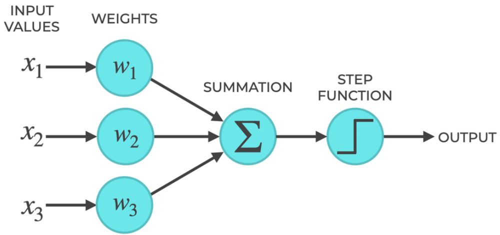
Neural Network Training

Tianxiang (Adam) Gao

School of Computing
DePaul University

Outline

Recap: Definition of MLPs



An MLP with L layers computes an output $\hat{y} = \boldsymbol{x}^L$, where each layer $\ell \in [L]$ is defined recursively as:

$$\boldsymbol{z}^\ell = \boldsymbol{W}^\ell \boldsymbol{x}^{\ell-1} + \boldsymbol{b}^\ell,$$
$$\boldsymbol{x}^\ell = \phi(\boldsymbol{z}^\ell),$$

where the initial input is $\boldsymbol{x}^0 = \boldsymbol{x}$ and $\phi(\cdot)$ is an activation function.

Conclusion

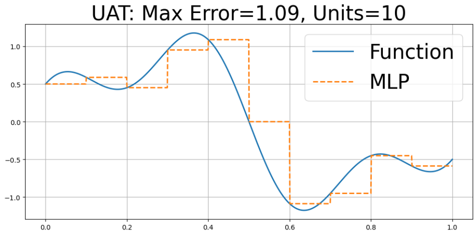
MLPs can solve **nonlinear problems** like XOR that a single perceptron cannot handle.

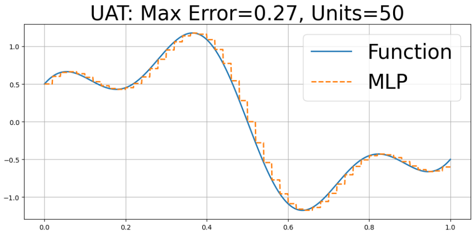
Outline

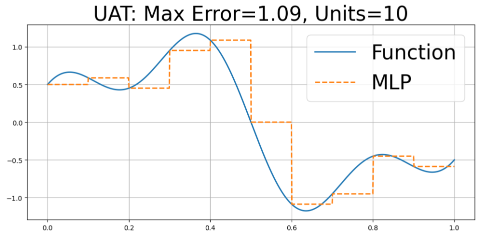
Universal Approximation Theorem (UAT) of MLPs

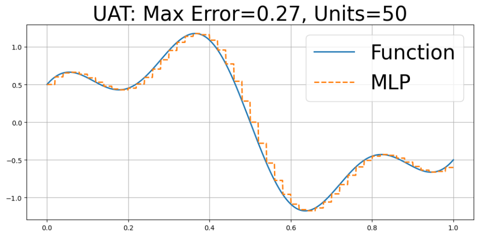
- A MLP can be expressed as a **parameterized** function $f(\boldsymbol{x};\boldsymbol{\theta})$ or $f_{\boldsymbol{\theta}}(\boldsymbol{x})$, where $\boldsymbol{\theta}$ is the collection of all weights $\{\boldsymbol{W}_\ell\}_\ell$ and biases $\{\boldsymbol{b}_\ell\}_\ell$.
- We assume the existence of a **true** function $f^*(\boldsymbol{x}) : \boldsymbol{x} \mapsto y$ maps the input $\boldsymbol{x}$ to the target y.
- The goal of the parameterized function $f_{\boldsymbol{\theta}}$ is to approximate f^* by finding optimal values for $\boldsymbol{\theta}$.

Universal Approximation Theorem (UAT): MLPs $f_{\boldsymbol{\theta}}$ can approximate "any" function f^* with arbitrarily small errors, given sufficient parameters (or neurons).









Universal Approximation Theorem (UAT):

- **Theorem**: MLPs f_θ can approximate "any" function f^* with arbitrarily small errors, given sufficient parameters (or neurons).
- The UAT holds because "any" function on a compact set can be approximated by many **simple local pieces**, and neural networks with nonlinear ϕ can construct **these pieces** and **smoothly combine them** to approximate complex functions.
- **Existence**: the UAT implies the **existence** of suitable parameter values.

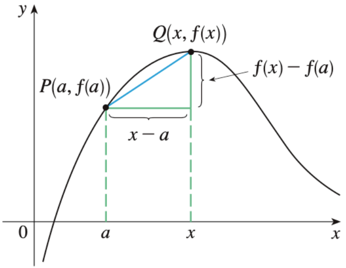
Key Question

How can we find the appropriate values of $\boldsymbol{\theta}$ in practice?

Outline

Definition of Derivative

Definition: Given a real-valued function $f(x)$, the **derivative** of f measures how the output of the function changes with respect to (w.r.t.) changes in the input x.



- If the input changes from a to x, the change in x is $\Delta x = x - a$.
- Consequently, the change in the output is $\Delta y := f(x) - f(a)$.
- The derivative of f at a is the rate of change of f w.r.t. the change of the input:

$$f'(a) \approx \frac{\Delta y}{\Delta x} = \frac{f(x) - f(a)}{x - a}$$

Here, the approximation error is small when x is close to a

Notation: We often denote the derivative of f at x as

$$f'(x) = \frac{df}{dx}, \quad df \approx \Delta y, \quad dx \approx \Delta x,$$

where the approximation is exact in the limit as $\Delta x \to 0$.

James Stewart, "Calculus."

