

Neural Network Training

Tianxiang (Adam) Gao

Jan 16, 2025

Outline

- 1 Universal Approximation Theorem
- 2 Review of Derivatives
- 3 Optimization and Gradient Descent
- 4 Backpropagation

Outline

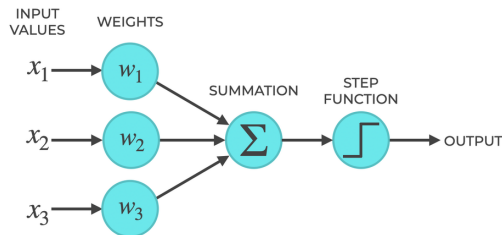
1 Universal Approximation Theorem

2 Review of Derivatives

3 Optimization and Gradient Descent

4 Backpropagation

Recap: Definition of MLPs



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

$$\mathbf{z}^\ell = \mathbf{W}^\ell \mathbf{x}^{\ell-1} + \mathbf{b}^\ell,$$

$$\mathbf{x}^\ell = \phi(\mathbf{z}^\ell),$$

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

Conclusion

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

Universal Approximation Theorem (UAT) of MLPs

- An MLP can be expressed as a **parameterized** function $f(\mathbf{x}; \boldsymbol{\theta})$ or $f_{\boldsymbol{\theta}}(\mathbf{x})$, where $\boldsymbol{\theta}$ is the collection of all weights and biases.
- We assume the existence of a **true** function $f^*(\mathbf{x}) : \mathbf{x} \mapsto y$ maps the input \mathbf{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):

- **Theorem:** MLPs $f_{\boldsymbol{\theta}}$ can approximate “any” function f^* with arbitrarily small errors, given sufficient parameters (or neurons).
- The UAT holds because of the **hierarchical** structure and the **nonlinear** activation function ϕ ,
- **Existence:** the UAT implies the **existence** of suitable parameter values.

Key Question

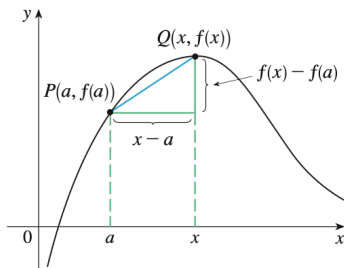
How can we find the appropriate values of $\boldsymbol{\theta}$ in practice?

Outline

- 1 Universal Approximation Theorem
- 2 Review of Derivatives
- 3 Optimization and Gradient Descent
- 4 Backpropagation

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 $\Delta x \rightarrow 0$.

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:

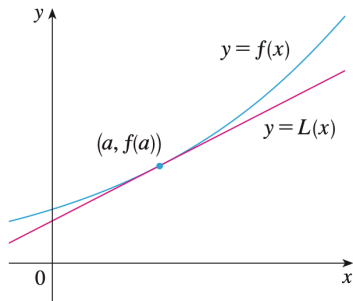
$$h'(x) = \frac{f'(x)g(x) - f(x)g'(x)}{[g(x)]^2}$$

- **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 P and Q . 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 of 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 vary only x , we compute the **partial derivative of f with respect to x** :

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

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

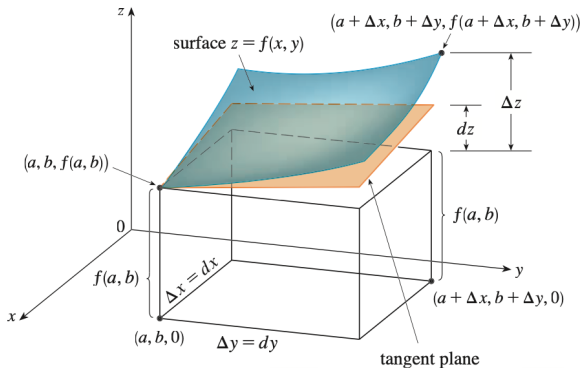
- Similarly, if we **fix** x and vary only y , we compute the **partial derivative of f with respect to y** :

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

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

Note: Partial derivatives measure how $f(x, y)$ changes with respect to 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 $f(\mathbf{x}) = f(x_1, \dots, x_n)$, where $\mathbf{x} \in \mathbb{R}^n$.

