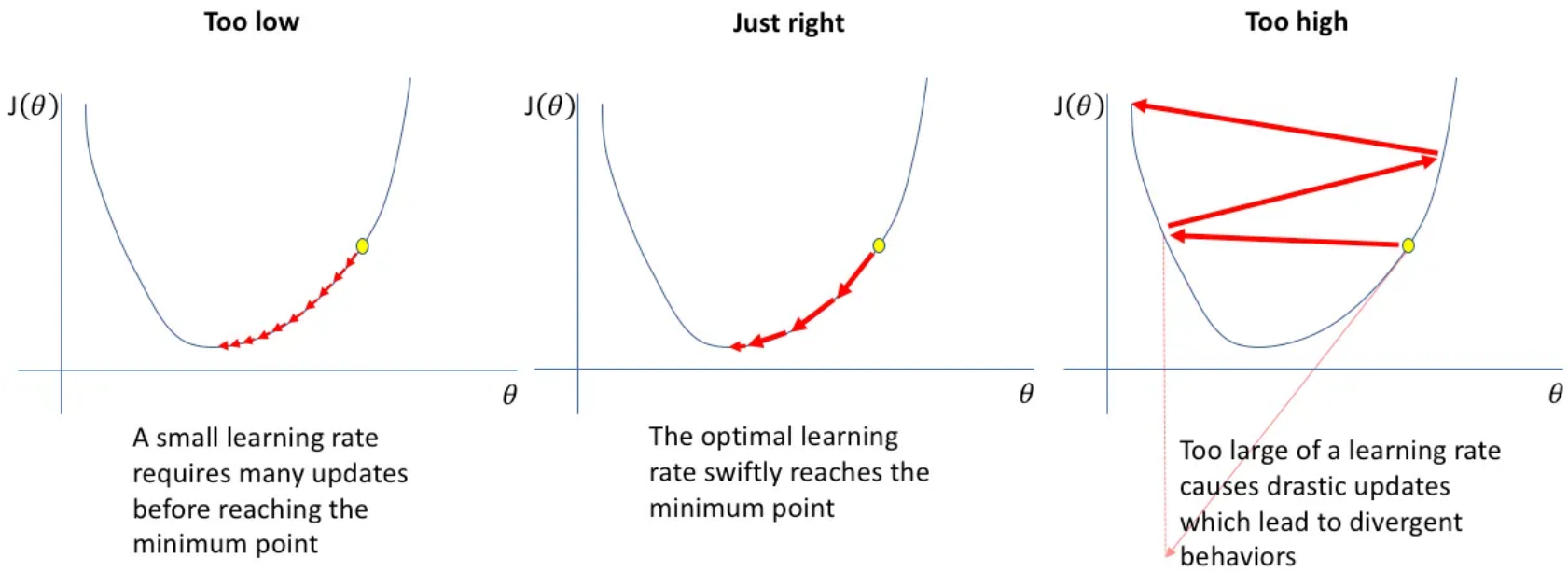
Learning rate $\eta$ is a crucial tuning parameter:

- Determines step size at each iteration
- Balances optimization time and accuracy
- **Small** $\eta$: Small steps, slow convergence
- **Large** $\eta$: Risk of overshooting minimum

