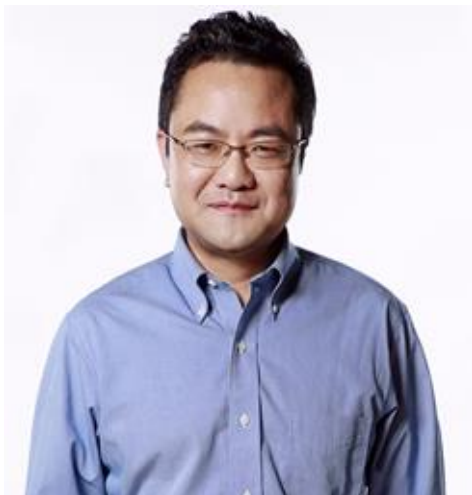


Deep Learning Methods in Advanced Statistical Problems

JSM 2026 Short Course

Agenda

- ❑ Major Architectures of Deep Learning Models (Hongtu)
- ❑ Provable Deep Generative Models (Xiao)
- ❑ Large Language Models and its Applications in Statistics (Runpeng)



Course Websites

<https://bios740.github.io/>

BIOS740



Syllabus

Fundamentals of Deep Learning

Lec1: [Introduction to Deep Learning, PyTorch & Basic Algorithms](#)

Lec2: [Neural Networks Fundamentals](#)

Basic Network Structures

Lec3: [Convolutional Neural Networks \(CNN\)](#)

HW 1

Lec4: [Recurrent Neural Networks \(RNN\) and Long Short-Term Memory \(LSTM\)](#)

HW 2

HW 1 DUE

Lec5: [Graph Neural Networks: GNN, GCN](#)

HW 3

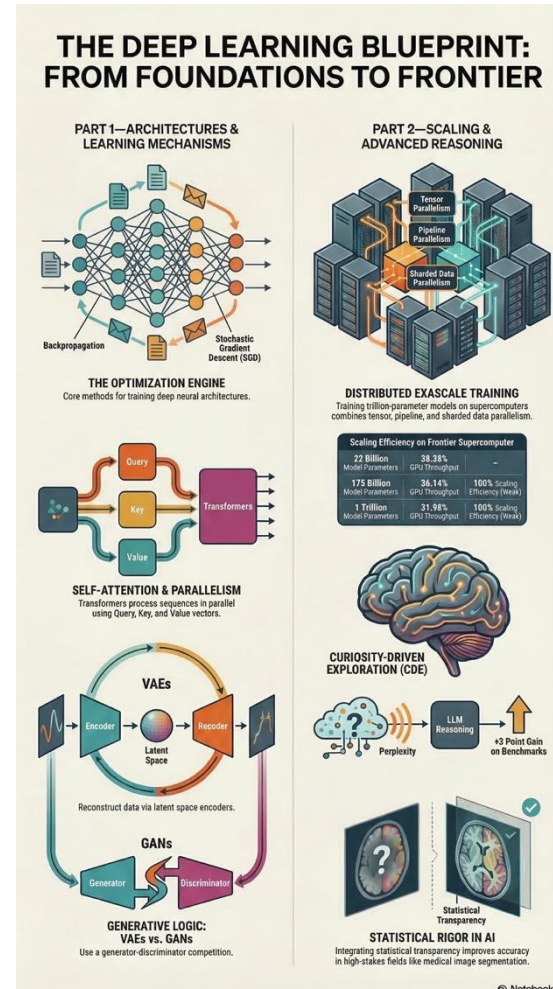
HW 2 DUE

Lec6: [Generative Adversarial Networks \(GAN\)](#)

HW 4

HW 3 DUE

<https://bios740.github.io/short/JSM2026>



© NotebookLM

Major Architectures of Deep Learning Models



Dr. Hongtu Zhu
Kenan Distinguished Professor
University of North Carolina at Chapel Hill
URL: www.med.unc.edu/bigs2/

Content

1 Introduction to Deep Learning

2 Neural Network Basics

3 Tabular Data and Fully Connected Neural Network (FCNN)

4 Loss Functions and Optimization Techniques

5 Computer Vision and Convolutional Neural Networks (CNN)

6 Natural Language Processing, Recurrent Neural Networks (RNN) and Transformer

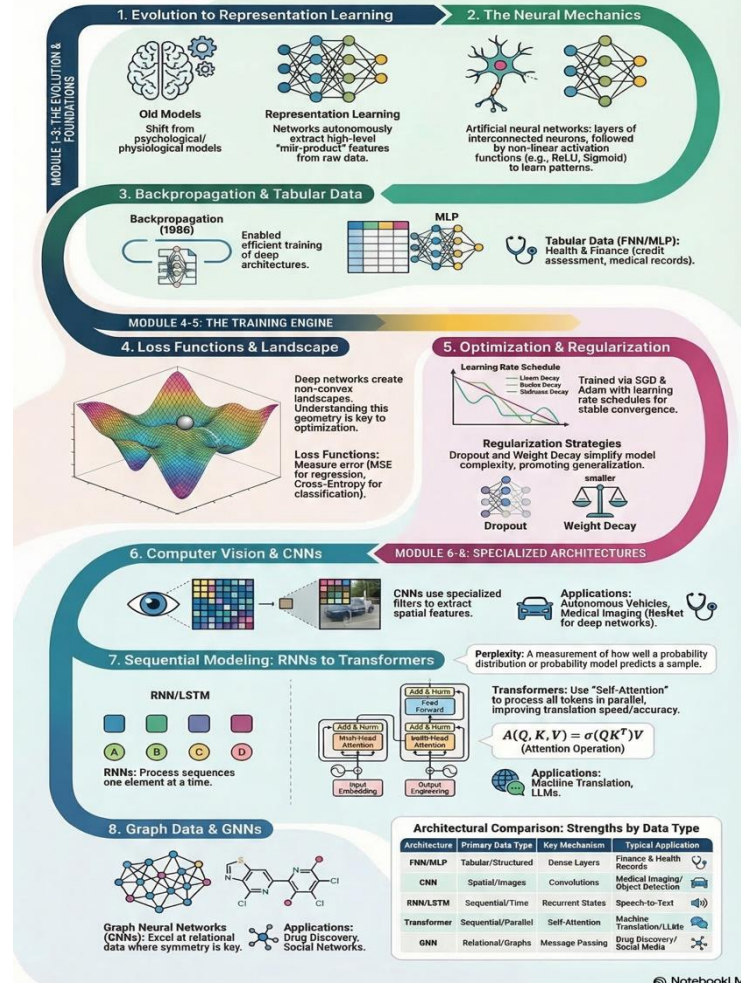
7 Graph Data and Graph Neural Networks (GNNs/GCNs)

8 Statistical Generative Models

Content

1 Introduction to Deep Learning

The Deep Learning Roadmap: From Foundations to Advanced Architectures



NotebookLM

Deep Learning

Deep

Many hidden layers

Learning

Supervised, semi-supervised, unsupervised
Learn adaptive parameters

- Use a cascade of multiple layers of nonlinear processing units for feature extract and transformation
- Learn in supervised and/or unsupervised manner
- Learn representations in different level of abstraction

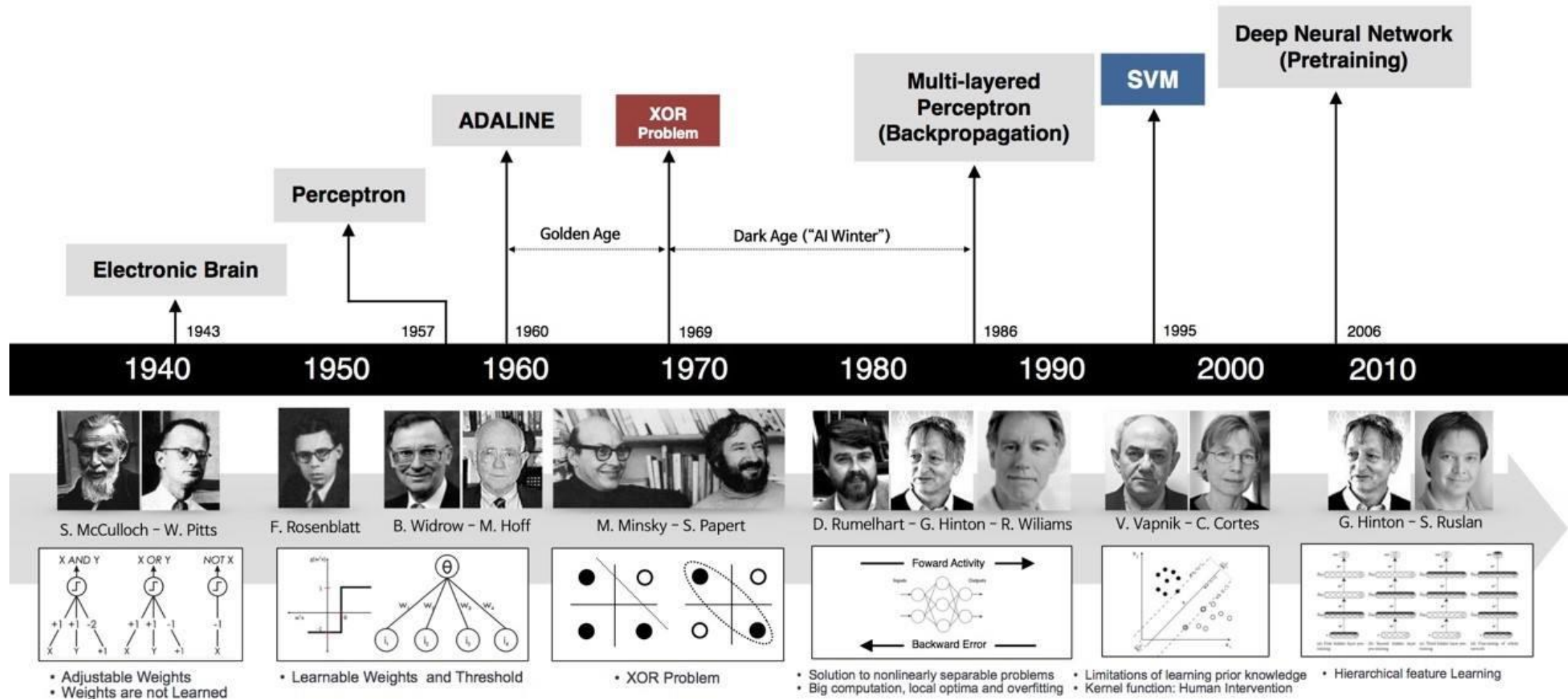
Why popular?

- Chip processing ability
- Increased size of data for training
- Advances in machine learning and signal/information researches



Deep models to efficiently exploit **complex, compositional nonlinear** functions to learn **distributed and hierarchical feature representations**, to make best use

Historical Summary



Deep Learning Explosion



Feature

THE MOST-CITED PAPERS OF THE TWENTY-FIRST CENTURY

An exclusive *Nature* analysis reveals the 25 highest-cited papers published this century and explores why they are breaking records. By Helen Pearson, Heidi Ledford, Matthew Hutson and Richard Van Noorden



1	Title
2	
3	Deep Residual Learning for Image Recognition
4	Analysis of Relative Gene Expression Data Using Real-Time Quantitative PCR and the 2-ΔΔCT Method
5	Using thematic analysis in psychology
6	Diagnostic and Statistical Manual of Mental Disorders, DSM-5
7	A short history of SHELX
8	Random Forests
9	Attention is all you need
10	ImageNet classification with deep convolutional neural networks
11	Global Cancer Statistics 2020: GLOBOCAN Estimates of Incidence and Mortality Worldwide for 20 GLOBOCAN 2020
12	Global cancer statistics 2018: GLOBOCAN estimates of incidence and mortality worldwide for 36
13	Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement
14	U-Net: Convolutional Networks for Biomedical Image Segmentation
15	Electric Field Effect in Atomically Thin Carbon Films
16	Fitting Linear Mixed-Effects Models Using lme4
17	Scikit-learn: Machine learning in Python
18	Deep learning
19	Common Method Biases in Behavioral Research: A Critical Review of the Literature and Recommendations for
20	Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2

Deep Learning Platforms

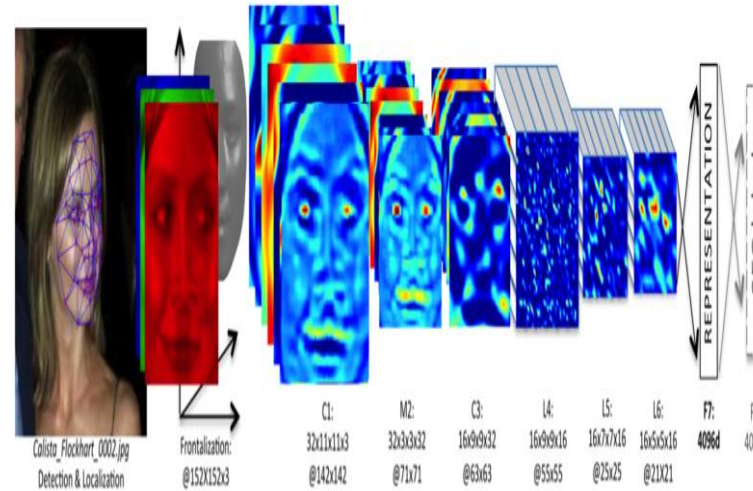


theano



Applications - Vision

IMAGENET

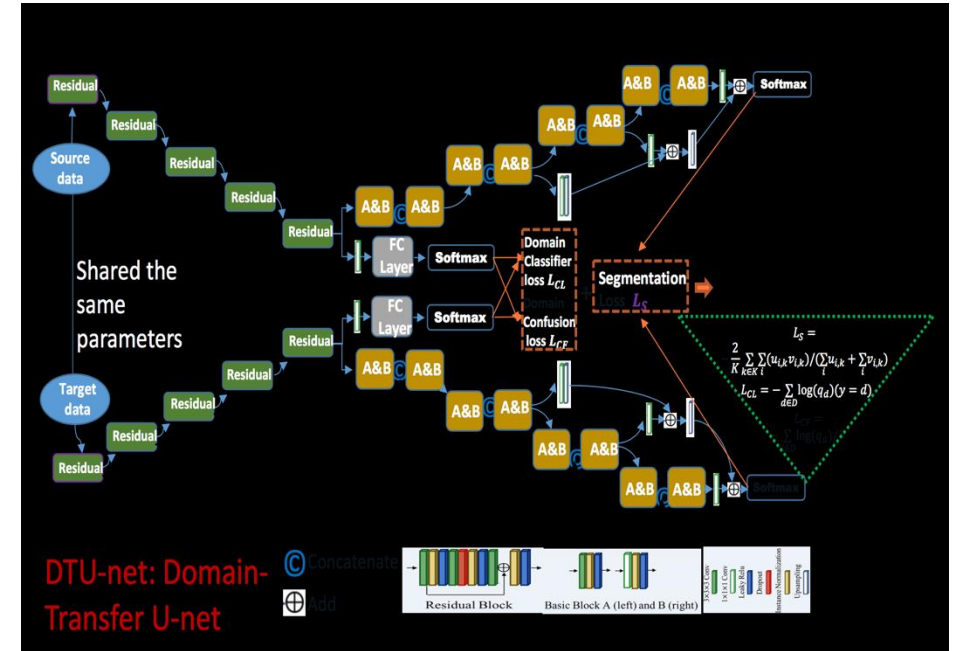


Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	##ing	[SEP]
Token Embeddings	$E_{[CLS]}$	E_{my}	$E_{[MASK]}$	E_{is}	E_{cute}	$E_{[SEP]}$	E_{he}	$E_{[MASK]}$	E_{play}	$E_{##ing}$	$E_{[SEP]}$
Sentence Embedding	+	+	+	+	+	+	+	+	+	+	+
Transformer Positional Embedding	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}

Applications – Medical Imaging

Segmentation Annotation

U-Nets



Liu, Q., Xu, Z., Bertasius, G., & Niethammer, M. (2023). SimpleClick: Interactive Image Segmentation with Simple Vision Transformers. ICCV, 22290-22300. 2023.