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

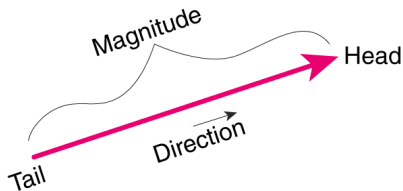
$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1} \quad \dots \quad \frac{\partial f(\mathbf{x})}{\partial x_n} \right]^\top.$$

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

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

where the approximation becomes *exact* if $\Delta \mathbf{x} \rightarrow 0$.

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



Steepest Descent Direction

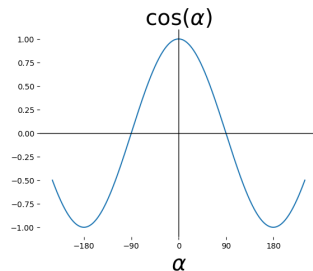
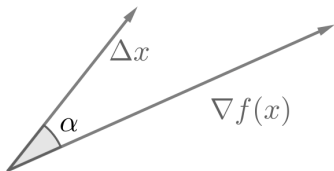
Descent Direction

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

- For simplicity, assume $\|\Delta \mathbf{x}\| = 1$. From the *output change approximation*, we have

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

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



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

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., $\Delta f \approx \nabla f(x) \cdot \Delta x$.
- The **negative** gradient direction is the steepest **descent** direction.

Discussion Questions

Compute the gradients of the following functions:

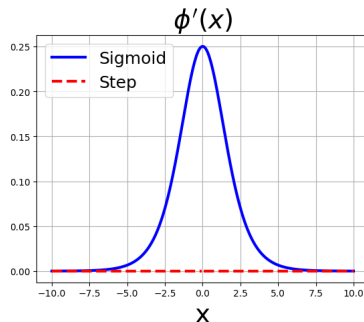
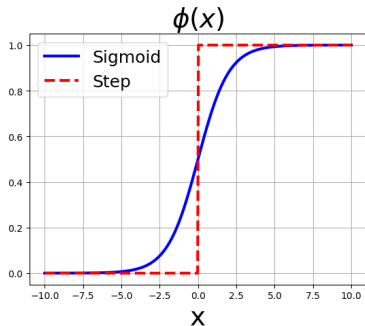
- $f(x) = \frac{1}{2}(x - y)^2$
- $f(x) = \mathbf{1}\{x \geq 0\}$, i.e., the step function: $f(x) = 1$ if $x \geq 0$, and $f(x) = 0$ otherwise
- $f(x) = \frac{1}{1+e^{-x}}$, i.e., sigmoid function. **Hint:** use the chain rule by $z := 1 + e^{-x}$.
- $f(\mathbf{x}) = \mathbf{a}^\top \mathbf{x}$, where $\mathbf{a}, \mathbf{x} \in \mathbb{R}^n$. **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.
- $f(x) = \frac{1}{1+e^{-x}}$, $f'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = f(x)(1 - f(x))$
- $f(\mathbf{x}) = \mathbf{a}^\top \mathbf{x}$, the partial derivative is $\frac{\partial f}{\partial x_i} = \mathbf{a}_i$, and the gradient is $\nabla f(\mathbf{x}) = \mathbf{a}$.