Properties of Derivatives

Here are some fundamental properties of derivatives:

- **Linearity**: The derivative of a linear combination of two functions $h(x) = af(x) + bg(x)$ is:

$$h'(x) = af'(x) + bg'(x)$$

- **Product Rule**: The derivative of the product of two functions $h(x) = f(x)g(x)$ is:

$$h'(x) = f'(x)g(x) + f(x)g'(x)$$

- **Quotient Rule**: The derivative of the quotient of two functions $h(x) = \frac{f(x)}{g(x)}$ (where $g(x) \neq 0$) is:

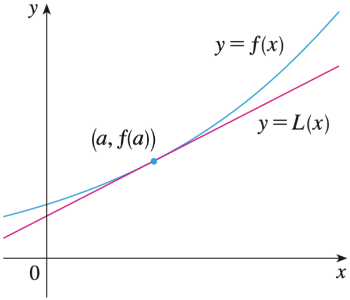
$$h'(x) = \frac{f'(x)g(x) - f(x)g'(x)}{[g(x)]^2}$$

- **Chain Rule**: The derivative of a composition of two functions $h(x) = g(f(x))$ is:

$$h'(x) = g'(f(x)) \cdot f'(x)$$

Linear Approximation

A curve of $f(x)$ lies very close to the **line segment** between the points on the graph. By zooming in toward the point a, the graph looks more and more like its straight line.



- Rewriting the "definition" formula of the derivative, we have:

$$f(x) \approx f(a) + f'(a) \cdot (x - a) := L(x)$$

- Here, $L(x)$ is a **linear** function in x and it is called the **linear approximation of** f **at** a.
- The approximation error decreases as x gets closer to a.
- The function $L(x)$ is the **tangent line** to $f(x)$ at $x = a$.

Multivariate Function and Partial Derivatives

Consider a **multivariate** function $f(x, y)$, where changes in the input can come from either x or y.

- If we fix y and only vary x, we compute the **partial derivative of f w.r.t. x**:

$$\frac{\partial f}{\partial x} \approx \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x} = \frac{\Delta_x f}{\Delta x}$$

  Here, $\Delta_x f$ denotes the change in f caused **only** by changes in x.

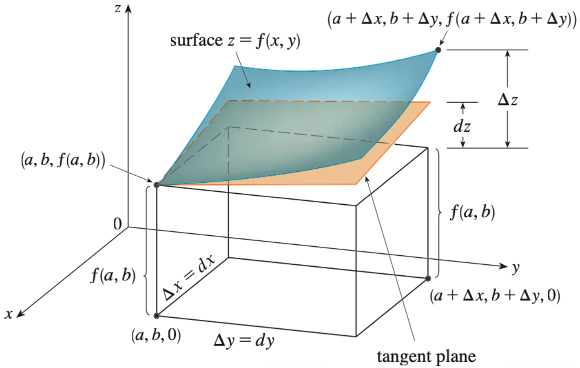
- Similarly, if we fix x and only vary y, we compute the **partial derivative of f w.r.t. y**:

$$\frac{\partial f}{\partial y} \approx \frac{f(x, y + \Delta y) - f(x, y)}{\Delta y} = \frac{\Delta_y f}{\Delta y}$$

  Here, $\Delta_y f$ denotes the change in f caused **only** by changes in y.

Note: Partial derivatives measure how $f(x, y)$ changes w.r.t. one variable while keeping the other variable constant.

Tangent Plane as a Linear Approximation



Similar to a single-variable function $f(x)$, a function $f(x, y)$ has a linear approximation given by:

$$f(x, y) \approx f(a, b) + \frac{\partial f}{\partial x}(a, b) \cdot (x - a) + \frac{\partial f}{\partial y}(a, b) \cdot (y - b) := L(x, y)$$

Here, $L(x, y)$ represents the **tangent plane** to the surface $f(x, y)$ at the point $(a, b, f(a, b))$.

Gradient Vector

Consider a multivariate function $f(\boldsymbol{x}) = f(x_1, \ldots, x_n)$, where $\boldsymbol{x} \in \mathbb{R}^n$.

- **Gradient**: The **gradient** of $f(\boldsymbol{x})$ is a vector of partial derivatives, defined as:

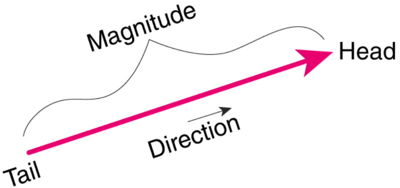
$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}_1} & \cdots & \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}_n} \end{bmatrix}^\top.$$

- **Linear Approximation**: The output change Δf can be approximated by:

$$\Delta f \approx \frac{\partial f}{\partial x_1} \cdot \Delta x_1 + \cdots + \frac{\partial f}{\partial x_n} \cdot \Delta x_n = \nabla f(\boldsymbol{x}) \cdot \Delta \boldsymbol{x},$$

  where the approximation becomes *exact* if $\Delta \boldsymbol{x} \to 0$.

- **Vector Field**: The gradient ∇f is a **vector field** that comprises both **magnitude** and **direction**, where the magnitude is the **Euclidean norm** defined by $\|\boldsymbol{a}\| = \sqrt{\sum_{i=1}^n a_i^2}$.



Steepest Descent Direction

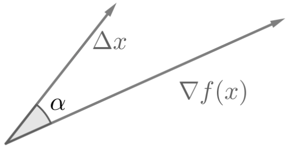
Descent Direction

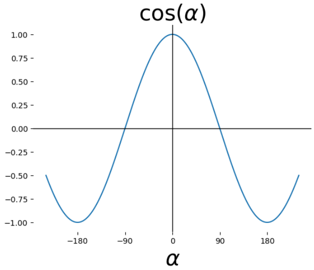
The gradient direction is the steepest **ascent** direction for the function f. Hence, the **negative** gradient is the steepest **descent** direction.