Azad *et al.*, “Medical Image Segmentation Review: The success of U-Net.” arXiv, Nov. 27, 2022.

Minaee, Shervin, et al. "Image segmentation using deep learning: A survey." *IEEE PAMI* 44.7 (2021): 3523-3542

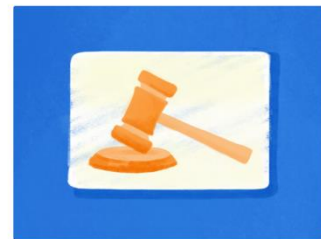
Application - Language

Language Translation in Natural Language Processing

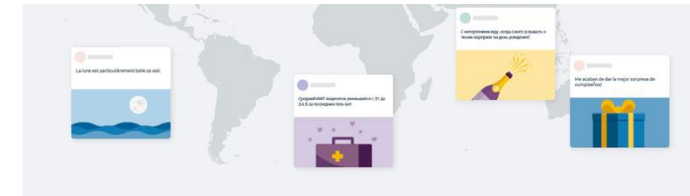
Deep learning enhances real-time, accurate translation of languages, as seen in tools like Google Translate. The following picture shows the translation of a webpage from English to Chinese.



- Facebook AI is introducing M2M-100, the first multilingual machine translation (MMT) model that can translate between any pair of 100 languages without relying on English data. It's open sourced [here](#).
- When translating, say, Chinese to French, most English-centric multilingual models train on Chinese to English and English to French, because English training data is the most widely available. Our model directly trains on Chinese to French data to better preserve meaning. It outperforms English-centric systems by 10 points on the widely used BLEU metric for evaluating machine translations.
- M2M-100 is trained on a total of 2,200 language directions — or 10x more than previous best, English-centric multilingual models. Deploying M2M-100 will improve the quality of translations for billions of people, especially those that speak low-resource languages.
- This milestone is a culmination of years of Facebook AI's foundational work in machine translation. Today, we're sharing details on how we built a more diverse MMT training data set and model for 100 languages. We're also [releasing the model, training, and evaluation setup](#) to help other researchers reproduce and further advance multilingual models.



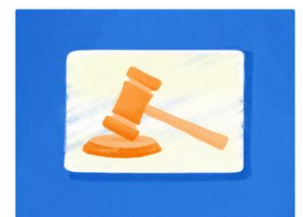
Meta
Meta and Christian Louboutin File
Joint Lawsuit Against Counterfeiter
November 16, 2023



- Facebook AI 正在推出 M2M-100，这是第一个多语言机器翻译 (MMT) 模型，可以在 100 种语言中的任意对之间进行翻译，而无需依赖英语数据。[这里是开源的](#)。
- 例如，在将中文翻译成法语时，大多数以英语为中心的多语言模型都会在中文到英语和英语到法语上进行训练，因为英语训练数据是最广泛可用的。我们的模型直接对中文到法语的数据进行训练，以更好地保留含义。在广泛使用的用于评估机器翻译的 BLEU 指标上，它比以英语为中心的系统高出 10 个点。
- M2M-100 接受了总共 2,200 种语言方向的训练，比以前最好的、以英语为中心的多语言模型多了 10 倍。部署 M2M-100 将为数十亿人提高翻译质量，尤其是那些使用资源匮乏语言的人。
- 这一里程碑是 Facebook AI 多年来在机器翻译领域基础工作的结晶。今天，我们将分享如何为 100 种语言构建更加多样化的 MMT 训练数据集和模型的详细信息。我们还发布了[模型、训练和评估设置](#)，以帮助其他研究人员重现和进一步推进多语言模型。



元
支持独立研究的新工具
2023 年 11 月 21 日



元
Meta 和 Christian Louboutin 对仿冒者提

Application - Language

Large language models

Large language models can perform various tasks such as answering questions, generating creative content, summarizing text, translating languages, and engaging in conversations. It's designed to understand and generate text in a coherent and contextually relevant manner.



<https://www.nature.com/articles/d41586-025-03015-6>

Application - Decision

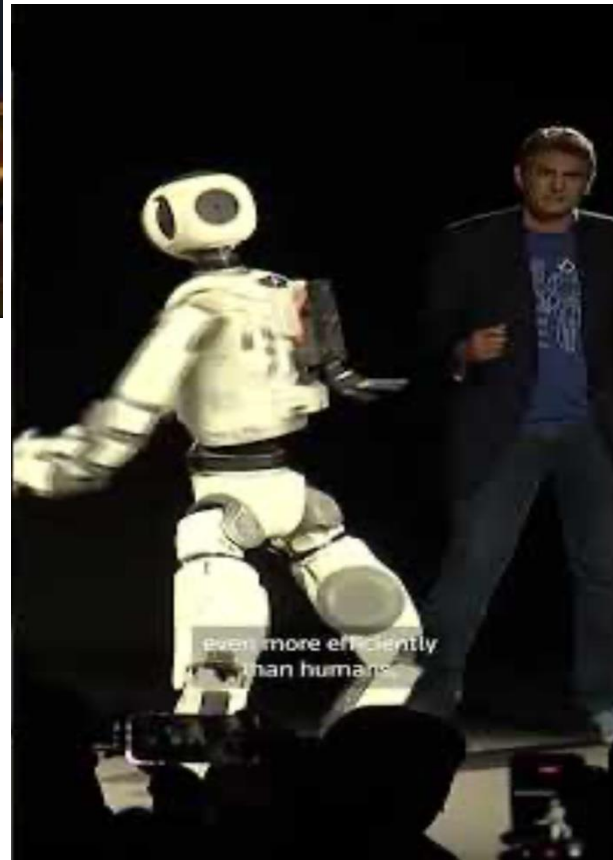


March 2016, AlphaGo made headlines by defeating Lee Sedol.



Reinforcement learning methods have shown priority in video games.

Application – Autonomous Driving & Robotics

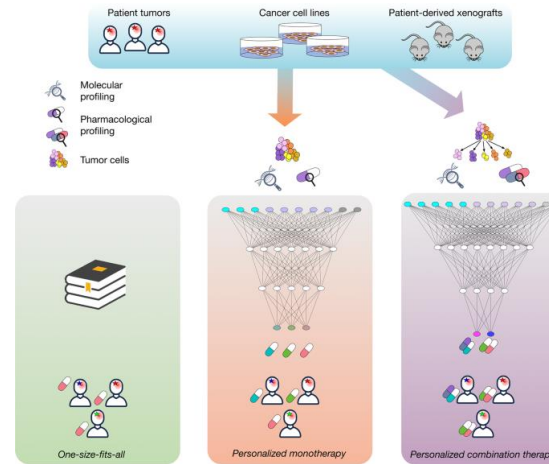


Some Future Directions in DL for Biostatistics



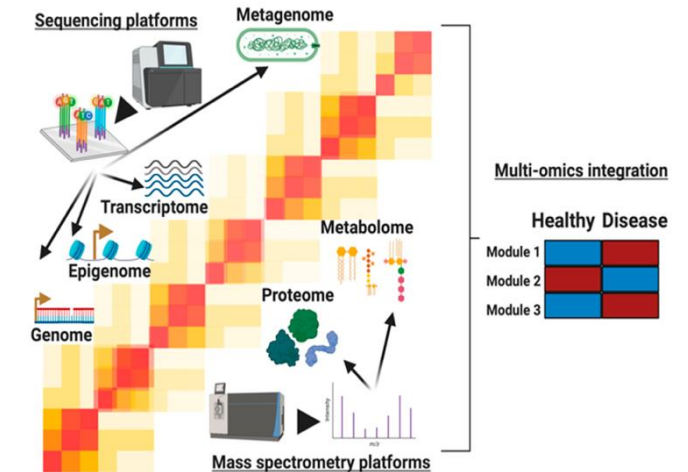
AI-driven Public Health Interventions

Utilizing deep learning models to analyze large-scale public health data for informed decision-making and policy development. Allowing better resource allocation, and more effective epidemic control strategies.



Advanced Drug Response Modeling

Using deep learning to model and predict individual responses to drugs, considering genetic, environmental, and lifestyle factors. Developing more effective personalized treatments.

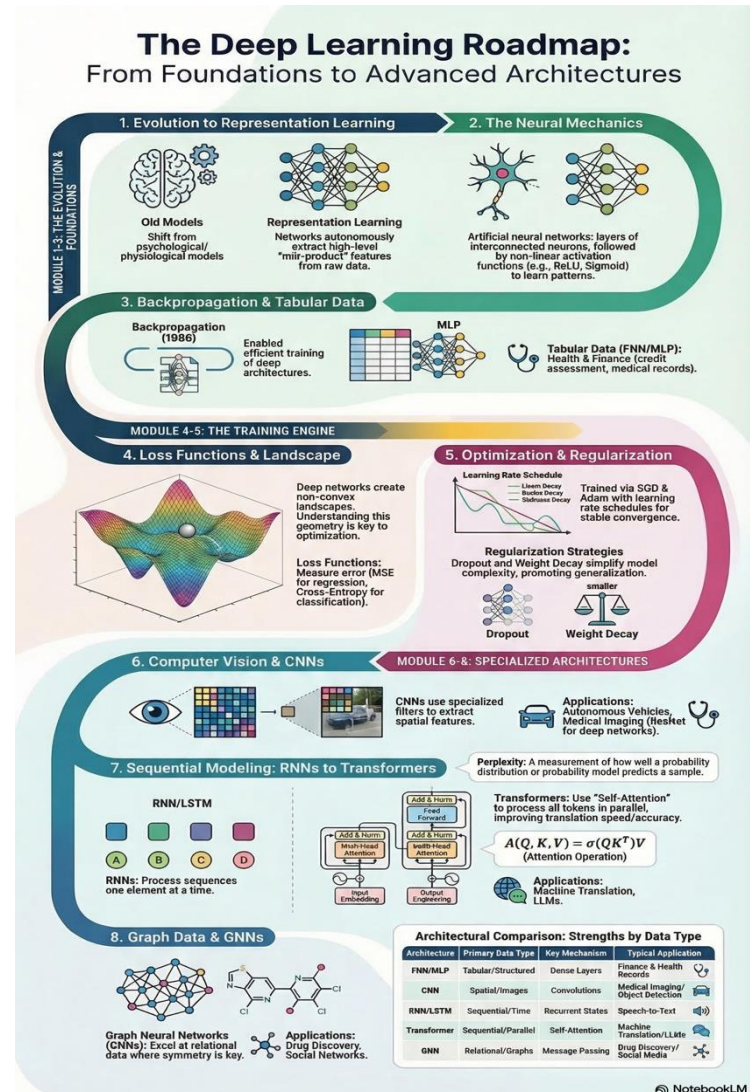


Integrative Analysis of Multi-omic Data

Leveraging deep learning to integrate and analyze data from genomics, proteomics, metabolomics, and other omics fields for a comprehensive understanding of biological processes and disease mechanisms.

Content

2 Neural Network Basics



Neural Network Basics

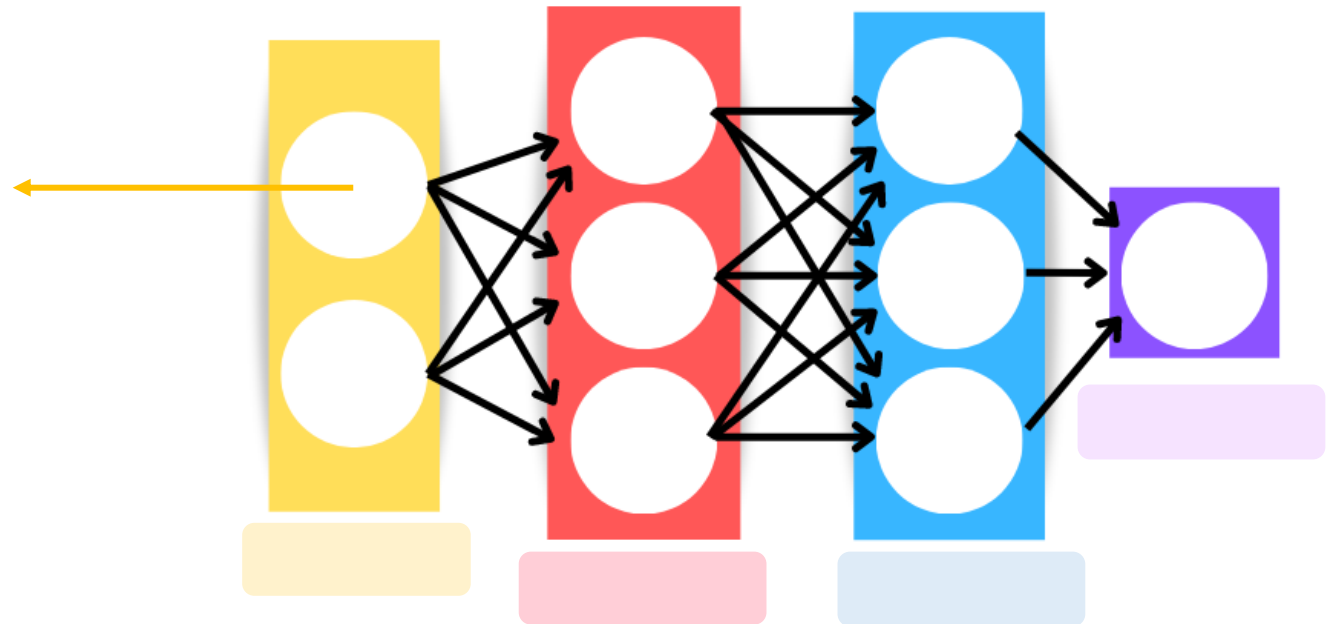
Recap:

What is the relationship between neural network and deep learning?