Zero Derivative

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

Outline

- 1 Universal Approximation Theorem
- 2 Review of Derivatives
- 3 Optimization and Gradient Descent
- 4 Backpropagation

Introduction to Training Process

For a general machine learning (ML) model including MLPs f_{θ} , 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.*, $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$
- A ML model f_{θ} computes $\hat{y}_i = f_{\theta}(\mathbf{x}_i)$ as an estimate to y_i . Our goal is to find θ such that

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

- To measure the divergence between \hat{y} and y , we use a **loss function** $\ell : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}_+$.
- The objective or **cost** is the average of divergence among the training data:

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

- The training process aims to **iteratively** update the parameters θ by 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 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

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

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 $\mathcal{L}(\theta)$, the learning problem of finding θ to best fit each y_i by $f_\theta(x_i)$ in the training set is equivalent to solving the following **optimization problem**:

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

which can be interpreted as:

“Minimize the objective function \mathcal{L} with respect to (w.r.t.) the variable θ .”

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

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

where:

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

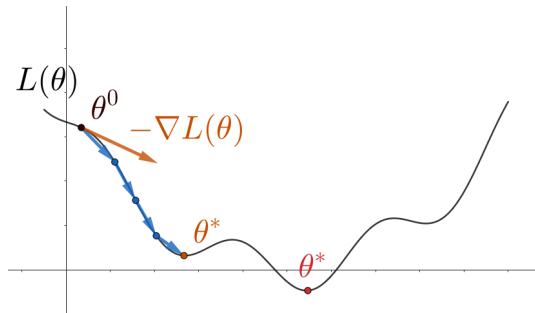
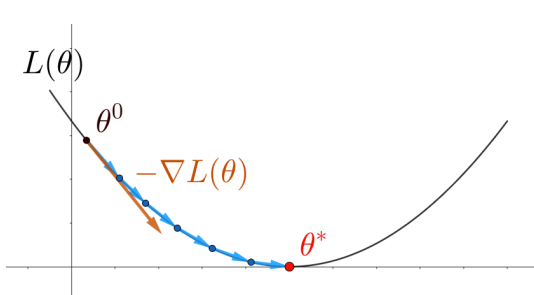
$$\nabla_{\theta} \mathcal{L}(\theta) = \left[\frac{\partial \mathcal{L}(\theta)}{\partial \theta_1} \quad \dots \quad \frac{\partial \mathcal{L}(\theta)}{\partial \theta_p} \right]^T$$

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

Gradient Descent Intuition

Gradient Descent:

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



Warning

Learning rate η and initialization θ^0 are crucial to the performance of gradient descent.

Summary of Gradient Descent

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

Outline

- 1 Universal Approximation Theorem
- 2 Review of Derivatives
- 3 Optimization and Gradient Descent
- 4 Backpropagation

Perceptron

Gradient Computation for Perceptron

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

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

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

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

- 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

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

where

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

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

Computational Graph in Perceptron

Computation:

- Denote $d\theta := \partial\ell(a, y)/\partial\theta$, where θ represents *any* variable involved, e.g., a , z , w , and b .
- Rewrite gradient computation using $d\theta$ notation:

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

$d\mathbf{w}$

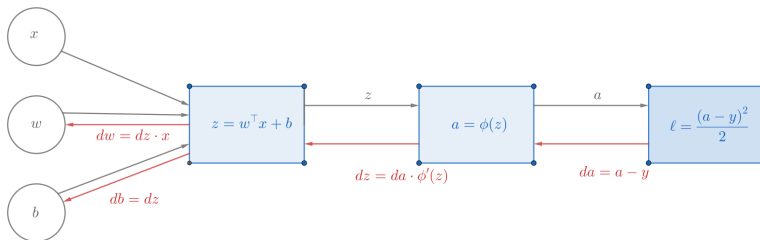
$$\frac{\partial\ell(a, y)}{\partial b} = \underbrace{\frac{\partial\ell(a, y)}{\partial a}}_{da} \cdot \underbrace{\frac{\partial a}{\partial z}}_{dz} \cdot \frac{\partial z}{\partial b}$$

db

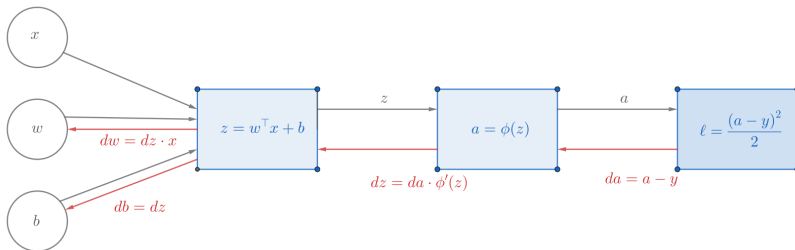
- Using this relation, compute the gradients of the perceptron in a **backward** order:

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

- Computational graph:**



Information Propagation in Perceptron



Forward propagation to compute the loss:

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

Backward propagation to compute the gradients:

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

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:

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

- 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

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

That is the **average** of $d\boldsymbol{\theta} = \partial \ell(a, y) / \partial \boldsymbol{\theta}$ over all training samples.

- The gradient descent update rules for training the perceptron are:

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

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: $z = \mathbf{w}^\top \mathbf{x} + b \implies a = \phi(z) \implies \ell = (a - y)^2 / 2$

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

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

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

$$\mathbf{X} = [\mathbf{x}_1 \quad \mathbf{x}_2 \quad \cdots \quad \mathbf{x}_n] \quad \text{and} \quad \mathbf{y} = [y_1 \quad y_2 \quad \cdots \quad y_n]$$

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

$$\mathbf{z} = [z_1 \quad \cdots \quad z_n] = [\mathbf{w}^\top \mathbf{x}_1 + b \quad \cdots \quad \mathbf{w}^\top \mathbf{x}_n + b] = \mathbf{w}^\top \mathbf{X} + [b \quad \cdots \quad b] = \mathbf{w}^\top \mathbf{X} + b\mathbf{e}^\top$$

where \mathbf{e} is a vector whose entries are all ones.

- The forward propagation becomes

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

- Accordingly, the backpropagation becomes

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

where \odot is the element-wise product.

Pseudocode for Training Perceptron with Square Loss

Initialize weights vector w and bias b

Set learning rate η

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

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

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

$$\mathbf{z}^{\ell} = \mathbf{W}^{\ell} \mathbf{x}^{\ell-1} + \mathbf{b}^{\ell}, \quad \forall \ell \in \{1, 2, \dots, L\},$$

$$\mathbf{x}^{\ell} = \phi(\mathbf{z}^{\ell}), \quad \forall \ell \in \{1, 2, \dots, L\}.$$

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

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

$$d\mathbf{W}^{\ell} = d\mathbf{z}^{\ell} \mathbf{x}^{\ell\top}, \quad \forall \ell \in \{1, 2, \dots, L-1\},$$

$$d\mathbf{b}^{\ell} = d\mathbf{z}^{\ell}, \quad \forall \ell \in \{1, 2, \dots, L-1\}.$$

Derivation of Gradient Descents in MLP

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

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

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

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:

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

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

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

which is derived from

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

Vectorization for MLPs

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

$$\mathbf{X} = [\mathbf{x}_1 \quad \mathbf{x}_2 \quad \cdots \quad \mathbf{x}_n], \quad \mathbf{Y} = [\mathbf{y}_1 \quad \mathbf{y}_2 \quad \cdots \quad \mathbf{y}_n].$$

With the square loss, the cost function becomes

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

where $\|\cdot\|_F$ is the Frobenius norm.

- With $\mathbf{X}^0 = \mathbf{X}$ and $\hat{\mathbf{Y}} = \mathbf{X}^L$, the forward propagation becomes

$$\mathbf{Z}^\ell = \mathbf{W}^\ell \mathbf{X}^{\ell-1} + \mathbf{b}^\ell \mathbf{e}^\top, \quad \forall \ell \in [L]$$

$$\mathbf{X}^\ell = \phi(\mathbf{Z}^\ell), \quad \forall \ell \in [L]$$

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

$$d\mathbf{Z}^\ell = \phi'(\mathbf{Z}^\ell) \odot [\mathbf{W}^{(\ell+1)\top} d\mathbf{Z}^{\ell+1}], \quad \forall \ell \in [L-1]$$

