Neural Networks I

Building Blocks

Overview

- Part I: Building blocks (today!)
- Part II: Convolutional Neural networks, transformer (Monday, July 21)
- Part III: Neural networks in unsupervised learning and reinforcement learning (Monday July 21)

History

Year	Milestone	What changed?
1943	McCulloch–Pitts neuron	Binary logic with weighted sums
1958	Rosenblatt Perceptron	First learning rule for a single neuron
1986	Rumelhart–Hinton–Williams	Back-propagation revives multilayer nets
2006	Hinton's deep belief nets	Unsupervised pre-training enables depth
2012	AlexNet wins ImageNet	GPUs + ReLU ignite the deep-learning boom
2018-25	Foundation models	GPT, AlphaFold, Claude, Gemini ...

Applications in 2025

And many other applications in almost every field

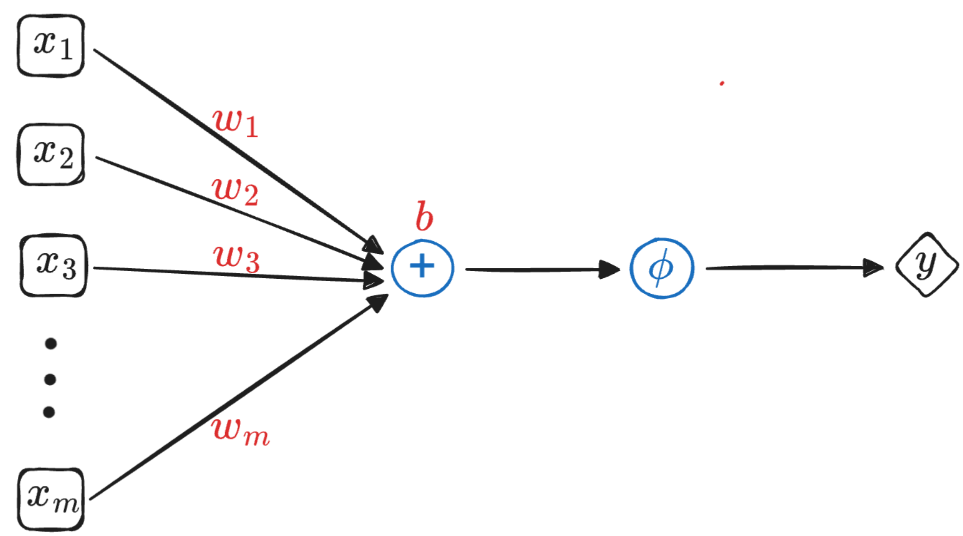
- Augmented Reality (AR glasses)
- AI for science/math (alphaFold)
- Robots (Boston dynamics, Unitree)
- ...

Perceptron (McCulloch–Pitts)

Formulation:

$$y = \phi\left(w_1 x_1 + w_2 x_2 + \ldots + w_m x_m + b\right)$$

- Motivated by biological neurons
- Key words: weights w_j, offset b, activation function $\phi(\cdot)$



Link to illustration

Try out different perceptrons in Tensorflow playground

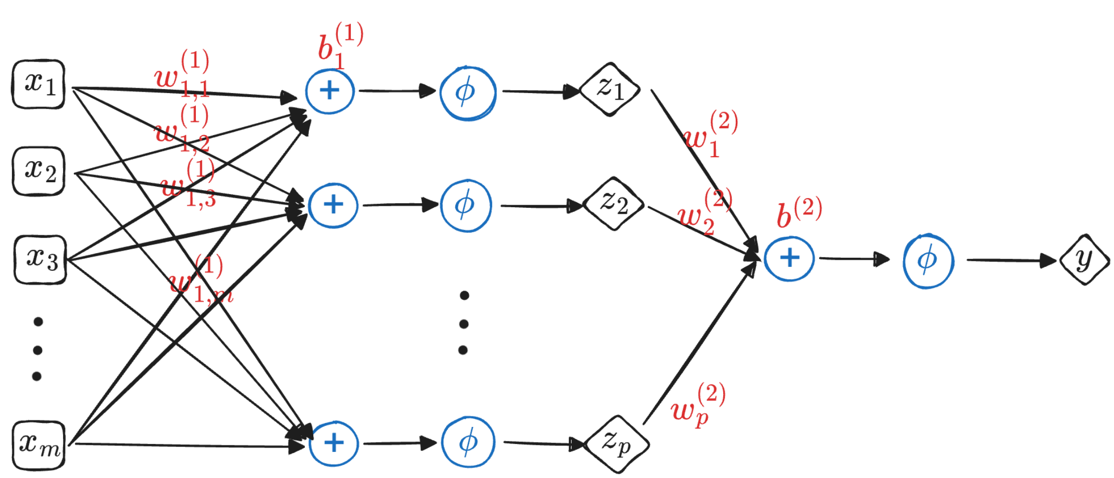
Multi-Layer Perceptron (MLP)