What are the three types of layers in neural network?

Neurons (Nodes)

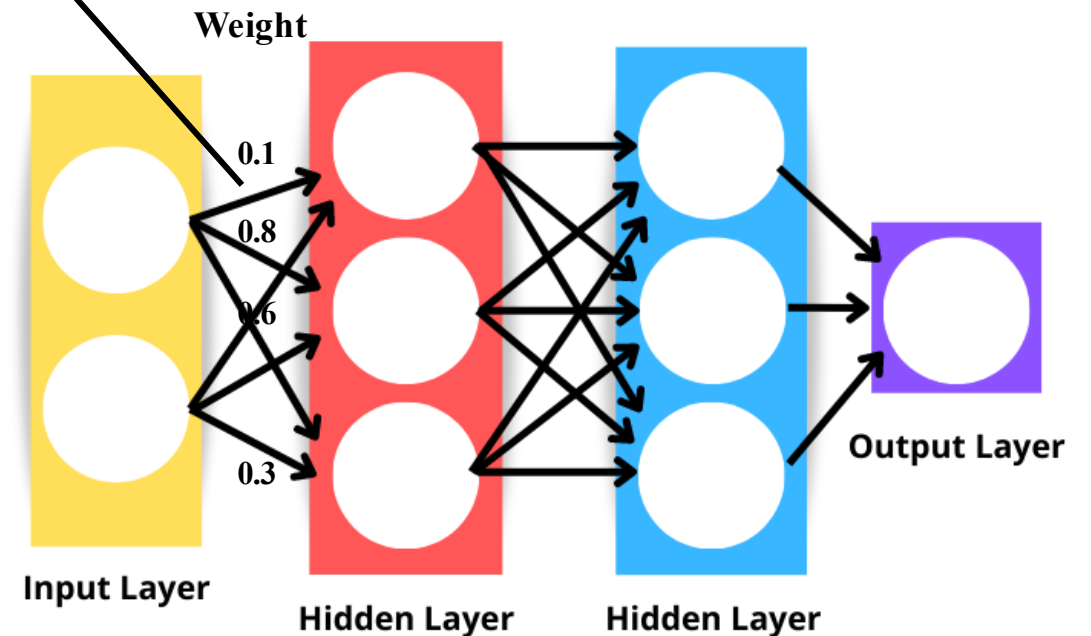
- Fundamental units of a neural network.
- Receive input signals and perform computations and produce an output.
- Neurons in hidden and output layers may use activation functions.
- Activation functions introduce non-linearities for learning complex patterns.



Neural Network Basics

Channels (connections)

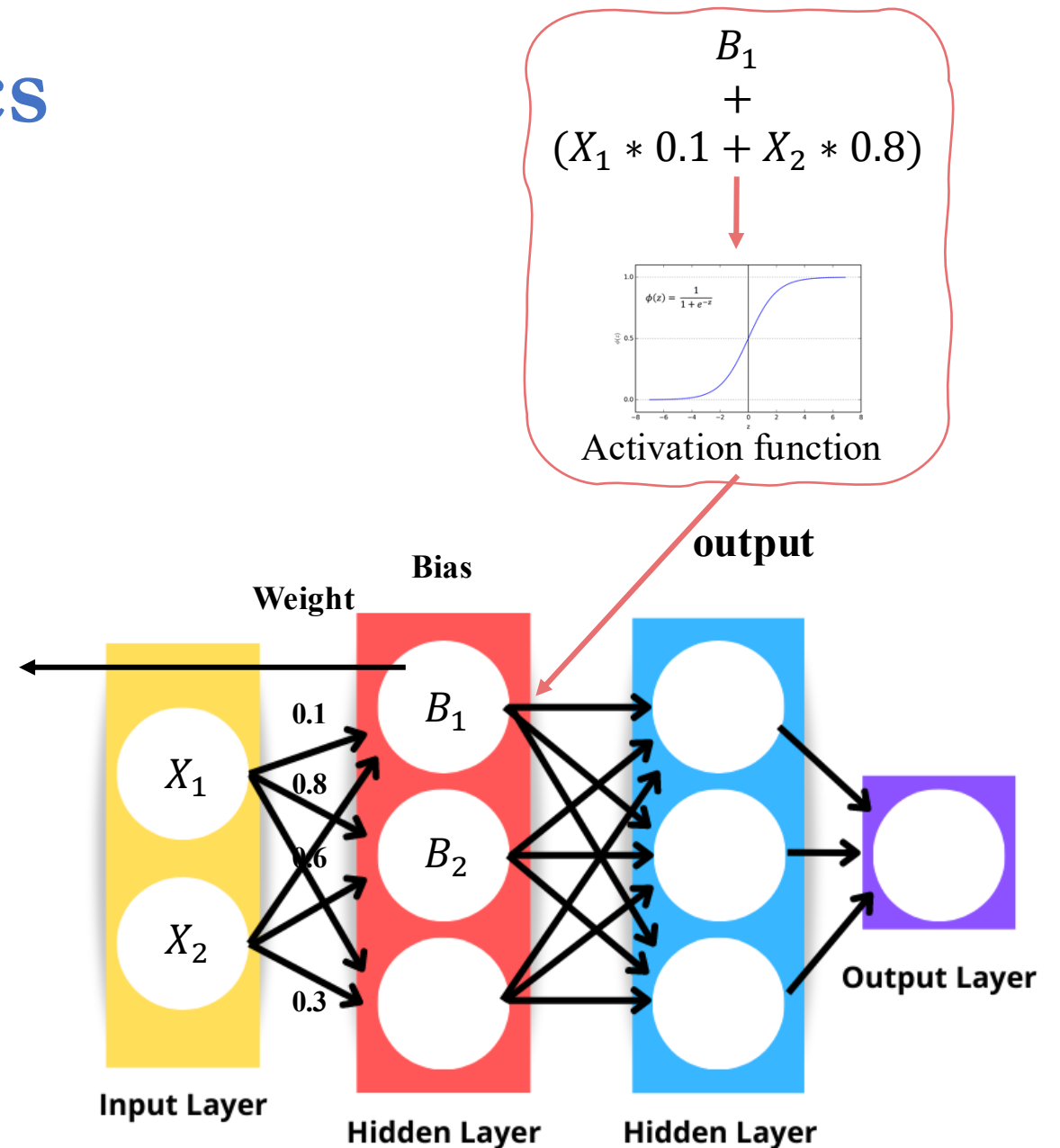
- The information is transferred from one layer (or neurons) to another layer (or neurons) over connecting channels.
- Each connection is associated with a **weight** value that determines the strength of the connection. These weights can be adjusted during training to influence the network's behavior.
- The output of one neuron is multiplied by the weight of the connection and passed as input to the connected neuron in the subsequent layer.



Neural Network Basics

Bias

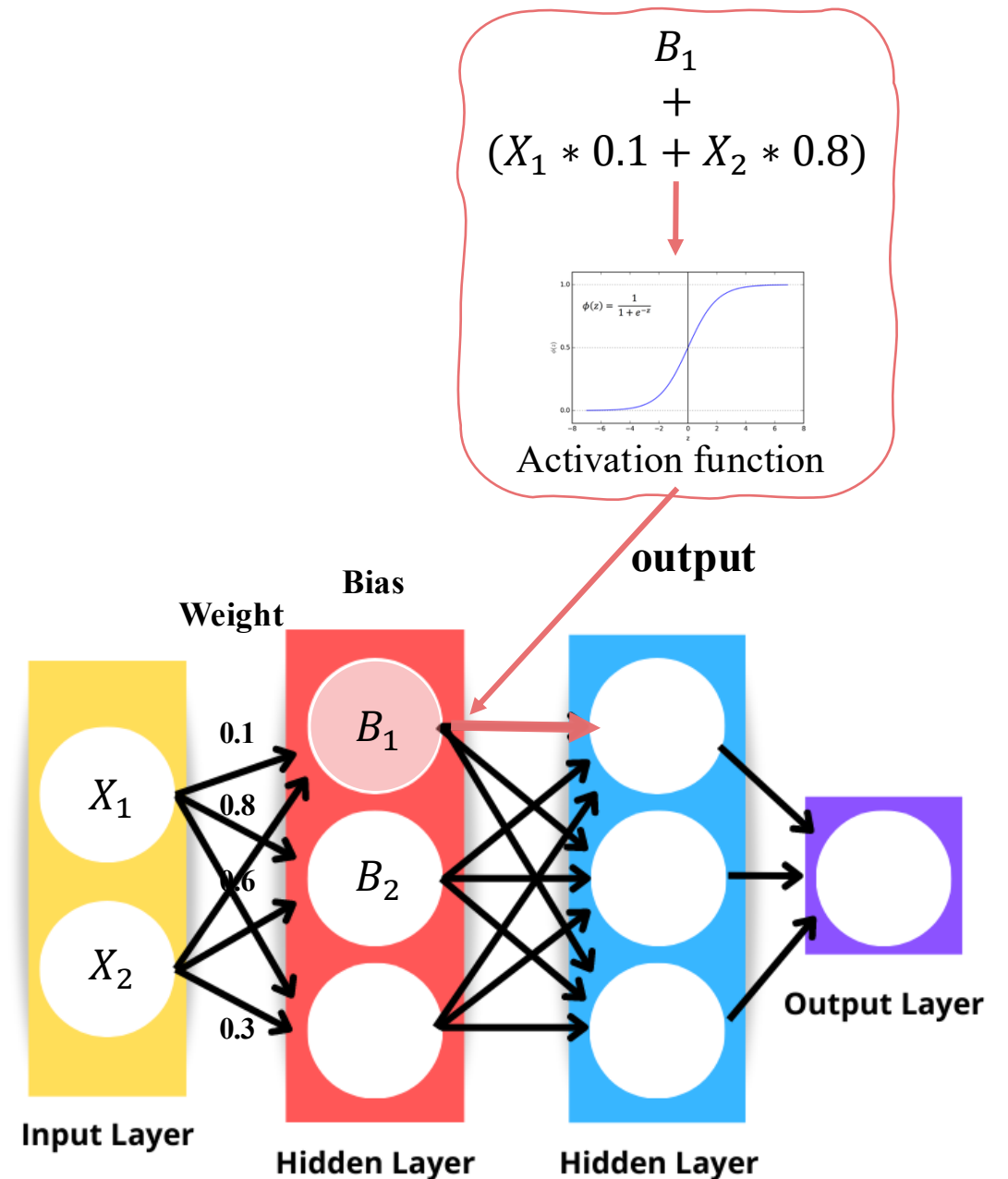
- Biases are also adjustable parameters associated with the connections between neurons in neural networks, which is added to the weighted sum of inputs at each neuron and then applied to **activation function**.
- It allows the network to account for potential systematic errors or deviations from the ideal relationship between inputs and outputs.
- Bias is conceptually similar to the intercept in linear regression, providing flexibility for the network to fit data more accurately.



Neural Network Basics

Activation function

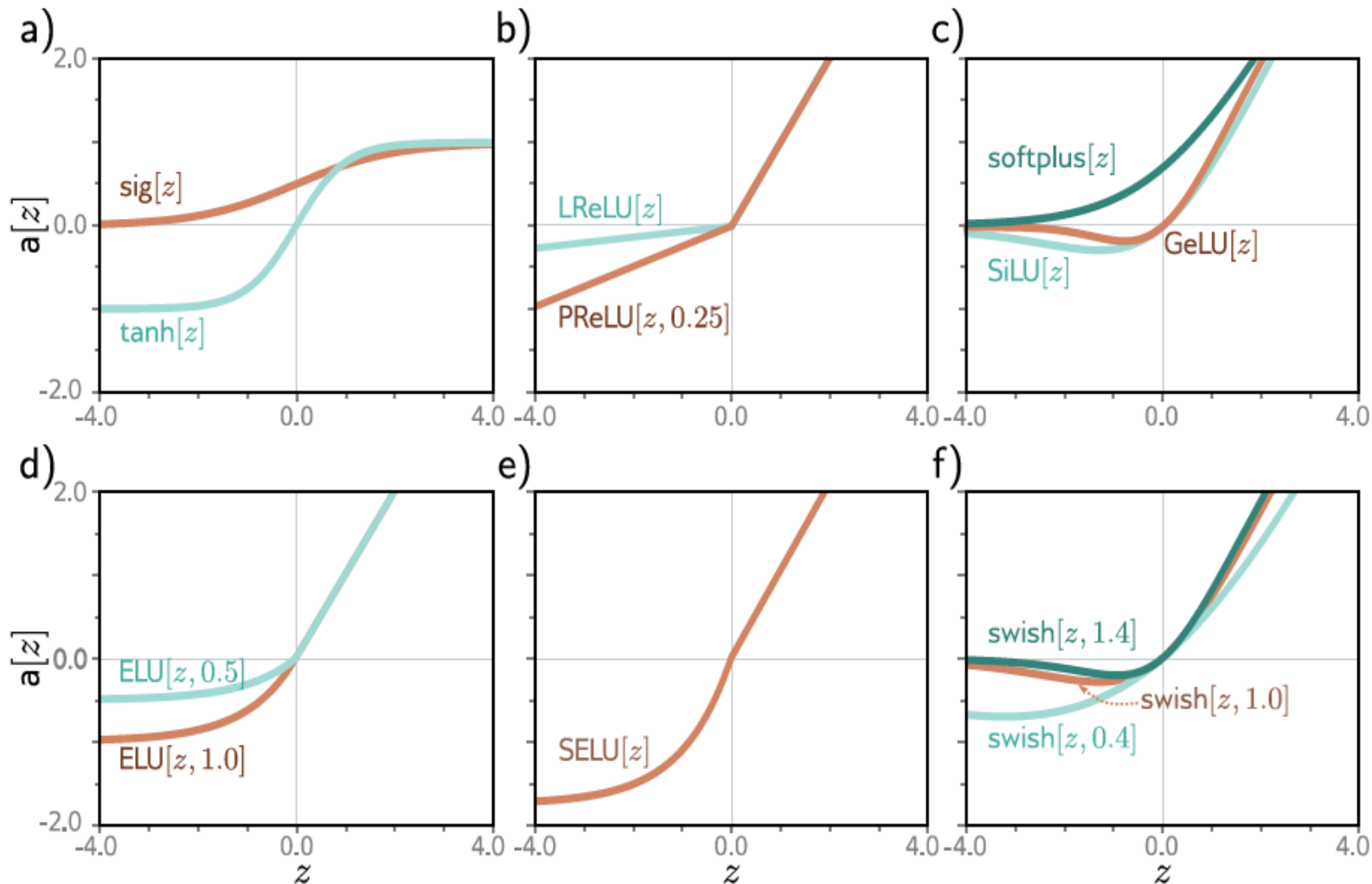
- Activation functions are threshold values that introduce non-linearities into the neural network, enabling it to comprehend complex relationships between inputs and outputs.
- **Common activation functions:** sigmoid, tanh, ReLU (Rectified Linear Unit), and softmax.
- The results of the activation function determine if the particular neuron will get activated or not.
- An activated neuron transmits data (or information) to the neurons of the next layer through channels.