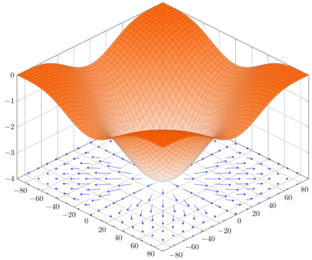
- From the *linear approximation*, we have

$$\Delta f \approx \nabla f(\boldsymbol{x}) \cdot \Delta \boldsymbol{x} = \|\nabla f(\boldsymbol{x})\| \cdot \|\Delta \boldsymbol{x}\| \cdot \cos \alpha$$

where α is the angle between $\nabla f(\boldsymbol{x})$ and $\Delta \boldsymbol{x}$.







- The steepest **ascent** in Δf is obtained when $\alpha = 0$, *i.e.*, $\Delta \boldsymbol{x} \propto \nabla f(\boldsymbol{x})$
- The steepest **descent** in Δf is obtained when $\alpha = \pi$, *i.e.*, $\Delta \boldsymbol{x} \propto -\nabla f(\boldsymbol{x})$.

Summary

- The derivative f' of a function f is the rate of change of the outputs w.r.t. to its input.
- Linearity, product rule, quotient rule, **chain rule**, partial derivatives, gradient
- The output change can be approximated by the inner product of ∇f and Δx, *i.e.*, $\Delta f \approx \nabla f(\boldsymbol{x}) \cdot \Delta \boldsymbol{x}$.
- The **negative** gradient direction is the steepest **descent** direction.

Discussion Questions

Compute the gradients of the following functions:

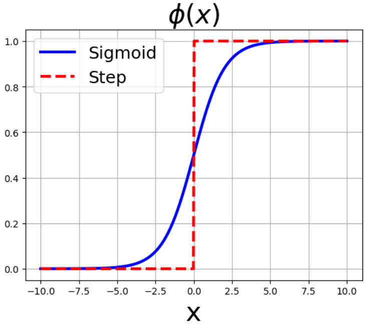
- $f(x) = \frac{1}{2}(x - y)^2$
- $f(x) = \mathbf{1}\{x \geq 0\}$, *i.e.*, the step function: $f(x) = 1$ if $x \geq 0$, and $f(x) = 0$ otherwise
- $f(x) = \frac{1}{1+e^{-x}}$, *i.e.*, sigmoid function. **Hint**: use the chain rule by $z := 1 + e^{-x}$.
- $f(\boldsymbol{x}) = \boldsymbol{a}^\top \boldsymbol{x}$, where $\boldsymbol{a}, \boldsymbol{x} \in \mathbb{R}^n$. **Hint**: write the dot product as summation.

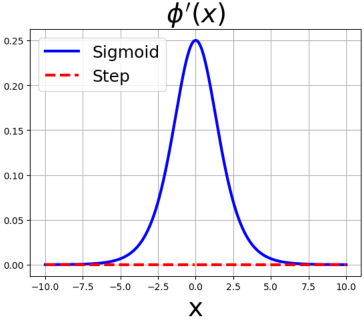
Instructions: Discuss these questions in small groups of 2-3 students.

Solutions to the Discussion Questions

Compute the derivatives of the following functions:

- $f(x) = \frac{1}{2}(x-y)^2$, $f'(x) = x - y$
- $f(x) = \mathbf{1}\{x \geq 0\}$, $f'(x) = 0$ for all x, except $x = 0$ where $f'(x)$ is not defined.
- $f(x) = \frac{1}{1+e^{-x}}$, $f'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = f(x)(1 - f(x))$
- $f(\boldsymbol{x}) = \boldsymbol{a}^\top \boldsymbol{x}$, the partial derivative is $\frac{\partial f}{\partial \boldsymbol{x}_i} = \boldsymbol{a}_i$, and the gradient is $\nabla f(\boldsymbol{x}) = \boldsymbol{a}$.





Zero Derivative

The step function's derivative, $\phi'(x)$, is zero (everywhere except at $x = 0$).

Outline

Introduction to Training Process

For a general machine learning (ML) model including MLPs $f_{\boldsymbol{\theta}}$, it is almost impossible to assign parameter values manually. Instead, we rely on the process called **training**:

- The **training set** is a collection of input-output pairs, *i.e.*, $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$
- A ML model $f_{\boldsymbol{\theta}}$ computes $\hat{y}_i = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$ as an estimate to y_i. Our goal is to find $\boldsymbol{\theta}$ such that

$$\hat{y}_i \approx y_i, \quad \forall i \in [n] := \{1, 2, \cdots, n\},$$

- To measure the divergence between $\hat{y}$ and y, we use a **loss function** $\ell(y, \hat{y})$.
- The objective function or **total cost** is the average of divergence among the training data:

$$\mathcal{L}(\boldsymbol{\theta}) := \frac{1}{n}\sum_{i=1}^{n} \ell(\hat{y}_i, y_i) = \frac{1}{n}\sum_{i=1}^{n} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), y_i)$$

- The training process aims to **iteratively** update the parameters $\boldsymbol{\theta}$ to gradually reduce the cost $\mathcal{L}$.

Loss Function

The choice of loss functions depends on the **learning task**:

- If the output $y \in \mathbb{R}$ is real-valued, the learning problem is called **regression**
- If the output $y \in \{0, 1\}$ is binary value, it is called **(binary) classification** and y is called **label**.
- **Square loss**: as a common loss function in a regression problem, defined

$$\ell(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

- **Cross-entropy loss**: as a broadly used loss function in classification, defined

$$\ell(\hat{y}, y) = -\Big(y \log \hat{y} + (1 - y) \log(1 - \hat{y})\Big),$$

  where $\log(\cdot)$ is the log function, which can be taken with a natural base e or base 10.