$$d\mathbf{W}^\ell = d\mathbf{Z}^\ell \mathbf{X}^{(\ell-1)\top}, \quad \forall \ell \in [L]$$

$$d\mathbf{b}^\ell = d\mathbf{Z}^\ell \mathbf{e}, \quad \forall \ell \in [L]$$

Pseudocode: Training an MLP with Gradient Descent

```
1 Initialize weights W and biases b for all layers
2 Set learning rate eta and number of epochs E
3
4 For epoch = 1 to E do:
5     # Forward Propagation
6     Set A[0] = X
7     For l = 1 to L do:
8         Z[l] = W[l] * A[l-1] + b[l] # Linear transformation
9         A[l] = phi(A[l]) # Apply activation function
10
11     # Compute the cost function
12     C = ||A[L] - Y||^2 / (2 * n) # Square loss between predicted and true output
13
14     # Backward Propagation
15     dZ[L] = (A[L]-Y) * \phi'(Z[L]) # Gradient of the loss w.r.t to Z[L]
16     dW[L] = dZ[L] * A[L-1] # Gradient of w.r.t. W[L]
17     db[L] = sum(dZ[L]) # Gradient of w.r.t. b[L]
18     for l = L-1 to 1 do:
19         dZ[l] = W[l+1].T * dZ[l+1] * \phi'(Z[l])
20         dW[l] = dZ[l] * A[l-1].T # Gradient with respect to W[l]
21         db[l] = sum(dZ[l]) # Gradient with respect to b[l]
22
23     # Gradient Descent Update
24     for l = 1 to L do:
25         W[l] = W[l] - eta * dW[l]
26         b[l] = b[l] - eta * db[l]
27
28 End For
```

Initialization

Initialization

Problematic Zero Initialization

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

$$\mathbf{z}^\ell = \mathbf{W}^\ell \mathbf{x}^{\ell-1}, \quad \forall \ell \in \{0, 1, 2, \dots, L\}$$

$$\mathbf{x}^\ell = \phi(\mathbf{z}^\ell),$$

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

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

$$d\mathbf{W}^\ell = d\mathbf{z}^\ell \mathbf{x}^{(\ell-1)\top}$$

Zero Initialization Issues:

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

Symmetric Activation Patterns

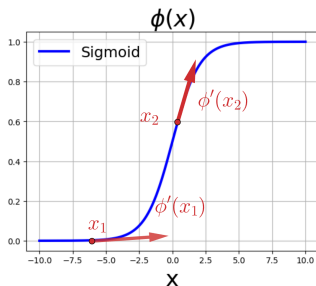
Zero initialization in DNNs results in **symmetric activation patterns** problem in deep learning models.

Random Initialization

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

$$W_{ij}^\ell \stackrel{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$$

- Notably, σ is usually a small number to prevent large values in W^ℓ . Large weights can cause z to fall into the **flat** regions of the activation function ϕ .



- If so, $\phi'(z)$ becomes small, so as small gradients and slowing down training.
- A common choice for σ is proportional to $1/\sqrt{m}$, where m is the input dimension of current layer.
- This choice helps maintain the scale of gradients as they propagate through the network.

Summary: Neural Network Training

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

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

$$\min_{\theta} \mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f_{\theta}(x_i), y_i)$$

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

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

where $\eta > 0$ is the **learning rate**.

- The gradients in MLPs can be computed using the **chain rule** backward from the total cost.

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

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

$$\mathbf{z}^\ell = \mathbf{W}^\ell \mathbf{x}^{\ell-1}, \quad \forall \ell \in \{0, 1, 2, \dots, L\}$$

$$\mathbf{x}^\ell = \phi(\mathbf{z}^\ell),$$

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

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

$$d\mathbf{W}^\ell = d\mathbf{z}^\ell \mathbf{x}^{(\ell-1)\top}$$

- **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?