Activation Function - The Gateway to Non-Linearity

- **Introducing Non-Linearity:** Activation functions **introduce non-linear properties to the network**, enabling it to learn complex data patterns beyond the capability of linear models.
- **Transforming Inputs to Outputs:** It takes input from previous layers and converts it to some form of input for the next layers.
- **Essential Building Blocks:** It decides what is to be fired to the next neuron.
- **Beyond Linear Modeling:** Without non-linearity, neural networks would be limited to linear decision boundaries, similar to linear regression.
- **Crucial for Performance:** Non-linear functions allow neural networks to solve advanced problems like image and speech recognition, and natural language processing.

Activation Functions



- a) Logistic sigmoid and tanh functions.
- b) Leaky ReLU and parametric ReLU with parameter 0.25.
- c) SoftPlus, Gaussian error linear unit, and sigmoid linear unit.
- d) Exponential linear unit with parameters 0.5 and 1.0.
- e) Scaled exponential linear unit.
- f) Swish with parameters 0.4, 1.0, and 1.4.

Training of a Neural Network

2 Steps:

1. Forward Propagation

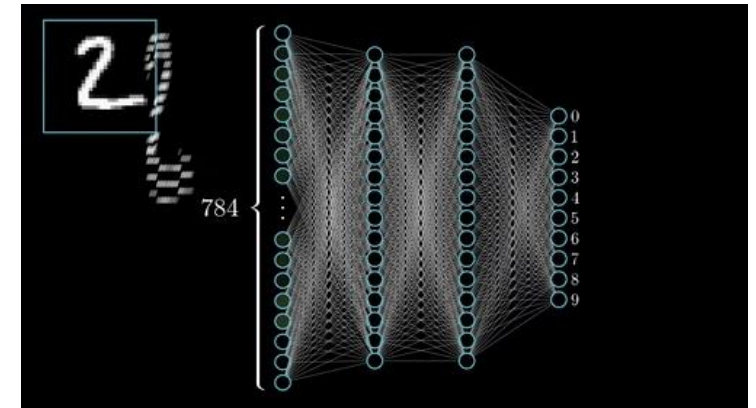
- Forward propagation is **how neural networks make predictions**.
- Involves passing input data through the network layer by layer to the output.

2. Backpropagation

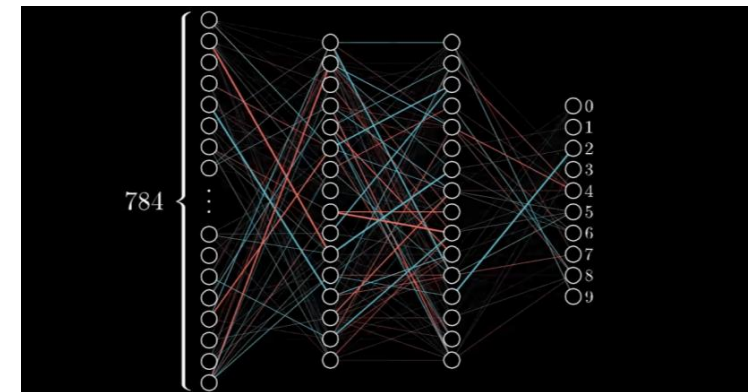
- Backpropagation is the process of adjusting the weights of the network by propagating through the neural network **backward**.
- Involves calculating the gradient of the loss function with respect to each weight by the chain rule.
- The weights are adjusted in the direction that reduces the loss.

Both steps are iteratively repeated for several epochs to minimize the loss and improve the model's accuracy.

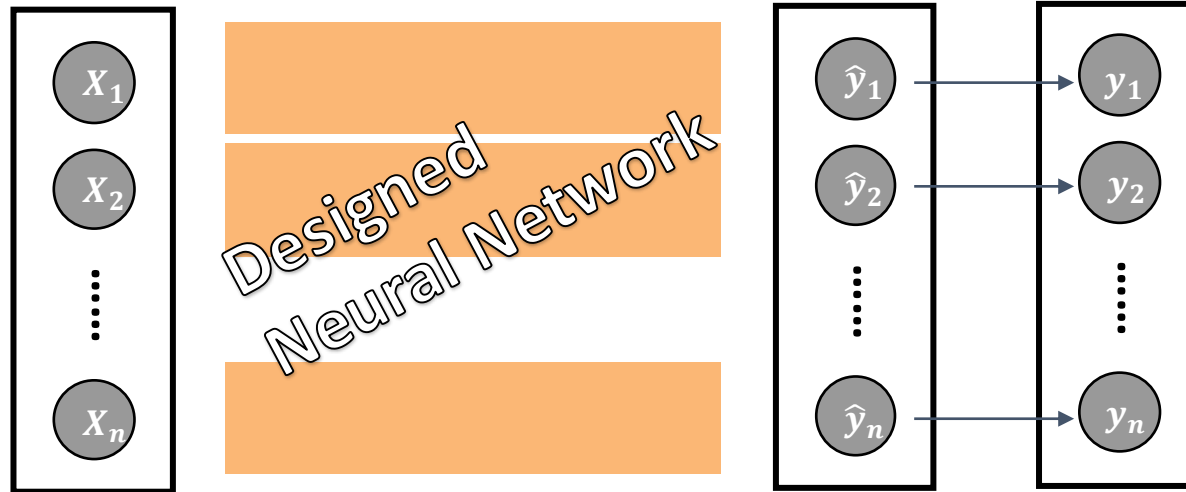
Forward Propagation



Backpropagation



Fitting DL Models



$$\begin{aligned}
 \mathbf{h}_1 &= \mathbf{a}[\beta_0 + \Omega_0 \mathbf{x}] \\
 \mathbf{h}_2 &= \mathbf{a}[\beta_1 + \Omega_1 \mathbf{h}_1] \\
 \mathbf{h}_3 &= \mathbf{a}[\beta_2 + \Omega_2 \mathbf{h}_2] \\
 &\vdots \\
 \mathbf{h}_K &= \mathbf{a}[\beta_{K-1} + \Omega_{K-1} \mathbf{h}_{K-1}] \\
 \mathbf{y} &= \beta_K + \Omega_K \mathbf{h}_K.
 \end{aligned}$$

A **loss function** is needed here,
to measure the difference between the output and truth

Total loss:
$$L = \sum \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i)$$

$$\hat{\mathbf{y}}_i = \beta_K + \Omega_K \mathbf{h}_K(\mathbf{x}_i; [(\beta_0, \Omega_0), \dots, (\beta_{K-1}, \Omega_{K-1})])$$

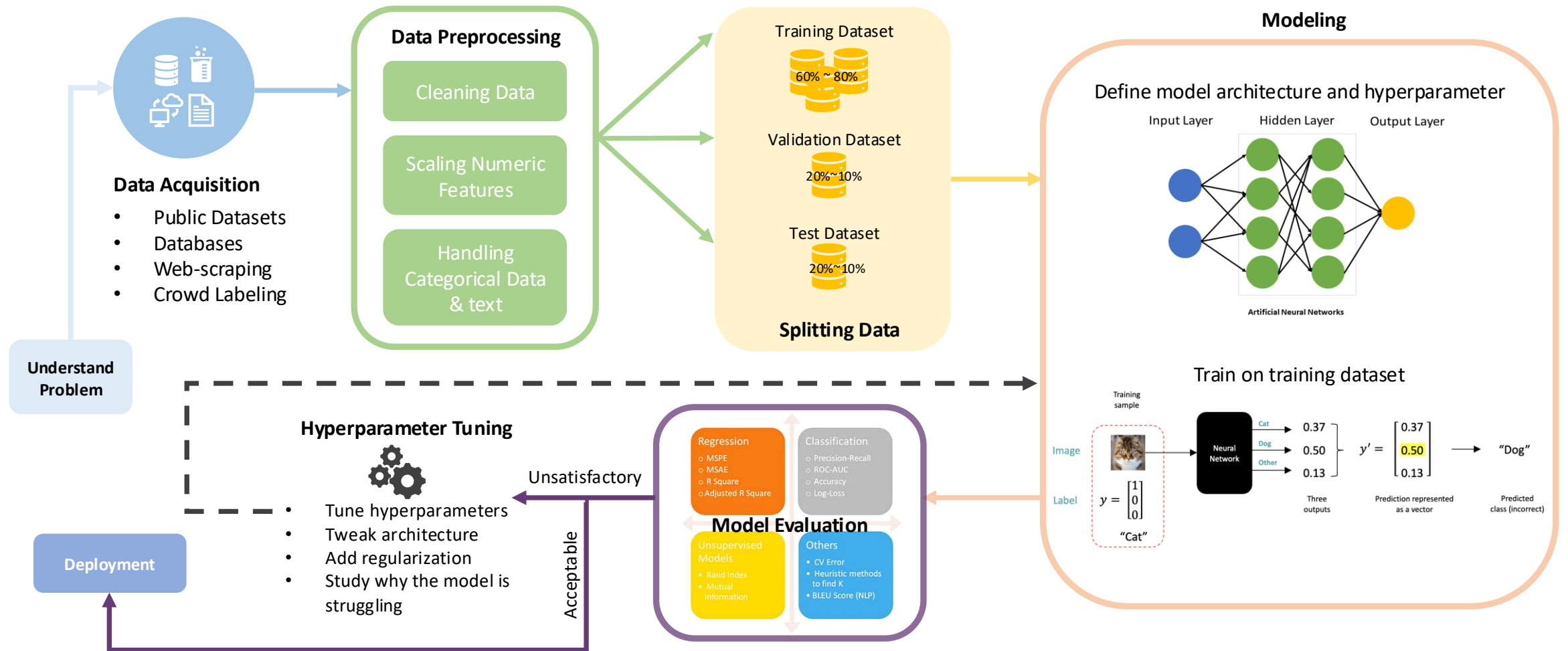
Find the network parameters to minimize the loss

Design the neural network

Find a criterion/measurement of goodness

Get the best model for the problem

Workflow of a Typical DL Project

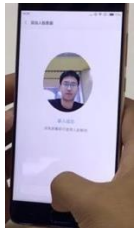


Face Recognition System

Image Acquisition



Camera



Iphone



Monitor Videos

Face Recognition

Face Detection



Feature Extraction



Face Matching



Recognition Results

Model Training



Gabor, LBP etc.

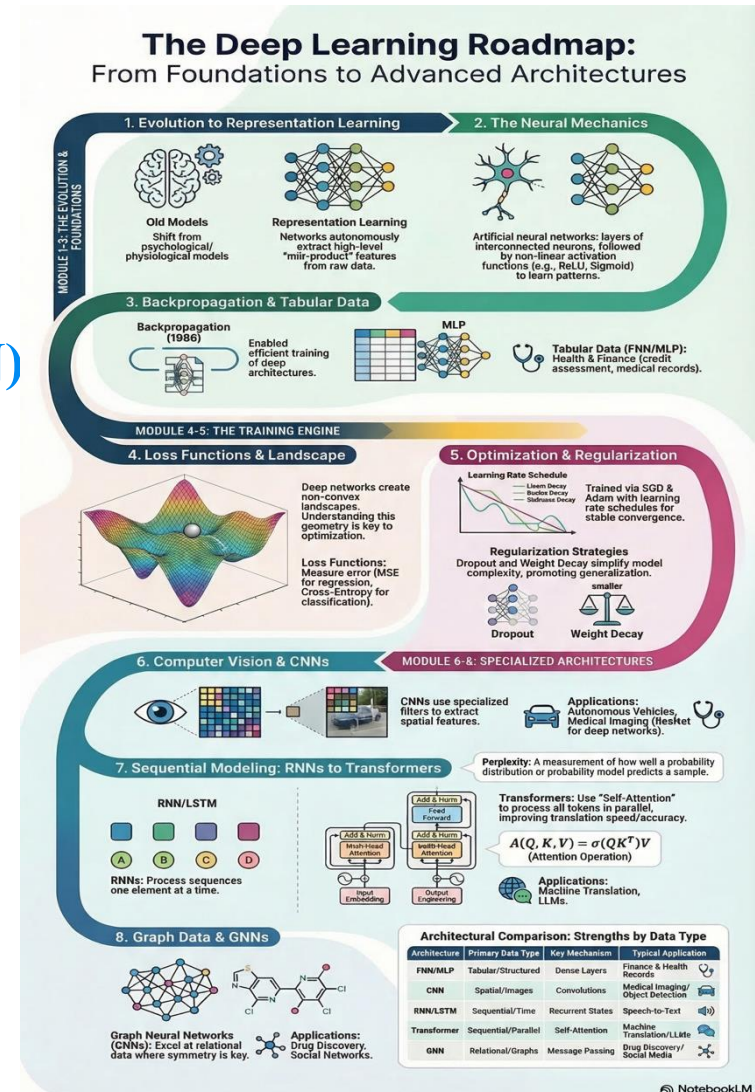


Deep Learning



Content

3 Tabular Data and Fully Connected Neural Network (FCNN)



NotebookLM

What is Tabular Data?

Tabular data are organized like a spreadsheet: rows are units (patients, visits, samples, genes, hospitals), and columns are variables measured on those units.

Clearly tabular examples

- Clinical trial CRFs
- Cohort / registry data
- Survey data (e.g., demographics, exposure, outcomes)
- Administrative / claims tables

Also tabular in practice

- EHR extracts (labs, meds, diagnoses)
- Biobank phenotypes and covariates
- Proteomics / metabolomics matrices
- Gene-expression count matrices

Tabular after summarization

- Imaging-derived features
- NLP features from notes
- Wearable summary metrics
- Variant annotations and burden scores

Tabular

Structured patient data (EHR, biobanks, surveys)

Patient	Age	Sex	Blood Pressure	Diabetes
001	48	F	35-00.0	1
003	38	F	35-29.0	0
004	25	F	34-00.0	1
004	66	F	35-04.0	1

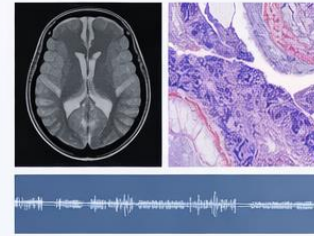
- Risk prediction
- Survival analysis
- Causal inference

Key Tasks

- Risk prediction
- Survival analysis
- Causal inference

Image

Medical imaging (MRI/CT, pathology slides)



- Detection & segmentation
- Diagnosis / grading
- Modality translation

Key Tasks

- Detection sequence
- Phenotyping
- Modality translation

Text

Unstructured language (clinical notes)

clinical note:

Chest pain of possible cardiac origin.