Example

Generally, our estimate $\hat{y}$ is not binary value but a positive number between 0 and 1, *e.g.*, $\hat{y} = 0.6$:

- If $y = 1$, then $\ell(\hat{y}, y) = -[1 \cdot \log 0.6 + (1 - 1) \log(1 - 0.6)] = -\log 0.6 \approx 0.22$,
- If $y = 0$, then $\ell(\hat{y}, y) = -[0 \cdot \log 0.6 + (1 - 0) \log(1 - 0.6)] = -\log 0.4 \approx 0.40$,

where we assume base 10.

Gradient Descent

Given an objective function $\mathcal{L}(\boldsymbol{\theta})$, the learning problem of finding $\boldsymbol{\theta}$ to best fit each y_i by $f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$ in the training set is equivalent to solving the following **optimization problem**:

$$\min_{\boldsymbol{\theta}} \quad \mathcal{L}(\boldsymbol{\theta}),$$

which can be interpreted as:

"Minimize the objective function $\mathcal{L}$ with respect to (w.r.t.) the variable $\boldsymbol{\theta}$."

To solve this optimization problem, the **gradient descent** method iteratively updates $\boldsymbol{\theta}$ by moving in **steepest descent direct**. For each iteration $k = 0, 1, 2, \ldots,$ the update rule is:

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^k),$$

where:

- $\boldsymbol{\theta}^k \in \mathbb{R}^p$ is the current value of the parameters, assuming $\boldsymbol{\theta}$ has p components.
- $\boldsymbol{\theta}^{k+1} \in \mathbb{R}^p$ is the updated value.
- $\boldsymbol{\theta}^0 \in \mathbb{R}^p$ is the **initial value** chosen by the practitioner.
- $\eta > 0$ is the **learning rate**, controlling the step size of each update.
- $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$ is the **gradient** of $\mathcal{L}$ w.r.t. $\boldsymbol{\theta}$:

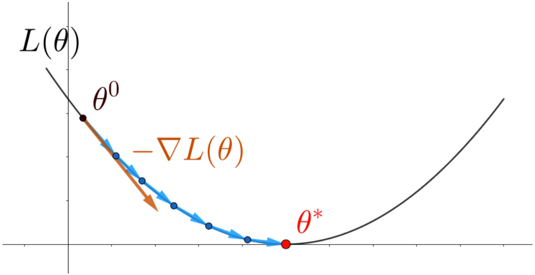
$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} & \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_2} & \cdots & \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_p} \end{bmatrix}^\top$$

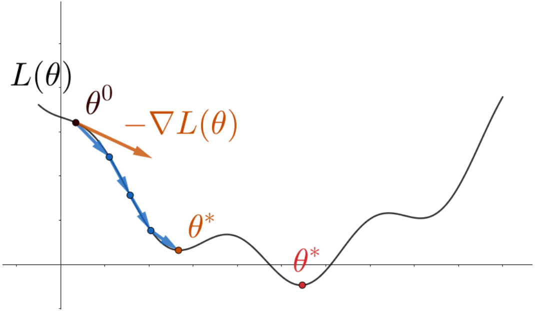
with each $\partial \mathcal{L}(\boldsymbol{\theta}) / \partial \boldsymbol{\theta}_i$ representing the partial derivative of $\mathcal{L}$ w.r.t. $\boldsymbol{\theta}_i$ for all $i \in [p]$.

Gradient Descent Intuition

Gradient Descent:

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} - \eta \nabla \mathcal{L}(\boldsymbol{\theta}^{k}).$$





Warning

Learning rate η and initialization $\boldsymbol{\theta}^0$ are crucial to the performance of gradient descent.

Summary of Gradient Descent

- MLPs are **parameterized** functions $f_{\boldsymbol{\theta}}(\boldsymbol{x})$, where $\boldsymbol{\theta}$ represents the weights and biases.
- Given a **training set**, our goal is to find the optimal $\boldsymbol{\theta}$ that best fits the training samples.
- The divergence between the estimate $\hat{y}_i = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$ and the true value y_i is measured by the **loss function** ℓ.
- The **cost** $\mathcal{L}$ is the average loss over the training samples.
- Finding the optimal $\boldsymbol{\theta}$ is equivalent to solving an **optimization problem** that minimizes the cost $\mathcal{L}$ with respect to $\boldsymbol{\theta}$.
- The **gradient descent** method iteratively updates $\boldsymbol{\theta}$ to reduce the cost $\mathcal{L}$.

Outline

Perceptron

Perceptron

Gradient Computation for Perceptron

- **Perceptron**: Recall $\hat{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ with $\boldsymbol{\theta} = \{\boldsymbol{w}, b\}$ is defined as follows:

$$z = \boldsymbol{w}^\top \boldsymbol{x} + b, \quad a = \phi(z), \quad f_{\boldsymbol{\theta}}(\boldsymbol{x}) = a.$$

- Given a training sample $(\boldsymbol{x}, y)$, with $\hat{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) = a$, the loss is

$$\ell(a, y) = \frac{(\hat{y} - y)^2}{2} = \frac{(f_{\boldsymbol{\theta}}(\boldsymbol{x}) - y)^2}{2} = \frac{(a - y)^2}{2}$$

- Using the **chain rule**, the derivative of loss ℓ w.r.t. to each parameter θ is given by

$$\frac{\partial \ell(a, y)}{\partial \theta} = \frac{\partial \ell(a, y)}{\partial a} \cdot \frac{\partial a}{\partial \theta}$$

Specifically, we have

$$\frac{\partial \ell(a, y)}{\partial \boldsymbol{w}} = \frac{\partial \ell(a, y)}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial \boldsymbol{w}}, \qquad \frac{\partial \ell(a, y)}{\partial b} = \frac{\partial \ell(a, y)}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b},$$

where

$$\frac{\partial \ell(a, y)}{\partial a} = a - y, \qquad \frac{\partial a}{\partial z} = \phi'(z), \qquad \frac{\partial z}{\partial \boldsymbol{w}} = \boldsymbol{x}, \qquad \frac{\partial z}{\partial b} = 1$$

Question: Have you seen any **common terms** involved in the computation?