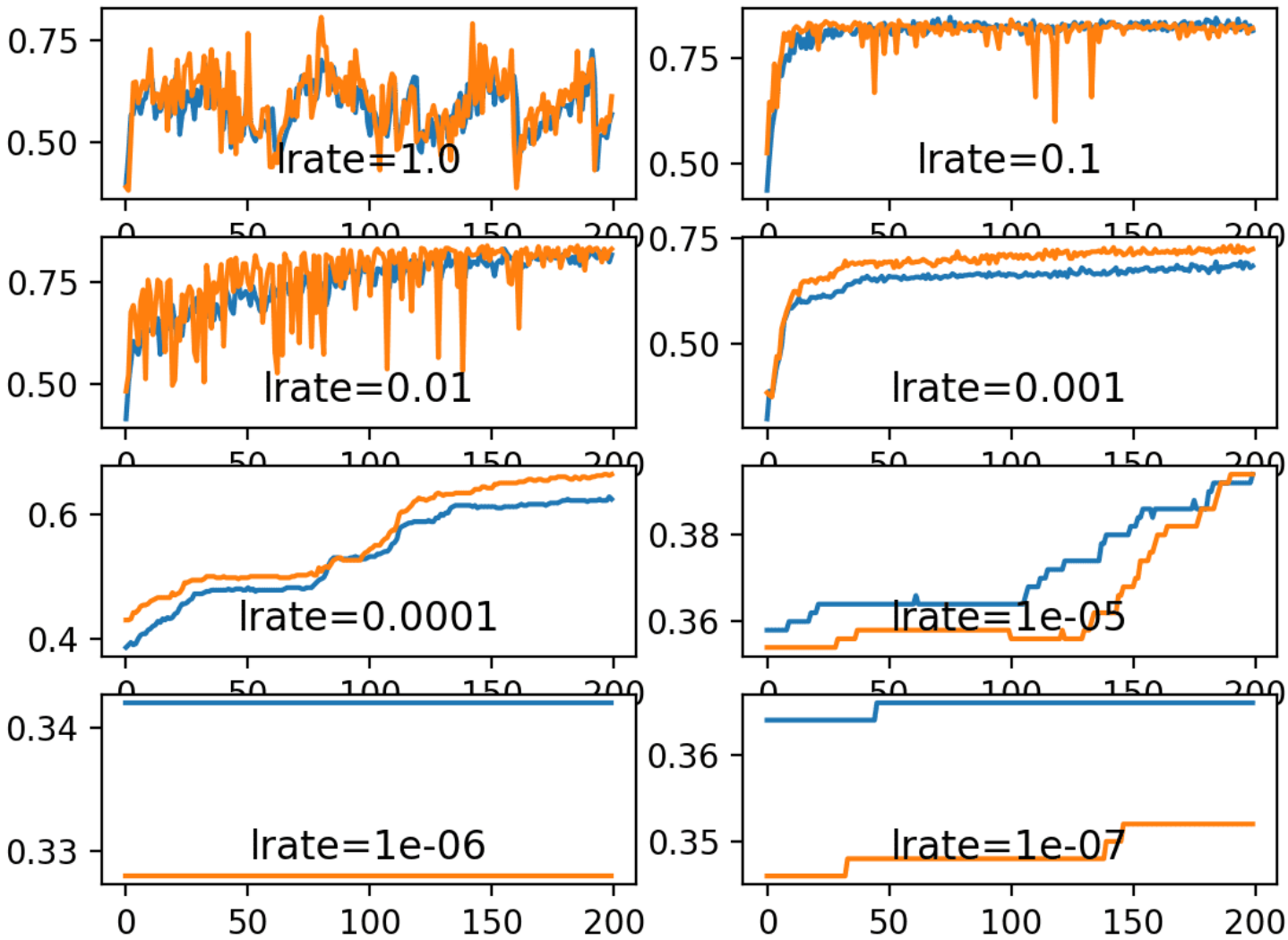
The step size depends on:

- Gradient of cost function (steepness of slope)
- Learning rate value
- Training stops when gradient becomes zero

# Importance of Learning Rate



# Importance of Learning Rate



# Variants of Gradient Descent (1)

**1. Stochastic Gradient Descent (SGD)**

- Updates using single example at a time
- Suitable for large datasets

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \mathcal{L}_i(\theta_t)}{\partial \theta_t}, \quad \text{where} \quad \mathcal{L}_i(\theta_t) = (y_i - NN(x_i; \theta))^2$$

**Pros:**

- Faster updates, suitable for large datasets
- Can escape local minima due to noisy updates

**Cons:**

- High variance in updates, may lead to instability

# Variants of Gradient Descent (2)

**2. Mini-Batch Gradient Descent**

- Updates using a small batch of examples
- Balance between computation time and precision