HPI: 55-year-old male presenting with chest pain radiating to the left arm. PMH significant for HTN, DM.

Assessment: Suspected acute coronary syndrome.

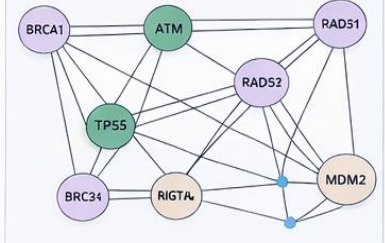
- Information extraction
- Phenotyping
- Summarization & retrieval

Key Tasks

- NLP pipelines
- BERT /transformers
- LLMs, RAG

Graph

Biological networks (PPI, pathways, connectomes)

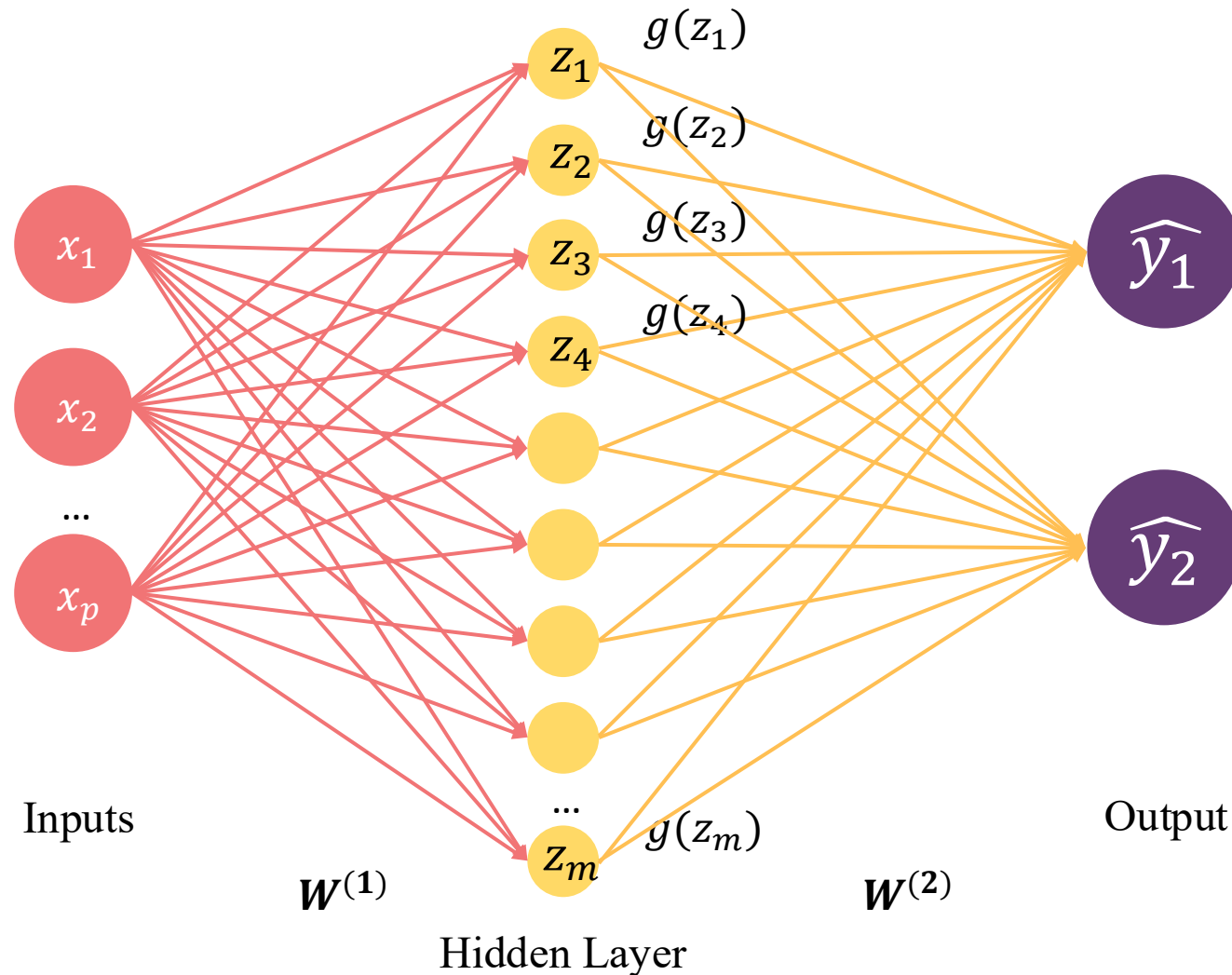


- Link prediction
- Node/graph classification
- Pathway analysis

Key Tasks

- Noisy/incomplete edges
- Scalability
- Evaluation/fiability

Single-Layer Neural Network



Hidden layer:

$$\mathbf{z} = \mathbf{b}_z + \mathbf{X}^T \mathbf{W}^{(1)}$$

Output:

$$\widehat{\mathbf{y}} = \mathbf{b}_y + \mathbf{X}^T \mathbf{W}^{(2)}$$

“Single layer” refers to a network that has **one layer of hidden nodes** between the input and the output layers.

Corresponding code:

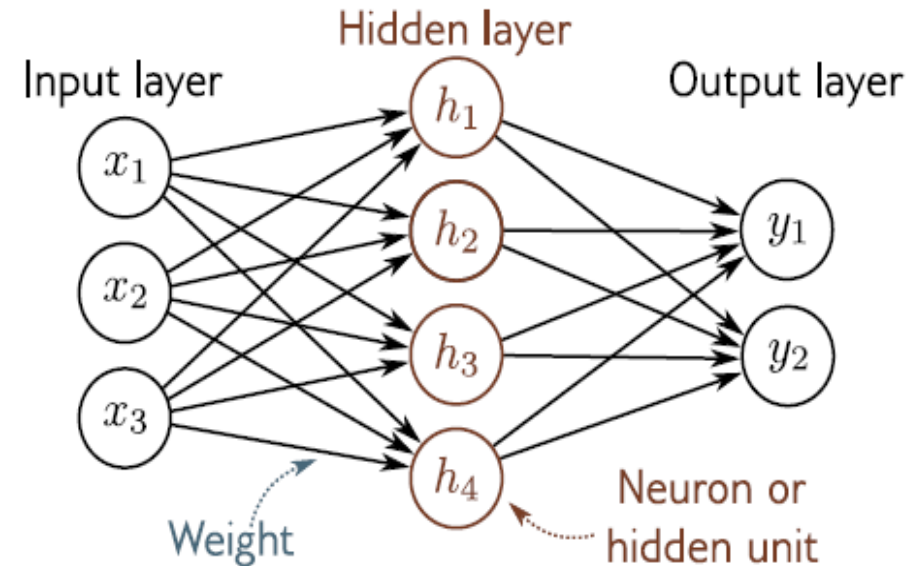
```
self.hidden = nn.Linear(3, m)
self.output = nn.Linear(m, 2)
```

Shallow Neural Network

Universal Approximation Theorem

Cybenko (1989) and Hornik (1991)

A feed-forward network with a **single hidden layer** containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function.



$$h_d = a \left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i \right],$$

$$y_j = \phi_{j0} + \sum_{d=1}^D \phi_{jd} h_d,$$

Multilayer Perceptrons (MLP)

- **Definition:** A Multilayer Perceptron (MLP) is a class of **feedforward neural network (or fully connected network)** that consists of at least three layers of nodes: an input layer, 2+ hidden layers, and an output layer.
- **Hyperparameters:** The **width** of a network refers to the number of hidden units in each layer, while its **depth** indicates the number of hidden layers. **The total number of hidden units** serves as a measure of the network's overall **capacity**.
- **Key Characteristics:**
 - **Multiple Layers:** Unlike single-layer perceptrons, MLPs have multiple layers of neurons in a directed graph, meaning that each layer feeds into the next.
 - **Dense Connections:** Each neuron in one layer connects with a certain weight to every neuron in the following layer, facilitating complex data representations.
- **Why Multilayer?**
 - Single-layer networks are only capable of learning linearly separable functions. MLPs can overcome this by learning non-linear decision boundaries.

Fully Connected Network

$$\mathbf{h}_1 = \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}]$$

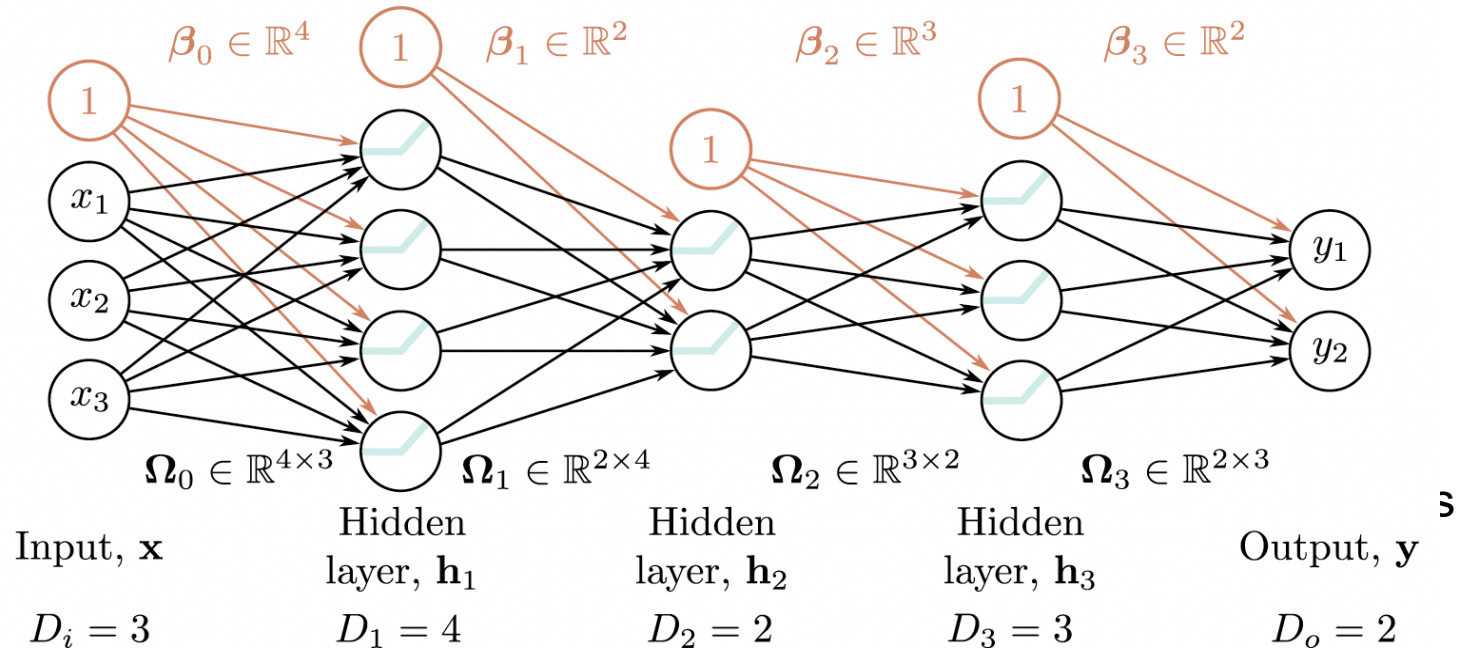
$$\mathbf{h}_2 = \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1]$$

$$\mathbf{h}_3 = \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2]$$

$\vdots$

$$\mathbf{h}_K = \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} \mathbf{h}_{K-1}]$$

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{h}_K.$$



$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{a} [\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} \mathbf{a} [\dots \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{a} [\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{a} [\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}]] \dots]] .$$

❖ When it's used:

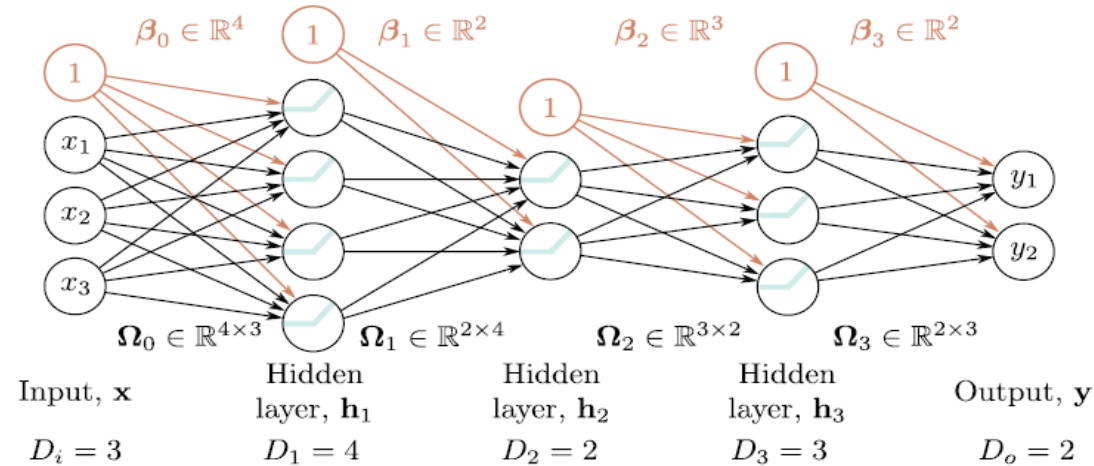
❖ Great for **tabular data** and low-dimensional features

❖ Common as the **classifier “head”** on top of CNN/Transformer features

❖ Limitation:

❖ For images or sequences, fully connected layers can be **parameter-heavy** and ignore spatial/temporal structure, so CNNs/Transformers are often preferred for those inputs.

Deep Neural Network



Shallow vs deep networks

- ❖ both networks can approximate any function given enough capacity,
- ❖ deep networks produce many more linear regions per parameter,
- ❖ some functions can be approximated much more efficiently by deep networks,
- ❖ in practice, the best results for most tasks are achieved using deep networks with many layers.