Computational Graph in Perceptron

Denote $d\theta := \partial\ell(a,y)/\partial\theta$, where θ represents *any* variable involved, *e.g.*, a, z, $\boldsymbol{w}$, and b.

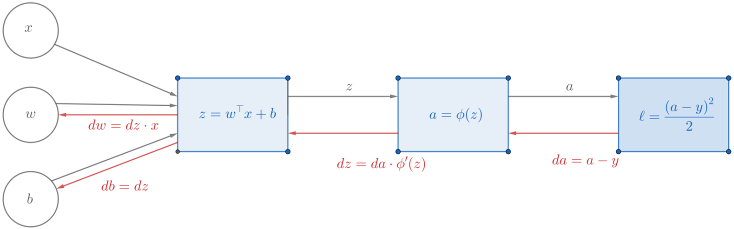
- Rewrite gradient computation using $d\theta$ notation:

$$\frac{\partial \ell(a,y)}{\partial \boldsymbol{w}} = \underbrace{\underbrace{\underbrace{\frac{\partial \ell(a,y)}{\partial a}}_{da} \cdot \frac{\partial a}{\partial z}}_{dz} \cdot \frac{\partial z}{\partial \boldsymbol{w}}}_{d\boldsymbol{w}}, \qquad \frac{\partial \ell(a,y)}{\partial b} = \underbrace{\underbrace{\underbrace{\frac{\partial \ell(a,y)}{\partial a}}_{da} \cdot \frac{\partial a}{\partial z}}_{dz} \cdot \frac{\partial z}{\partial b}}_{db}$$

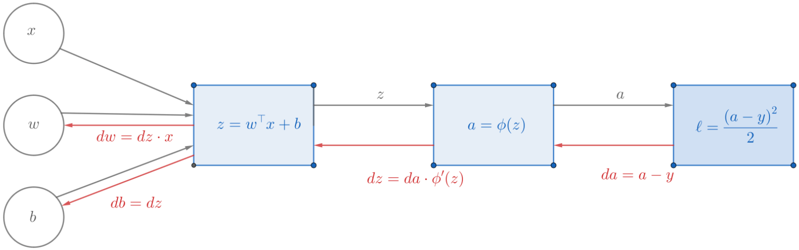
- Using this relation, compute the gradients of the perceptron in a **backward** order:

$$da = a - y, \qquad dz = da \cdot \phi'(z), \qquad d\boldsymbol{w} = dz \cdot \boldsymbol{x}, \qquad db = dz$$

- **Computational graph**:



Information Propagation in Perceptron



Forward propagation to compute the loss:

$$z = \boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b}, \qquad a = \phi(z), \qquad \ell = (a-y)^2/2$$

Backward propagation to compute the gradients:

$$da = a - y, \qquad dz = da \cdot \phi'(z), \qquad d\boldsymbol{w} = dz \cdot \boldsymbol{x}, \qquad db = dz$$

Observations

- For gradient computation, perform one forward-backward pass and **store** intermediate variables.
- By the **chain rule**, break down the gradient computation into **smaller** computational units.
- The same concept applies to MLPs, where **each perceptron** or **layer** acts as a computational unit.

Training Perceptron using Gradient Descent

- **Backward propagation** for gradient computation:

$$da = a - y, \qquad dz = da \cdot \phi'(z), \qquad d\boldsymbol{w} = dz \cdot \boldsymbol{x}, \qquad db = dz$$

- Recall that the cost is given by $\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(a_i, y_i)$.
- Using linearity, the gradient is

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial}{\partial \theta}\left[\frac{1}{n}\sum_{i=1}^{n} \ell(a_i, y_i)\right] = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial \ell(a_i, y_i)}{\partial \theta}$$

  That is the **average** of $d\theta = \partial \ell(a, y)/\partial \theta$ over all training samples.
- The gradient descent update rules for training the perceptron are:

$$\boldsymbol{w}^{+} = \boldsymbol{w} - \frac{\eta}{n}\sum_{i=1}^{n}(a_i - y_i) \cdot \phi'(z_i) \cdot \boldsymbol{x}_i,$$

$$b^{+} = b - \frac{\eta}{n}\sum_{i=1}^{n}(a_i - y_i) \cdot \phi'(z_i).$$

Choice of Activation Function

The sigmoid function is chosen as the activation function, since the step function has a **zero** derivative.

Vectorization for Perceptron

Forward propagation: $z = \boldsymbol{w}^\top \boldsymbol{x} + b \Longrightarrow a = \phi(z) \Longrightarrow \ell = (a-y)^2/2$

Backward propagation: $da = a - y \Longrightarrow dz = da \cdot \phi'(z) \Longrightarrow d\boldsymbol{w} = dz \cdot \boldsymbol{x}$ and $db = dz$

Cost function: $\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}(a_i - y_i)^2$.