Multi-layer perceptron, or feed-forward neural network, is a chain of linear layers plus nonlinear activations, for hidden layer $l = 1, 2, \ldots, L$,

$$\mathbf{z}^{(l)} = \phi\left(\mathbf{W}^{(l)}\,\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}\right),$$

where $\mathbf{z}^{(0)} = \mathbf{x}$ and $\mathbf{z} \in \mathbb{R}^p$.

- Depth: L, number of hidden layers
- Width: p, number of neurons per layer



Link to illustration

MLP as Composition of Functions

If we write $f(\mathbf{z}^{(l-1)}, \mathbf{W}^{(l)}) = \phi\left(\mathbf{W}^{(l)}\,\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}\right)$ (dropping the offset $\mathbf{b}^{(l)}$ for ease of bookkeeping), we can see that

$$\mathbf{z}^{(l)} = f(\mathbf{z}^{(l-1)}, \mathbf{W}^{(l)}) = f(f(\mathbf{z}^{(l-2)}, \mathbf{W}^{(l-1)}), \mathbf{W}^{(l)}) = \ldots$$

Here $f(g(x))$ is the composition of two functions f and g:

$$\text{Let} \quad u = g(x) \quad \text{then} \quad f(g(x)) = f(u)$$

Try out different perceptrons with **linear** activation in Tensorflow playground

Linearity + Non-Linearity

Stacking linear layers without nonlinear activation is not helpful!

Observation: composition of **linear** mappings is still a **linear** mapping.

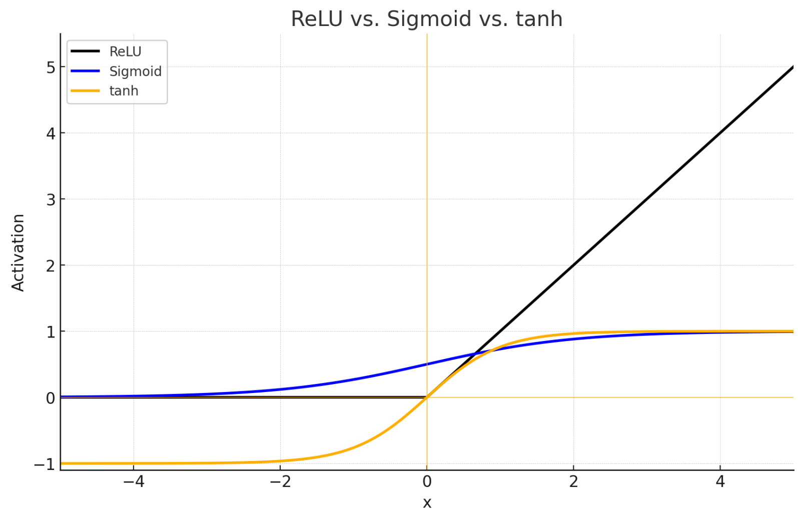
$$\mathbf{W}^{(2)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)} = \mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x} + \mathbf{W}^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)} = \mathbf{W}'\mathbf{x} + \mathbf{b}',$$

where $\mathbf{W}' \equiv \mathbf{W}^{(2)}\mathbf{W}^{(1)}$ and $\mathbf{b}' \equiv \mathbf{W}^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)}$.

Common Non-Linear Activation Functions

Here are the three most common activation functions:

- **ReLU**: rectified linear unit $\max(0, z)$
- Sigmoid: $1/(1+e^{-z})$
- Tanh: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$



Try out different perceptrons with **nonlinear** activation in Tensorflow playground

MLP as a Universal Approximator

Fact: A **single hidden layer** MLP with enough neurons can approximate *any* continuous function on a compact set (Cybenko 1989).

Practical take-aways:

- Depth is **not** required for approximation.
- Deep, *narrow* nets are far more parameter-efficient.
- Deep hierarchies capture **compositional structure** (i.e., features build on features).