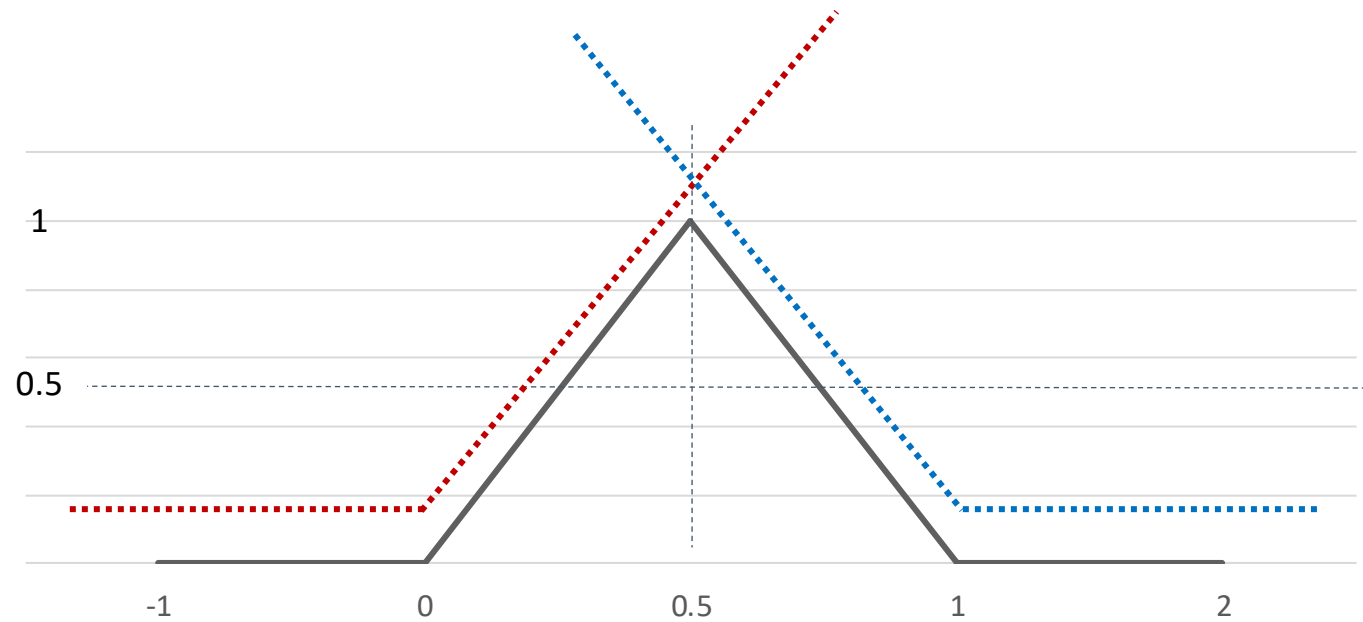
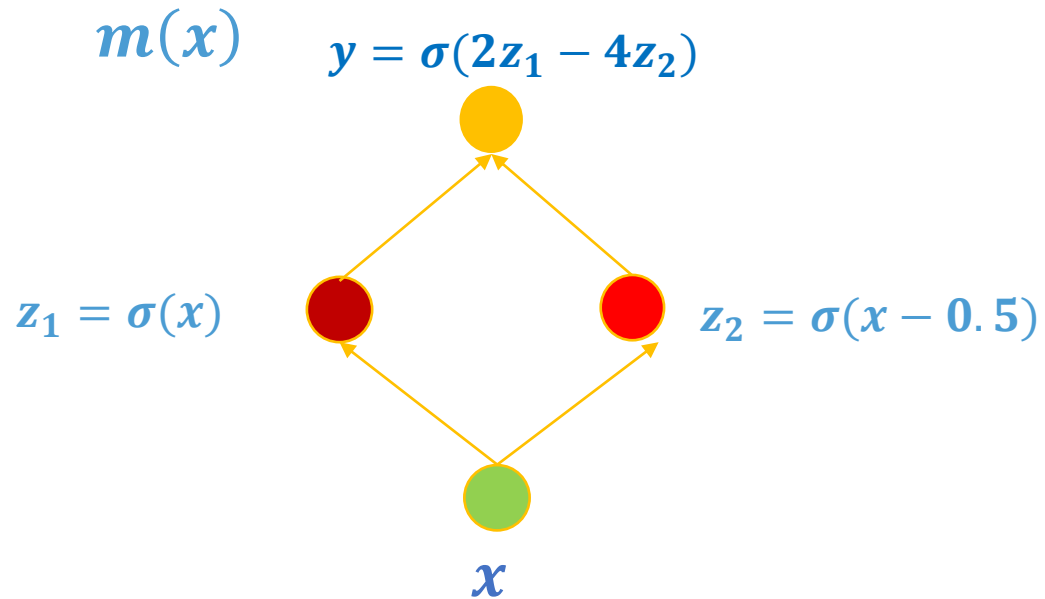
$$\mathbf{y} = \beta_K + \Omega_K \mathbf{a} [\beta_{K-1} + \Omega_{K-1} \mathbf{a} [\dots \beta_2 + \Omega_2 \mathbf{a} [\beta_1 + \Omega_1 \mathbf{a} [\beta_0 + \Omega_0 \mathbf{x}]] \dots]] .$$

- ❖ The number of hidden units in each layer is referred to as the **width** of the network, and the number of hidden layers as the **depth**. The total number of hidden units is a measure of the network's **capacity**.
- ❖ The **depth version** of the universal approximation theorem (Lu et al., 2017): There exists a network with ReLU activation functions and at least D_i+4 hidden units in each layer can approximate any specified D_i -dimensional Lebesgue integrable function to arbitrary accuracy given enough layers.

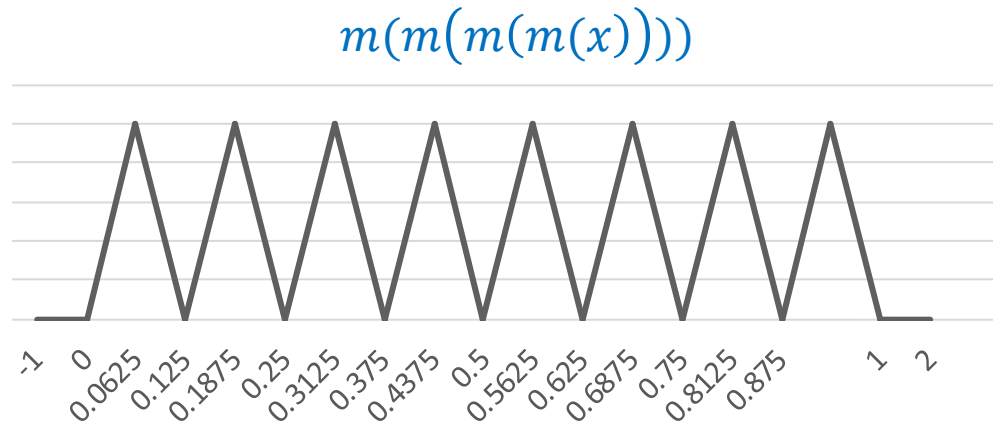
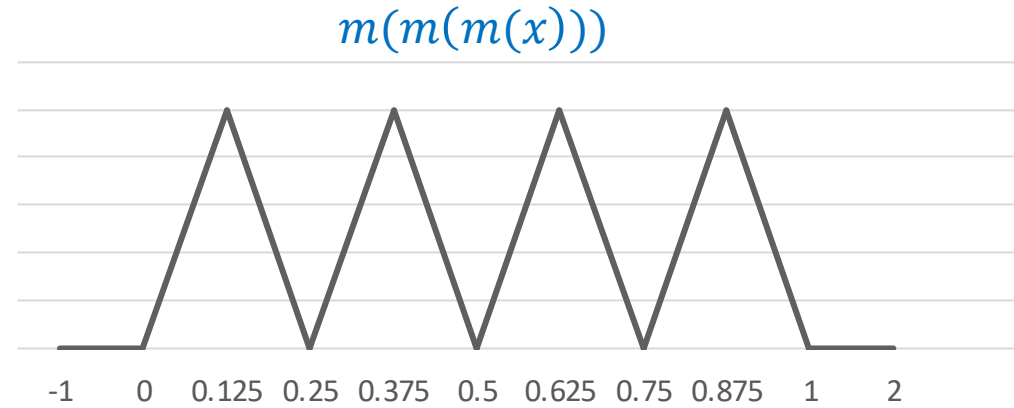
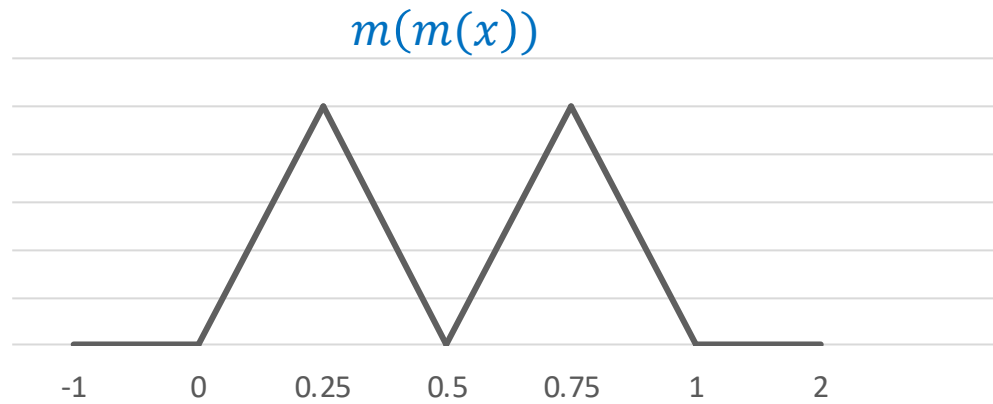
$$\begin{aligned} \mathbf{h}_1 &= \mathbf{a}[\beta_0 + \Omega_0 \mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{a}[\beta_1 + \Omega_1 \mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{a}[\beta_2 + \Omega_2 \mathbf{h}_2] \\ &\vdots \\ \mathbf{h}_K &= \mathbf{a}[\beta_{K-1} + \Omega_{K-1} \mathbf{h}_{K-1}] \\ \mathbf{y} &= \beta_K + \Omega_K \mathbf{h}_K. \end{aligned}$$

When does FCNN outperform traditional methods?

Consider a piecewise linear function $m(x) = \begin{cases} 2x, & x \in [0, 0.5] \\ 2 - 2x, & x \in [0.5, 1] \\ 0 & \text{otherwise} \end{cases}$ Define $\sigma(x) = \max(0, x)$

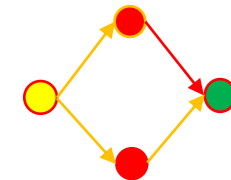


When does FCNN outperform traditional methods?



Generate a deep network:

$3n+1$ nodes to represent the $m^{(n)}(x)$
with width of each layer ≤ 2



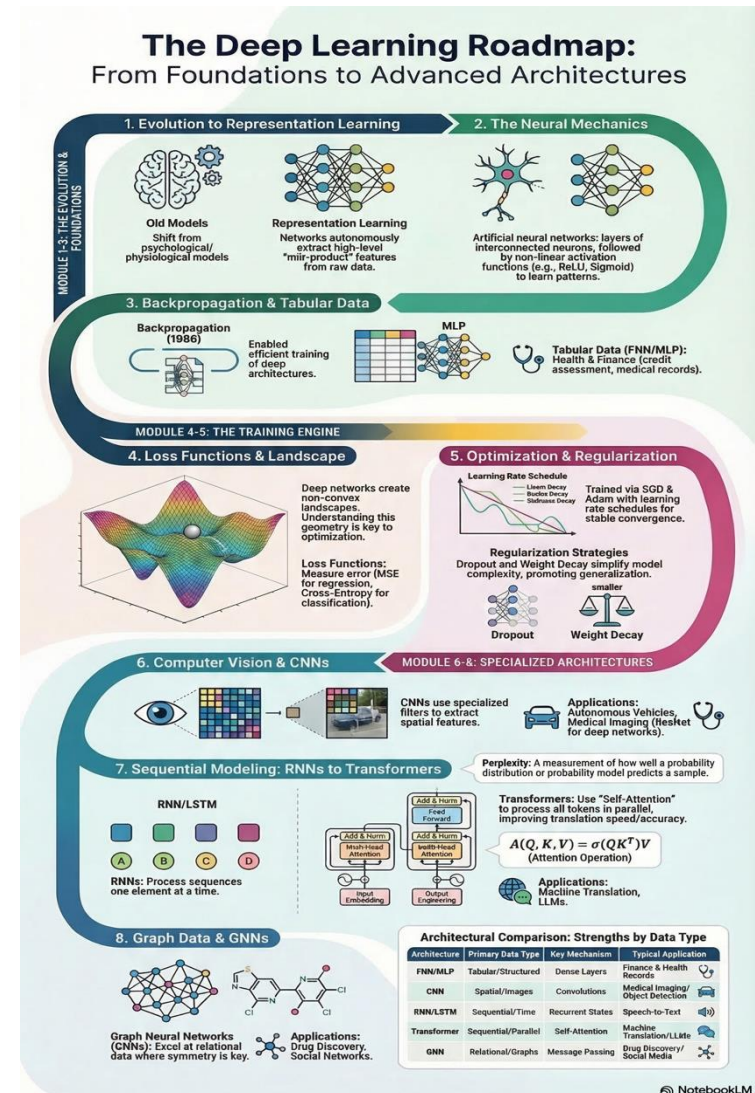
Depth
- Multiplicatively

If we generate a shallow one, we need 2^n nodes

Width
- additively

Content

4 Loss Functions and Optimization Techniques



NotebookLM

Loss Function

Definition: a measure of error between what your model predicts and what the actual value is.

Purpose: quantifies how well the neural network matches what we want to output and thus guides the optimization process.