- Define data matrix $\boldsymbol{X} \in \mathbb{R}^{n_x \times n}$ and output vector $\boldsymbol{y} \in \mathbb{R}^n$:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_n \end{bmatrix} \quad \text{and} \quad \boldsymbol{y} = \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix}$$

- The pre-activation $\boldsymbol{z}$ can be computed as follows:

$$\boldsymbol{z} = \begin{bmatrix} z_1 & \cdots & z_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}^\top \boldsymbol{x}_1 + b & \cdots & \boldsymbol{w}^\top \boldsymbol{x}_n + b \end{bmatrix} = \boldsymbol{w}^\top \boldsymbol{X} + \begin{bmatrix} b & \cdots & b \end{bmatrix} = \boldsymbol{w}^\top \boldsymbol{X} + b\boldsymbol{e}^\top$$

  where $\boldsymbol{e}$ is a vector whose entries are all ones.
- The forward propagation becomes

$$\boldsymbol{z} = \boldsymbol{w}^\top \boldsymbol{X} + b\boldsymbol{e}^\top, \qquad \boldsymbol{a} = \phi(\boldsymbol{z}), \qquad \mathcal{L} = \frac{1}{2n}\|\boldsymbol{a} - \boldsymbol{y}\|^2$$

- Accordingly, the backpropagation becomes

$$d\boldsymbol{a} = (\boldsymbol{a} - \boldsymbol{y})/n, \qquad d\boldsymbol{z} = d\boldsymbol{a} \odot \phi'(\boldsymbol{z}), \qquad d\boldsymbol{w} = d\boldsymbol{z} \cdot \boldsymbol{X} = \boldsymbol{X} d\boldsymbol{z}, \qquad db = d\boldsymbol{z} \cdot \boldsymbol{e} = \boldsymbol{e}^\top d\boldsymbol{z},$$

  where $\odot$ is the element-wise product.

Pseudocode for Training Perceptron with Square Loss

```
Initialize weights vector w and bias b
Set learning rate eta
Set number of iterations E

For epoch = 1 to E do:
    # Forward Propagation
    z = w.T * X + b * e.T
    a = phi(z)  # Apply activation function element-wise
    L = ||a - y||^2 / (2 * n)  # Compute the cost function

    # Backward Propagation
    da = (a - y)/n  # Derivative of the loss w.r.t. a
    dz = da * phi'(z)  # Derivative of the loss w.r.t. z (element-wise product)
    dw = X * dz  # Derivative of the loss w.r.t. w
    db = sum(dz)  # Derivative of the loss w.r.t. b (sum over all training samples)

    # Gradient Descent Update
    w = w - eta * dw
    b = b - eta * db

End For
```

Multilayer Perceptron

Multilayer Perceptron

Information Propagation in MLP

Let $\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{x}^L$ be an L-layer MLP. Given a training sample $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x} \in \mathbb{R}^{n_x}$ and $\boldsymbol{y} \in \mathbb{R}^{n_y}$:

- **Forward Propagation**: Starting with $\boldsymbol{x}^0 = \boldsymbol{x}$, the output $\hat{\boldsymbol{y}} = \boldsymbol{x}^L$ is computed as:

$$\begin{aligned} \boldsymbol{z}^\ell &= \boldsymbol{W}^\ell \boldsymbol{x}^{\ell-1} + \mathbf{b}^\ell, && \forall \ell \in \{1, 2, \ldots, L\}, \\ \boldsymbol{x}^\ell &= \phi(\boldsymbol{z}^\ell), && \forall \ell \in \{1, 2, \ldots, L\}. \end{aligned}$$

- **Backpropagation**: Given the loss $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{2}\|\hat{\boldsymbol{y}} - \boldsymbol{y}\|^2$, start with $d\boldsymbol{z}^L = (\boldsymbol{x}^L - \boldsymbol{y}) \odot \phi'(\boldsymbol{z}^L)$ and propagate gradients backward:

$$\begin{aligned} d\boldsymbol{z}^\ell &= \left[\boldsymbol{W}^{(\ell+1)\top} d\boldsymbol{z}^{\ell+1}\right] \odot \phi'(\boldsymbol{z}^\ell), && \forall \ell \in \{1, 2, \ldots, L-1\}, \\ d\boldsymbol{W}^\ell &= d\boldsymbol{z}^\ell \boldsymbol{x}^{\ell\top}, && \forall \ell \in \{1, 2, \ldots, L-1\}, \\ d\mathbf{b}^\ell &= d\boldsymbol{z}^\ell, && \forall \ell \in \{1, 2, \ldots, L-1\}. \end{aligned}$$

Derivation of Gradient Descents in MLP

- Using the chain rule, the derivative of loss $\ell(\boldsymbol{x}, \boldsymbol{y})$ w.r.t. $\boldsymbol{W}^\ell$ and $\boldsymbol{b}^\ell$ are given by

$$\frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{b}_i^\ell} = \sum_{\alpha=1}^{m} \frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{z}_\alpha^\ell} \frac{\partial \boldsymbol{z}_\alpha^\ell}{\partial \boldsymbol{b}_i^\ell} = \sum_{\alpha=1}^{m} \frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{z}_\alpha^\ell} \cdot \delta_{\alpha,i} = \frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{z}_i^\ell}$$

$$\frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{W}_{ij}^\ell} = \sum_{\alpha=1}^{m} \frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{z}_\alpha^\ell} \frac{\partial \boldsymbol{z}_\alpha^\ell}{\partial \boldsymbol{W}_{ij}^\ell} = \sum_{\alpha=1}^{m} \frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{z}_\alpha^\ell} \cdot \delta_{\alpha,i} \boldsymbol{x}_j^{\ell-1} = \frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{z}_i^\ell} \boldsymbol{x}_j^{\ell-1}$$

  where $\delta_{i,j} = 1$ if $i = j$ and 0 otherwise.