Training MLPs: Loss Function

Given inputs $\mathbf{x}$ and label y, we want to find a loss function to evaluate the model

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_i \ell\big(f(\mathbf{x}_i;\theta),\, y_i\big)$$

Here we use θ to represent the collection of $\mathbf{W}^{(1)}$, $\mathbf{b}^{(1)}$, $\mathbf{W}^{(2)}$, $\mathbf{b}^{(2)}$, ...

- Common choices: MSE for regression, cross-entropy for classification.

- Regularizers: we might add ℓ_2-norm as penalty on θ, use dropout in training...

Training MLPs: Optimization

Given the loss function $\mathcal{L}(\theta)$, we want to find $\hat{\theta}$ that minimizes the loss

$$\hat{\theta} = \arg\min_{\theta} \mathcal{L}(\theta).$$

This type of problem is known as an *optimization* problem

Common algorithms:

- **Gradient descent**, nesterov accelerated gradient, adaptive Gradient Algorithm, adaptive Moment Estimation, root Mean Square Propagation, ...

Optimization in Modern Deep Learning

Model (year)	Training cost	Number of parameters
GPT-5 (2025, in progress)	"500 million" (1)	5 trillion
GPT-4 (2023)	“More than 100 million” (2)	1.8 trillion
Gemini Ultra (Google, 2023)	"Close to 200 million" (3)	1.56 trillion
Claude 3.7 Sonnet (Anthropic, 2025)	“A few tens of millions” (3)	70 - 150 billion
Llama 3 (Meta, 2024)	“At least 500 million" (4)	405 billion
Human brain (300,000 BCE)	???	86 billion (neurons)

Gradient Descent

Iterate till convergence

$$\theta^{(k)} \leftarrow \theta^{(k-1)} - \eta \nabla_{\theta} \mathcal{L}|_{\theta=\theta^{(k-1)}}$$

- η: learning-rate, step size.
- $\nabla_{\theta}\mathcal{L}$: gradient* of $\mathcal{L}$ with respect to θ
- $\nabla_{\theta}\mathcal{L}|_{\theta=\theta^{(k-1)}}$: ... evaluated at $\theta^{(k-1)}$
- Convergence: $||\theta^{(k)} - \theta^{(k-1)}||_2 \leq \epsilon$ (ϵ is a user-specified threshold)

*:Roughly speaking, a ratio of the following form [the change in $\mathcal{L}$ given an incremental change in θ] / [the incremental change in θ]

Gradient Descent: Examples

Calculate the first three gradient updates by hand:

$$\hat{\theta} = \arg\min_{\theta} (y - x\theta)^2,$$

where $y = 10$ and $x = 2$.

- $\nabla_\theta \mathcal{L} = 2x(x\theta - y)$
- $\theta^{(0)} = 6$
- $\eta = 0.01$

We should be able to work out the solution easily. It might be even easier without using GD!

> Additional not-so-easy examples if time permits

Stochastic Gradient Descent

Recall that $\mathcal{L}(\theta) = \frac{1}{N}\sum_i \ell\big(f(\mathbf{x}_i;\theta),\, y_i\big)$

Each update of gradient descent has to go through all N observations -- might be too costly for big data!

Stochastic Gradient Descent: calculate gradients on mini-batches

Iterate till convergence

- Randomly draw n observations with indices $\{j_1, j_2, \ldots, j_n\}$
- Update $\theta^{(k)}$

$$\theta^{(k)} \leftarrow \theta^{(k-1)} - \frac{\eta}{n}\sum_{i=1}^{n} \nabla_\theta \ell\big(f(\mathbf{x}_{j_i};\theta),\, y_{j_i}\big)$$

> SGD exercise if time permits.

Backpropogation

- In the exercise, we have already found the challenges in deriving gradients (by-hand)
- For deep neural networks with many hidden layers, it is infeasible to derive the gradients by hand
- A better idea is to use the chain-rule in calculous, let $z = g(\theta)$

$$\frac{\partial f(g(\theta))}{\partial \theta} = \frac{\partial f}{\partial z}\frac{\partial g}{\partial \theta}$$

- Backpropogation helps break down the task of finding gradients of complicated composite functions into manageable, commonly-used gradients.
- More examples in Lecture 4 of Stanford CS231n by Fei-Fei Li et al. (starting from Page 57)

Training MLPs

- Training MLPs are fairly straightforward thanks to existing librarys
- Popular libraries: TensorFlow, PyTorch
- Play with an example using California Housing Data