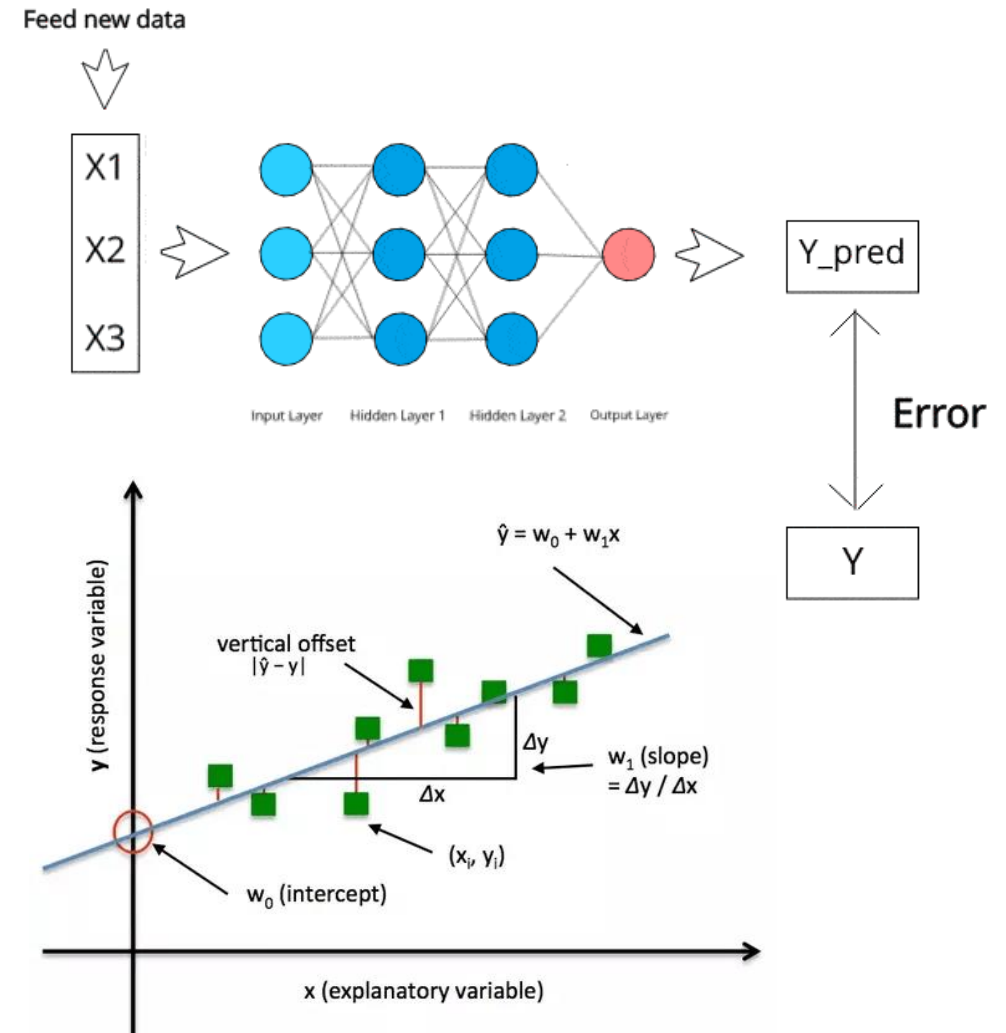
Importance: The choice of loss function directly impacts how the weights of the model are adjusted.

Examples: Mean Squared Error (Regression), Cross-Entropy (Classification).

Notation:

$$\mathcal{L}(f(X; W) \boxed{y})$$

Prediction True



Recipe for Constructing Loss Functions

Recipe for constructing loss functions

The recipe for constructing loss functions for training data $\{\mathbf{x}_i, \mathbf{y}_i\}$ using the maximum likelihood approach is hence:

1. Choose a suitable probability distribution $Pr(\mathbf{y}|\boldsymbol{\theta})$ defined over the domain of the predictions $\mathbf{y}$ with distribution parameters $\boldsymbol{\theta}$.
2. Set the machine learning model $\mathbf{f}[\mathbf{x}, \phi]$ to predict one or more of these parameters, so $\boldsymbol{\theta} = \mathbf{f}[\mathbf{x}, \phi]$ and $Pr(\mathbf{y}|\boldsymbol{\theta}) = Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \phi])$.
3. To train the model, find the network parameters $\hat{\phi}$ that minimize the negative log-likelihood loss function over the training dataset pairs $\{\mathbf{x}_i, \mathbf{y}_i\}$:

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}} [L[\phi]] = \underset{\phi}{\operatorname{argmin}} \left[- \sum_{i=1}^I \log [Pr(\mathbf{y}_i | \mathbf{f}[\mathbf{x}_i, \phi])] \right]. \quad (5.6)$$

4. To perform inference for a new test example $\mathbf{x}$, return either the full distribution $Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \hat{\phi}])$ or the value where this distribution is maximized.

Data Type	Domain	Distribution	Use
univariate, continuous, unbounded	$y \in \mathbb{R}$	univariate normal	regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	Laplace or t-distribution	robust regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	mixture of Gaussians	multimodal regression
univariate, continuous, bounded below	$y \in \mathbb{R}^+$	exponential or gamma	predicting magnitude
univariate, continuous, bounded	$y \in [0, 1]$	beta	predicting proportions
multivariate, continuous, unbounded	$\mathbf{y} \in \mathbb{R}^K$	multivariate normal	multivariate regression
univariate, continuous, circular	$y \in (-\pi, \pi]$	von Mises	predicting direction
univariate, discrete, binary	$y \in \{0, 1\}$	Bernoulli	binary classification
univariate, discrete, bounded	$y \in \{1, 2, \dots, K\}$	categorical	multiclass classification
univariate, discrete, bounded below	$y \in [0, 1, 2, 3, \dots]$	Poisson	predicting event counts
multivariate, discrete, permutation	$\mathbf{y} \in \text{Perm}[1, 2, \dots, K]$	Plackett-Luce	ranking

Loss Function for Classification

Binary Classification Task

Binary Cross-Entropy

Multi-Class Classification Task

Cross-Entropy Loss

Kullback Leibler Divergence Loss

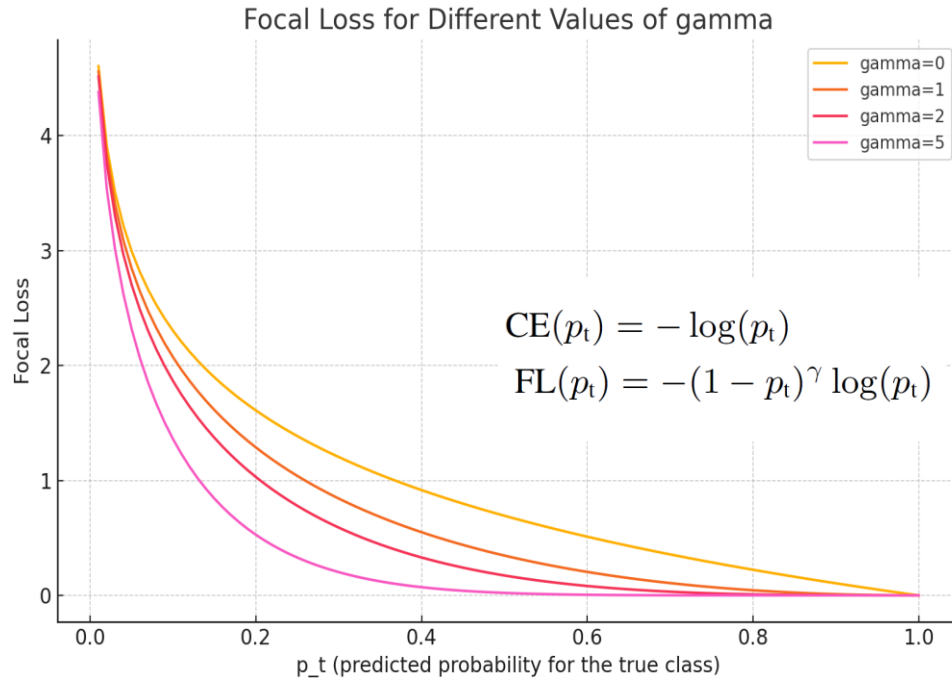
Negative Log Likelihood Loss

Imbalanced Data-Loss Functions

❖ Consider Data Characteristics:

❖ Imbalanced Data: Use Weighted Cross-Entropy or Focal Loss.

❖ Outliers: Use Huber Loss or Mean Absolute Error.



1. **Weighted Likelihood:** Modify the likelihood function to emphasize minority class samples:

$$\mathcal{L} = \sum_{i=1}^N w_{y_i} \log P(y_i|x_i; \theta)$$

where w_{y_i} is inversely proportional to the class frequency.

2. **Cost-Sensitive Likelihood:** Introduce class-specific penalties:

$$\mathcal{L}_{\text{weighted}} = - \sum_{i=1}^N \frac{1}{f_{y_i}} \log P(y_i|x_i; \theta)$$

where f_{y_i} is the frequency of class y_i .

3. **Focal Loss:** Focuses on hard-to-classify examples:

$$\text{Loss}_{\text{focal}} = -\alpha(1 - p_t)^\gamma \log(p_t)$$

where α controls class weighting, and γ modulates the focus on hard examples.

Class-Balanced Loss: Reweights classes based on their effective number of samples:

$$w_c = \frac{1 - \beta}{1 - \beta^{n_c}}$$

where n_c is the number of samples in class c , and $\beta \in [0, 1)$.

Outliers-Loss Functions

Huber Loss: Combines ℓ_1 and ℓ_2 loss to handle small and large residuals differently:

$$\text{Loss}_{\text{Huber}} = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{if } |y - f(x)| \leq \delta, \\ \delta|y - f(x)| - \frac{\delta^2}{2} & \text{otherwise.} \end{cases}$$

Tukey's Biweight Loss: Suppresses large residuals:

$$\text{Loss}_{\text{Tukey}} = \begin{cases} \frac{\delta^2}{6} \left(1 - \left(1 - \frac{r^2}{\delta^2} \right)^3 \right) & |r| \leq \delta, \\ \frac{\delta^2}{6} & |r| > \delta, \end{cases}$$

where $r = y - f(x)$.

Quantile Loss: Focuses on specific quantiles:

$$\text{Loss}_{\text{quantile}} = \max(\tau \cdot e, (1 - \tau) \cdot e)$$

where τ is the target quantile, and $e = y - f(x)$.

Barron, J. T. (2019). A general and adaptive robust loss function. In *CVPR*.

$$\rho(x, \alpha, c) = \begin{cases} \frac{1}{2} (x/c)^2 & \text{if } \alpha = 2 \\ \log \left(\frac{1}{2} (x/c)^2 + 1 \right) & \text{if } \alpha = 0 \\ 1 - \exp \left(-\frac{1}{2} (x/c)^2 \right) & \text{if } \alpha = -\infty \\ \frac{|\alpha-2|}{\alpha} \left(\left(\frac{(x/c)^2}{|\alpha-2|} + 1 \right)^{\alpha/2} - 1 \right) & \text{otherwise} \end{cases}$$

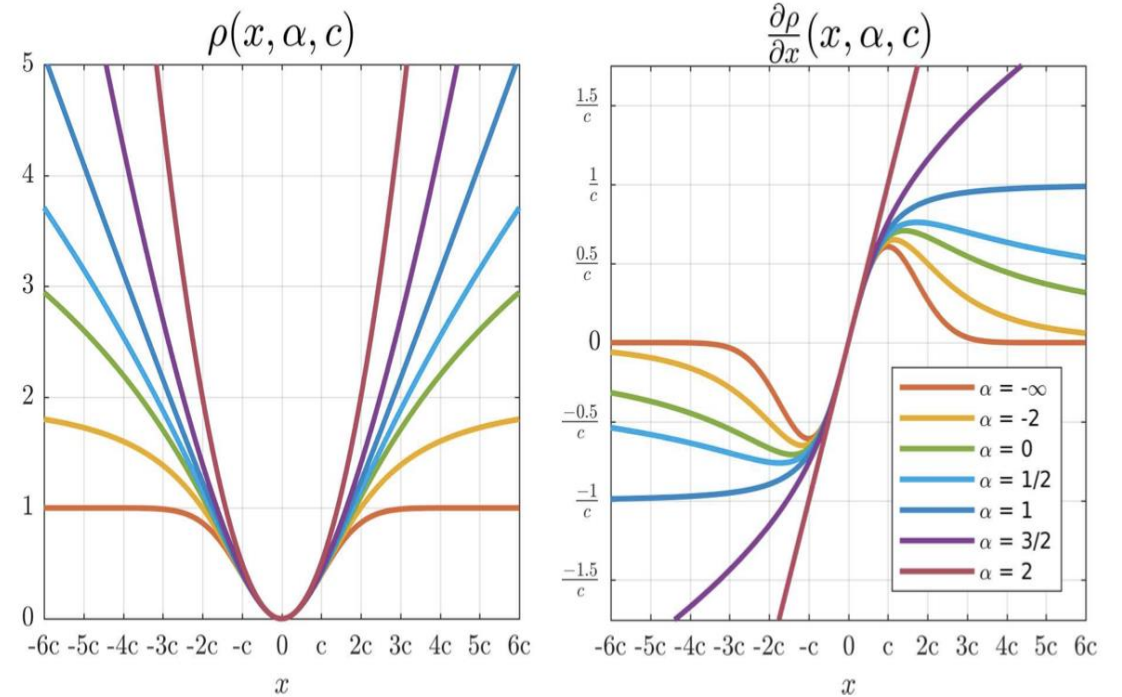


Figure 1. Our general loss function (left) and its gradient (right) for different values of its shape parameter α . Several values of α reproduce existing loss functions: L2 loss ($\alpha = 2$), Charbonnier loss ($\alpha = 1$), Cauchy loss ($\alpha = 0$), Geman-McClure loss ($\alpha = -2$), and Welsch loss ($\alpha = -\infty$).