- Using the $d\theta$ notation, we can put the derivatives in a matrix form:

$$d\boldsymbol{b}^\ell = d\boldsymbol{z}^\ell, \quad \text{and} \quad d\boldsymbol{W}^\ell = d\boldsymbol{z}^\ell \boldsymbol{x}^{\ell\top}$$

- By the computational graph, we can compute $d\boldsymbol{z}^\ell$ backward through a recurrent relation:

$$d\boldsymbol{z}^\ell = \left[\boldsymbol{W}^{(\ell+1)\top} d\boldsymbol{z}^{\ell+1}\right] \odot \phi'(\boldsymbol{z}^\ell),$$

  which is derived from

$$\frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{z}_\alpha^\ell} = \sum_{\beta=1}^{m} \frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{z}_\beta^{\ell+1}} \frac{\partial \boldsymbol{z}_\beta^{\ell+1}}{\partial \boldsymbol{z}_\alpha^\ell} = \sum_{\beta=1}^{m} \frac{\partial \ell(\boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{z}_\beta^{\ell+1}} \boldsymbol{W}_{\beta\alpha}^{\ell+1} \phi'(\boldsymbol{z}_\alpha^\ell), \quad \text{where} \quad \frac{\partial \boldsymbol{z}_\beta^{\ell+1}}{\partial \boldsymbol{z}_\alpha^\ell} = \boldsymbol{W}_{\beta\alpha}^{\ell+1} \phi'(\boldsymbol{z}_\alpha^\ell).$$

Vectorization for MLPs

- Define data matrix $\boldsymbol{X} \in \mathbb{R}^{d_x \times n}$ and target matrix $\boldsymbol{Y} \in \mathbb{R}^{d_y \times n}$:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_n \end{bmatrix}, \qquad \boldsymbol{Y} = \begin{bmatrix} \boldsymbol{y}_1 & \boldsymbol{y}_2 & \cdots & \boldsymbol{y}_n \end{bmatrix}.$$

  With the square loss, the cost function becomes

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{m} \frac{1}{2} \|\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i\|^2 = \frac{1}{2n} \|\hat{\boldsymbol{Y}} - \boldsymbol{Y}\|_F^2,$$

  where $\|\cdot\|_F$ is the Frobenius norm and $\hat{\boldsymbol{y}}_i = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) = \boldsymbol{x}_i^L$.
- With $\boldsymbol{X}^0 = \boldsymbol{X}$ and $\hat{\boldsymbol{Y}} = \boldsymbol{X}^L$, the forward propagation becomes

$$\begin{aligned} \boldsymbol{Z}^\ell &= \boldsymbol{W}^\ell \boldsymbol{X}^{\ell-1} + \boldsymbol{b}^\ell \boldsymbol{e}^\top, && \forall \ell \in [L] \\ \boldsymbol{X}^\ell &= \phi(\boldsymbol{Z}^\ell), && \forall \ell \in [L] \end{aligned}$$

- With $d\boldsymbol{Z}^L = \frac{1}{n}(\boldsymbol{X}^L - \boldsymbol{Y}) \odot \phi'(\boldsymbol{Z}^L)$, the backpropagation is given by

$$\begin{aligned} d\boldsymbol{Z}^\ell &= \phi'(\boldsymbol{Z}^\ell) \odot \left[\boldsymbol{W}^{(\ell+1)\top} d\boldsymbol{Z}^{\ell+1}\right], && \forall \ell \in [L-1] \\ d\boldsymbol{W}^\ell &= d\boldsymbol{Z}^\ell \boldsymbol{X}^{(\ell-1)\top}, && \forall \ell \in [L] \\ d\boldsymbol{b}^\ell &= d\boldsymbol{Z}^\ell \boldsymbol{e}, && \forall \ell \in [L] \end{aligned}$$

Pseudocode: Training an MLP with Gradient Descent

```
Initialize weights W and biases b for all layers
Set learning rate eta and number of epochs E

For epoch = 1 to E do:
    # Forward Propagation
    Set A[0] = X
    For l = 1 to L do:
        Z[l] = W[l] * A[l-1] + b[l] # Linear transformation
        A[l] = phi(A[l]) # Apply activation function

    # Compute the cost function
    C = ||A[L] - Y||^2 / (2 * n) # Square loss between predicted and true output

    # Backward Propagation
    dZ[L] = (A[L]-Y) * \phi'(Z[L]) # Gradient of the loss w.r.t to Z[L]
    dW[L] = dZ[L] * A[L-1] # Gradient of w.r.t. W[L]
    db[L] = sum(dZ[L]) # Gradient of w.r.t. b[L]
    for l = L-1 to 1 do:
        dZ[l] = W[l+1].T * dZ[l+1] * \phi'(Z[l])
        dW[l] = dZ[l] * A[l-1].T # Gradient with respect to W[l]
        db[l] = sum(dZ[l]) # Gradient with respect to b[l]

    # Gradient Descent Update
    for l = 1 to L do:
        W[l] = W[l] - eta * dW[l]
        b[l] = b[l] - eta * db[l]

End For
```

Initialization

Initialization

Problematic Zero Initialization

Forward Propagation (biases omitted): Start with $\boldsymbol{x}^0 = \boldsymbol{x}$

$$\boldsymbol{z}^\ell = \boldsymbol{W}^\ell \boldsymbol{x}^{\ell-1}, \quad \forall \ell \in \{0, 1, 2, \ldots, L\}$$
$$\boldsymbol{x}^\ell = \phi(\boldsymbol{z}^\ell),$$

Backward Propagation (biases omitted): Start with $d\boldsymbol{z}^L = (\boldsymbol{x}^L - \boldsymbol{y}) \odot \phi'(\boldsymbol{z}^L)$

$$d\boldsymbol{z}^\ell = \left[(\boldsymbol{W}^{\ell+1})^\top d\boldsymbol{z}^{\ell+1} \right] \odot \phi'(\boldsymbol{z}^\ell), \quad \forall \ell \in \{1, 2, \ldots, L-1\}$$
$$d\boldsymbol{W}^\ell = d\boldsymbol{z}^\ell \boldsymbol{x}^{(\ell-1)\top}$$

Zero Initialization Issues:

- If $\boldsymbol{W}^\ell = \boldsymbol{0}$, then $\boldsymbol{z}^\ell = \boldsymbol{0}$ and $\boldsymbol{x}^\ell = \phi(\boldsymbol{z}^\ell)$ will have **identical** coordinates across all layers. Since ϕ is applied element-wise, $\phi'(\boldsymbol{z}^\ell)$ and $d\boldsymbol{z}^\ell$ will also have **identical** coordinates. Consequently, $d\boldsymbol{W}^\ell$ will have **identical** rows.
- After one gradient step, $\boldsymbol{W}^\ell$ will contain **identical** rows (and only the last layer is updated), resulting in $\boldsymbol{z}^\ell$ and $\boldsymbol{x}^\ell$ having **identical** coordinates in subsequent iterations.
- This leads to **only one** active neuron per layer, drastically reducing the network's capacity.