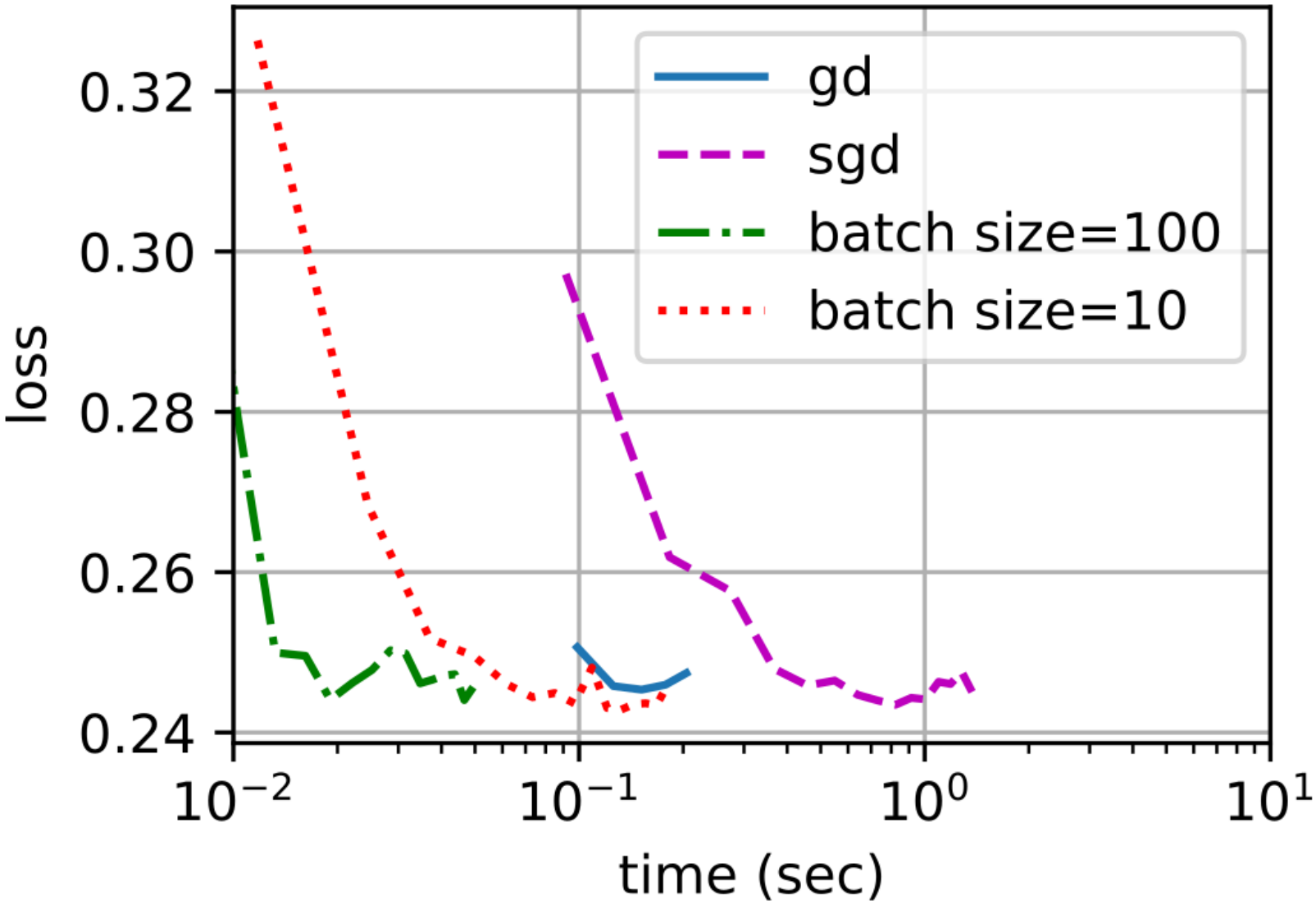
$$\theta_{t+1} = \theta_t - \eta \frac{\partial \mathcal{L}_b(\theta_t)}{\partial \theta_t}, \quad \text{where} \quad \mathcal{L}_b(\theta_t) = \frac{1}{m} \sum_{i=1}^{m} (y_i - NN(x_i; \theta))^2$$

where $m \ll n$ is the mini-batch size.

**Pros:**

- Reduces variance while maintaining computational efficiency

# Comparison SGD / Minibatch



# Variants of Gradient Descent (3)

**3. Momentum-Based Gradient Descent**

- Adds information from preceding steps to next step
- Helps penetrate flat areas and noisy gradients

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \mathcal{L}_i(\theta_t)}{\partial \theta_t} + \alpha \Delta \theta_t$$

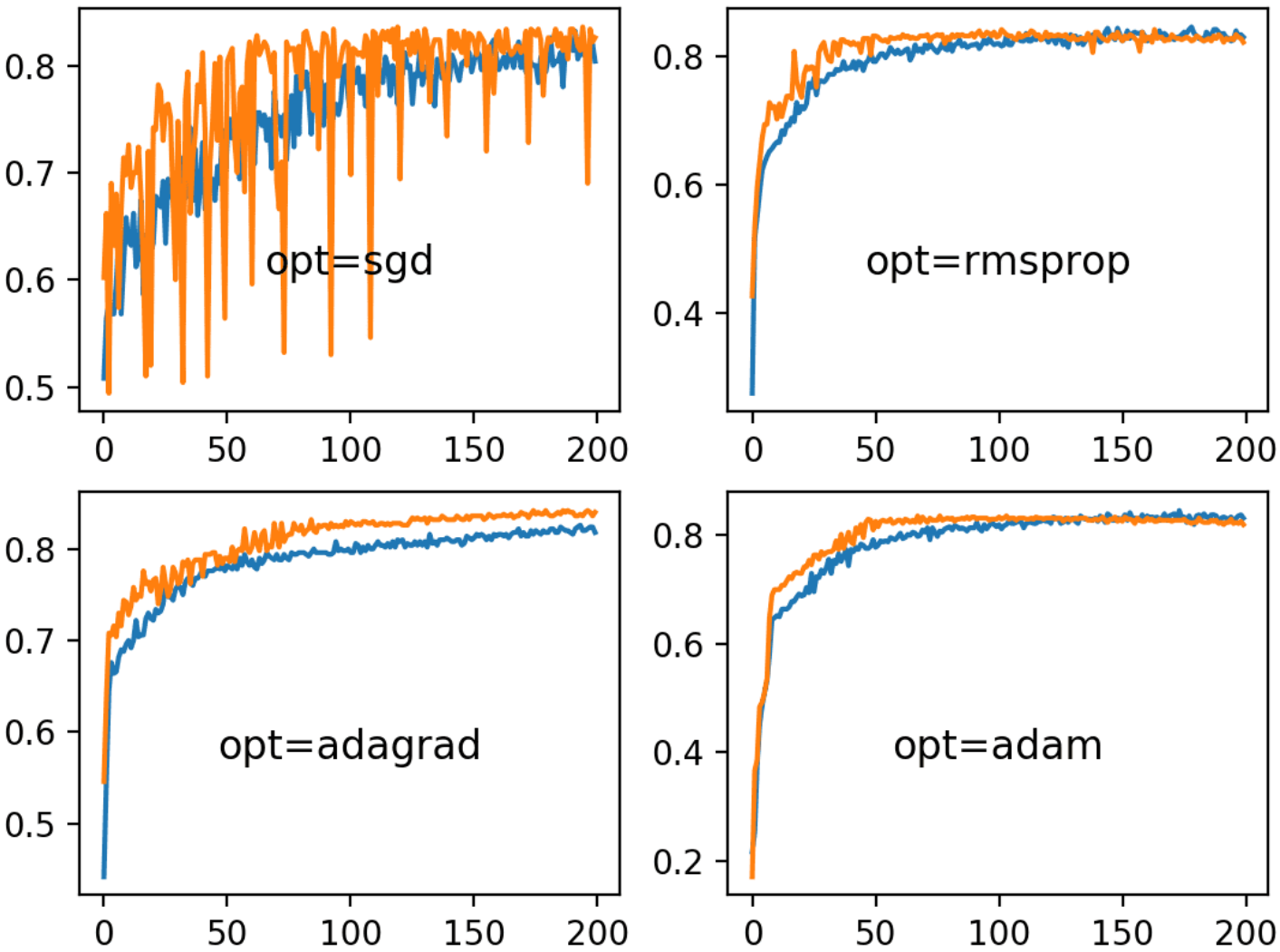
Where:

- $\Delta \theta_t$ = weight change at previous step
- $\alpha \in (0, 1)$ = momentum coefficient (typically 0.9)

**Pros:**

- Helps escape local minima
- Reduces oscillations and speeds up convergence

# Comparison SGD / Momentum-based



# Table of Contents

# Training Deep Neural Networks: Summary

1. **Network Architecture**
   - Key to data learning and processing
   - Based on data quality/quantity and domain information
2. **Parameter Initialization**
   - Initialize with suitable techniques (Normal, Xavier, etc.)
3. **Forward Propagation**
   - Pass input data through network layers
   - Apply linear transformations and activation functions
4. **Loss Function**
   - Measure prediction accuracy (MSE, Cross-Entropy, etc.)
5. **Backpropagation (Optimization)**
   - Calculate gradients and update parameters
   - Use gradient descent: $\theta_{t+1} = \theta_t - \eta \frac{\partial \mathcal{L}(\theta_t)}{\partial \theta_t}$

# References & Additional Resources


- Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.
- Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.


**Online Resources:**

- PyTorch Documentation: `https://pytorch.org/docs/stable/index.html`