Symmetric Activation Patterns

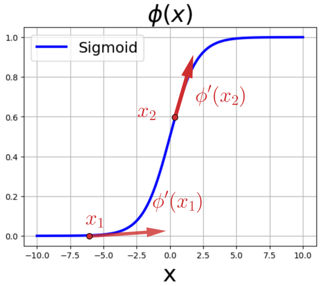
Zero initialization in DNNs results in **symmetric activation patterns** problem in deep learning models.

Random Initialization

To address this problem, we use **random** initialization for the weights. For example, $\boldsymbol{W}_{ij}^{\ell}$ is *i.i.d.* according to a Gaussian distribution with mean zero and variance σ^2:

$$\boldsymbol{W}_{ij}^{\ell} \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, \sigma_\ell^2)$$

- Notably, σ_ℓ is usually a small number to prevent large values in $\boldsymbol{W}^{\ell}$, *e.g.*, $\sigma_\ell = 0.02$. Large weights can cause $\boldsymbol{z}$ to fall into the **flat** regions of the activation function ϕ.



- If so, $\phi'(\boldsymbol{z})$ becomes small, so as small gradients and slowing down training.

Choosing Variance σ_ℓ^2

- Given $\boldsymbol{W}^\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ are independent of $\boldsymbol{x}^{\ell-1}$ and $\mathbb{E}[\boldsymbol{W}_{ij}^\ell] = 0$:

$$\mathbb{E}[\boldsymbol{z}_i^\ell] = n_{\ell-1}\mathbb{E}[\boldsymbol{W}_{ij}^\ell] \cdot \mathbb{E}[\boldsymbol{x}_j^{\ell-1}] = 0.$$

- The variance of $\boldsymbol{z}_i^\ell$ is:

$$\begin{aligned}\mathrm{Var}[\boldsymbol{z}_i^\ell] =& n_{\ell-1}\mathrm{Var}[\boldsymbol{W}_{ij}^\ell] \cdot \mathbb{E}[\boldsymbol{x}_j^{\ell-1}]^2 \\ =& n_{\ell-1}\sigma_\ell^2 \mathbb{E}[\phi(\boldsymbol{z}_j^{\ell-1})]^2 \\ =& n_{\ell-1}\sigma_\ell^2 \mathrm{Var}[\boldsymbol{z}_j^{\ell-1}],\end{aligned}$$

  where we use $\mathrm{Var}[\boldsymbol{W}_{ij}^\ell] = \sigma_\ell^2$ and assume ϕ is linear.
- Recursively applying this relation across layers:

$$\mathrm{Var}[\boldsymbol{z}_i^L] = \left[\prod_{\ell=2}^{L} n_{\ell-1}\sigma_\ell^2\right] \cdot \mathrm{Var}[\boldsymbol{z}_i^1].$$

- To ensure stable propagation (no vanishing or exploding features):

$$n_{\ell-1}\sigma_\ell^2 = 1 \implies \sigma_\ell = \frac{1}{\sqrt{n_{\ell-1}}}.$$

Summary: Neural Network Training

We use a **training process** iteratively update the parameters in MLPs:

- MLPs are **parameterized** function $f_{\boldsymbol{\theta}}$, where $\boldsymbol{\theta} = \{\boldsymbol{W}^{\ell}, \boldsymbol{b}^{\ell}\}$
- Given a **training set** $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{\ell}$ and a **loss** function ℓ, the training problem can be formulated as an optimization problem:

$$\min_{\boldsymbol{\theta}} \quad \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

- This optimization problem can be solved using **gradient descent**, which gradually reduces the cost $\mathcal{L}$ along the *steepest descent direction*:

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} - \eta \nabla \mathcal{L}(\boldsymbol{\theta}^{k})$$

  where $\eta > 0$ is the **learning rate**.
- The gradients in MLPs can be computed using the **chain rule** backward from the total cost.

Summary: Neural Network Training

- Using the **computational graph**, the gradients can be effectively computed through **backpropagation**:

  - Forward Propagation (biases omitted): Start with $\boldsymbol{x}^0 = \boldsymbol{x}$, and compute

$$\boldsymbol{z}^\ell = \boldsymbol{W}^\ell \boldsymbol{x}^{\ell-1}, \qquad \boldsymbol{x}^\ell = \phi(\boldsymbol{z}^\ell).$$

  - Backward Propagation (biases omitted): Start with $d\boldsymbol{z}^L = (\boldsymbol{x}^L - \boldsymbol{y}) \odot \phi'(\boldsymbol{z}^L)$ and calculate

$$d\boldsymbol{z}^\ell = \left[(\boldsymbol{W}^{\ell+1})^\top d\boldsymbol{z}^{\ell+1} \right] \odot \phi'(\boldsymbol{z}^\ell), \qquad d\boldsymbol{W}^\ell = d\boldsymbol{z}^\ell \boldsymbol{x}^{(\ell-1)\top}.$$

- **Random initialization** is preferred over zero initialization to avoid the issue of *symmetric patterns*.

Questions

- What are other common activation functions?
- How do I select the learning rate, width, and depth of the network?
- Does gradient descent always converge? How can I speed up training?
- Does good training performance guarantee good test performance?