Other Loss Functions

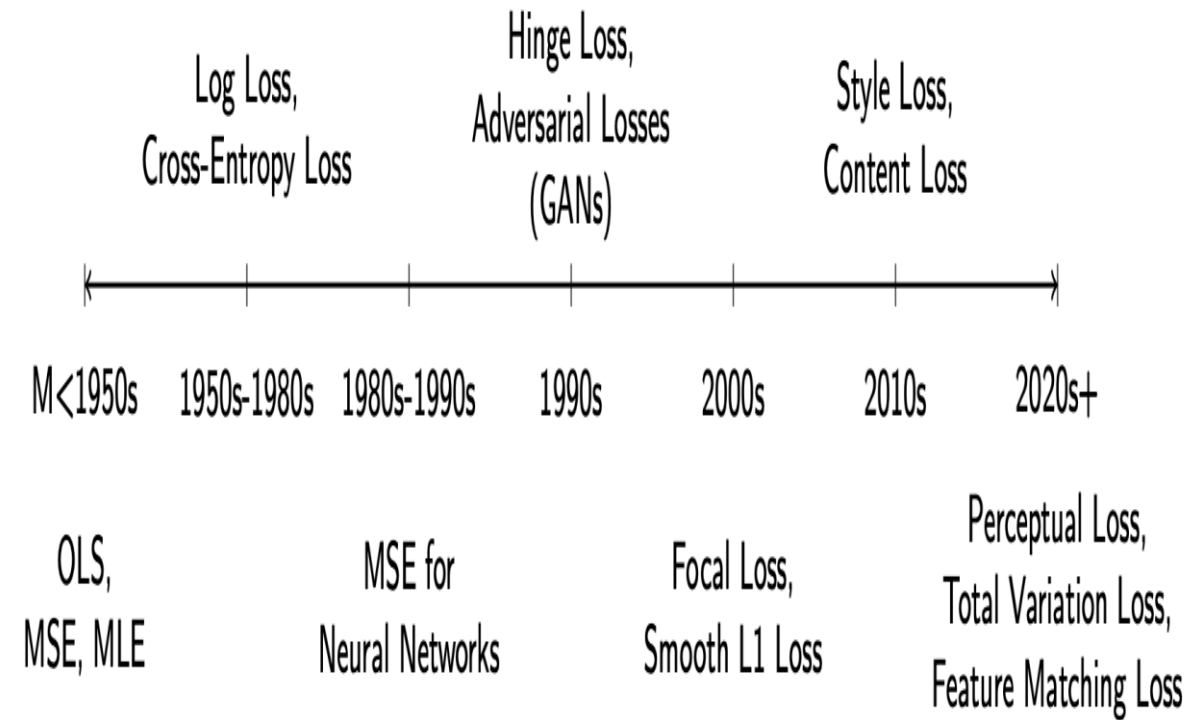
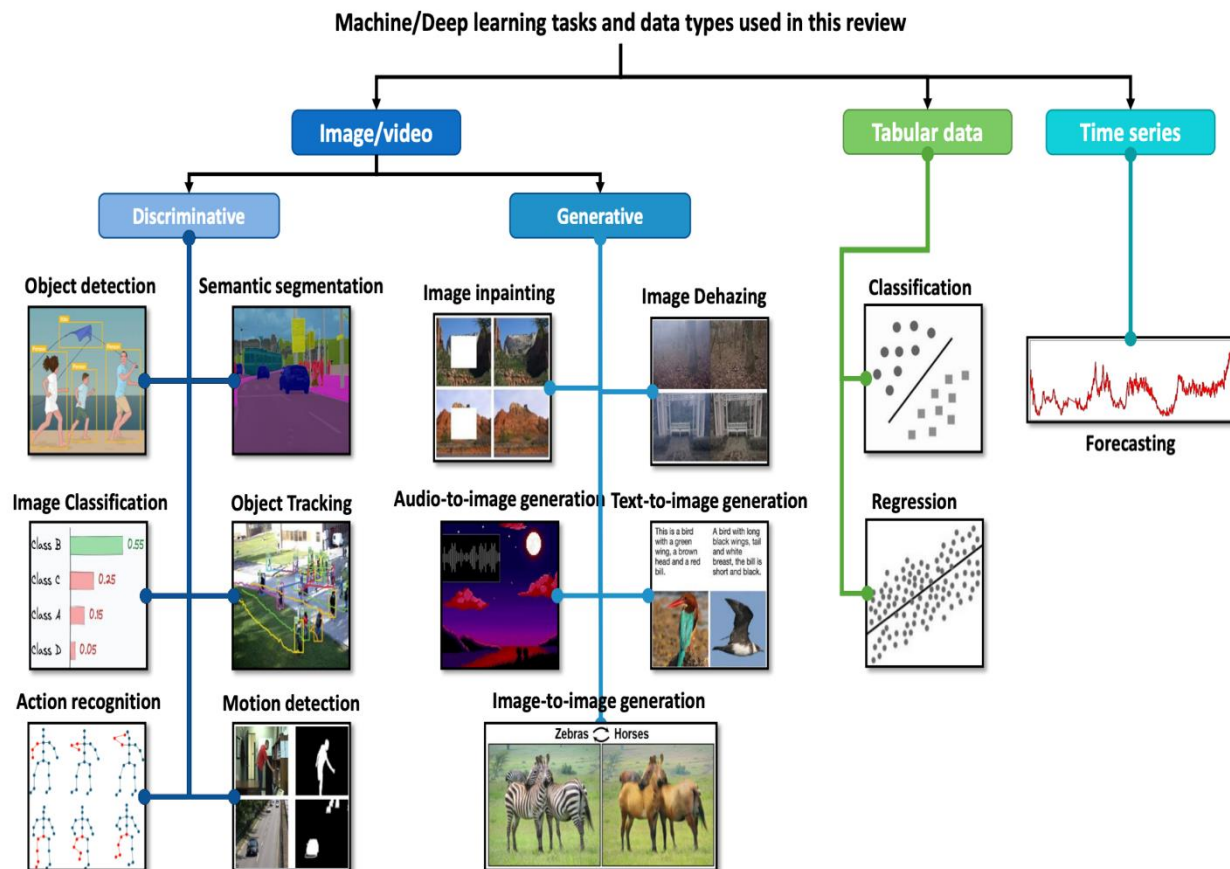


Figure 3: Timeline of the evolution of loss functions.

<https://arxiv.org/abs/2504.04242>

<https://arxiv.org/abs/2307.02694>

Other Loss Functions in Computer Vision

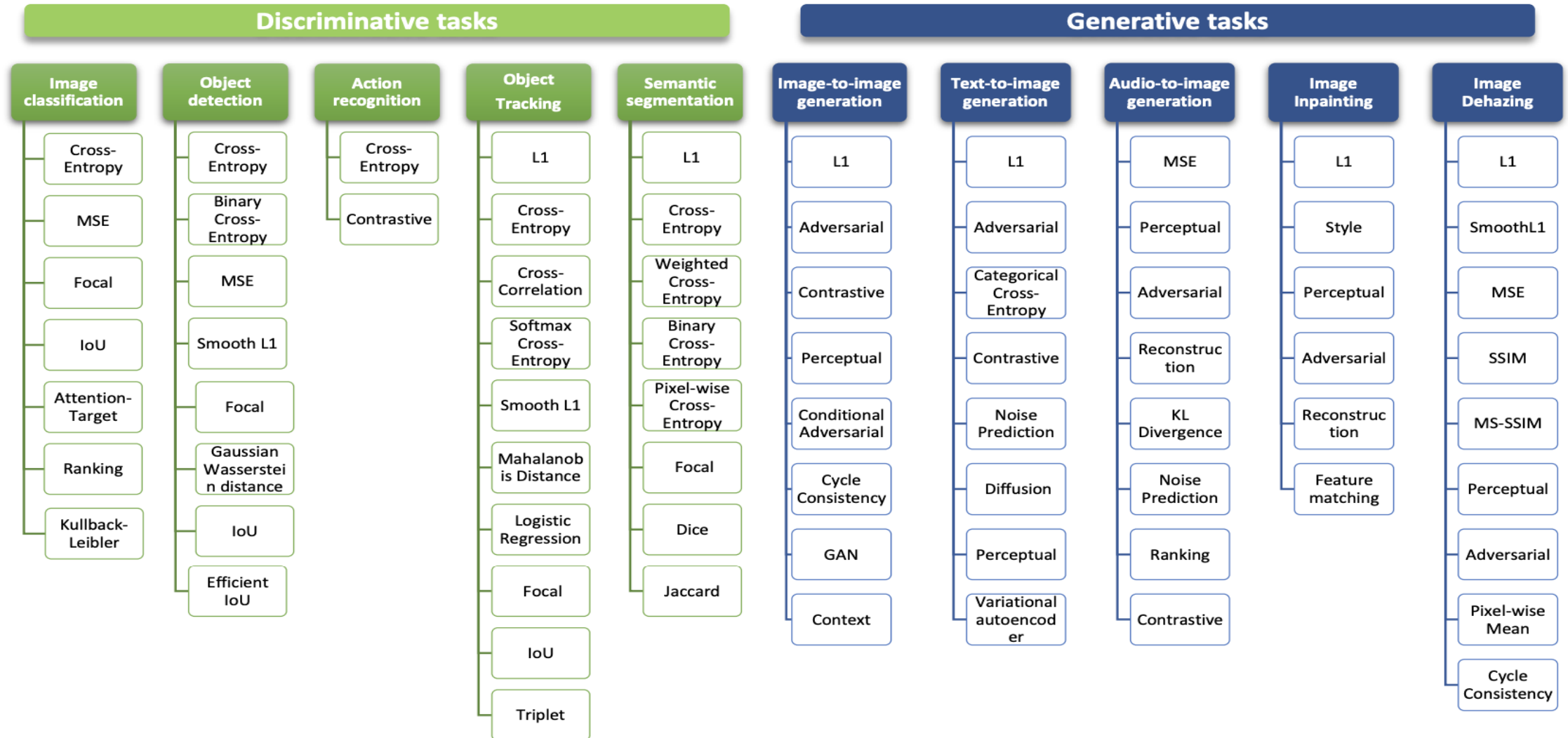
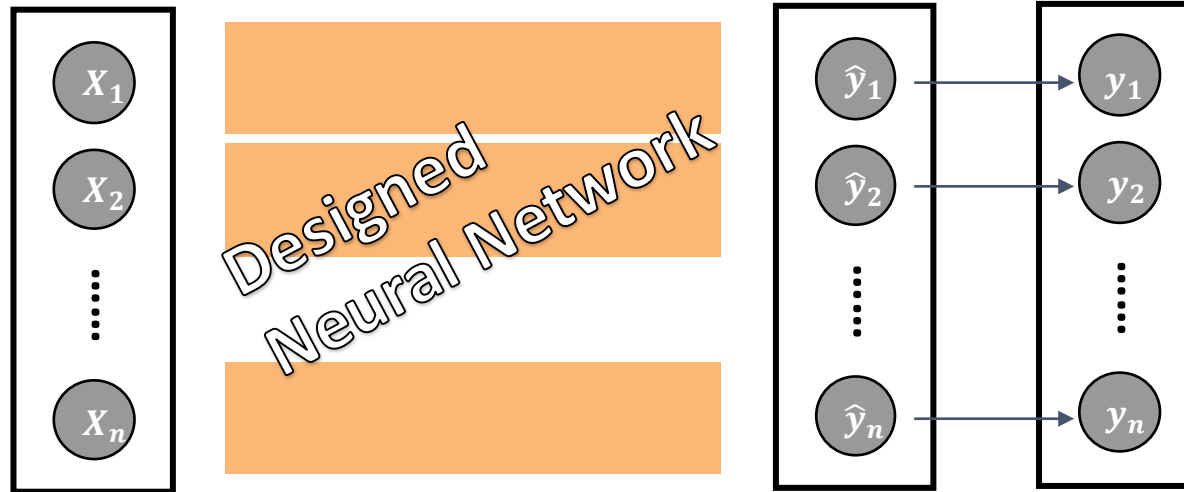


Figure 7: Loss function used in various computer vision (Discriminative, generative) tasks.

Fitting DL Models



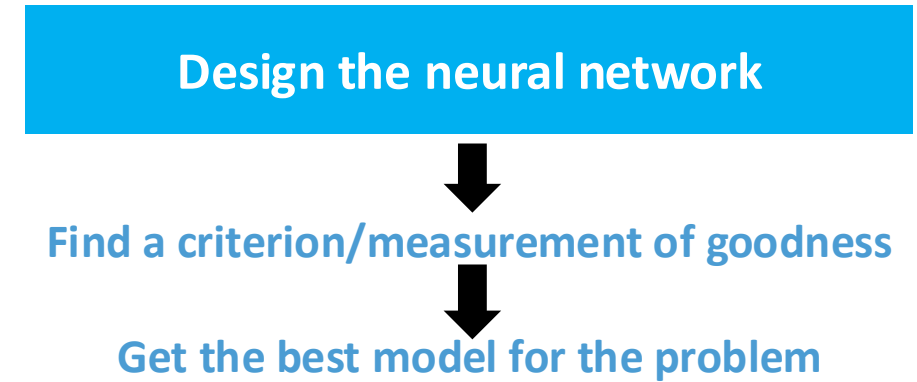
$$\begin{aligned}
 \mathbf{h}_1 &= \mathbf{a}[\beta_0 + \Omega_0 \mathbf{x}] \\
 \mathbf{h}_2 &= \mathbf{a}[\beta_1 + \Omega_1 \mathbf{h}_1] \\
 \mathbf{h}_3 &= \mathbf{a}[\beta_2 + \Omega_2 \mathbf{h}_2] \\
 &\vdots \\
 \mathbf{h}_K &= \mathbf{a}[\beta_{K-1} + \Omega_{K-1} \mathbf{h}_{K-1}] \\
 \mathbf{y} &= \beta_K + \Omega_K \mathbf{h}_K.
 \end{aligned}$$

A **loss function** is needed here,
to measure the difference between the output and truth

Total loss:
$$L = \sum \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i)$$

$$\hat{\mathbf{y}}_i = \beta_K + \Omega_K \mathbf{h}_K(\mathbf{x}_i; [(\beta_0, \Omega_0), \dots, (\beta_{K-1}, \Omega_{K-1})])$$

Find the network parameters to minimize the loss



Loss Optimization

Goal: find the network weight that achieve the lowest loss.

Find the value of the parameters that help the loss function reach the lowest value.

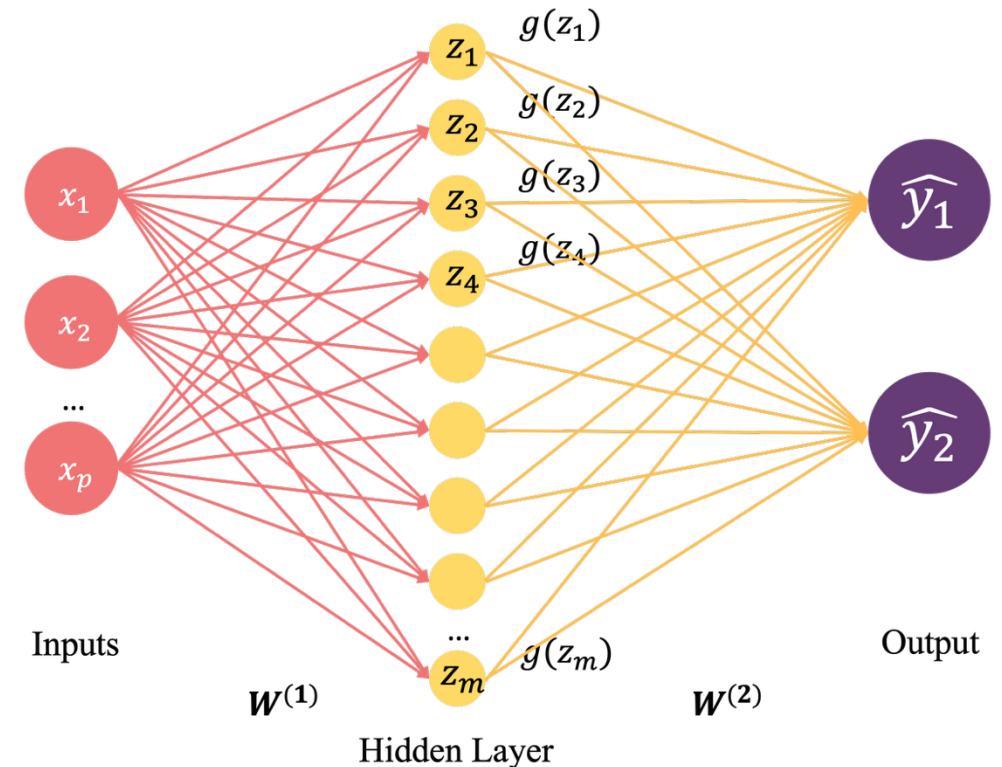
Write this goal in mathematical format:

$$\hat{W} = \underset{W}{\operatorname{argmin}} \mathcal{L}(\underbrace{f(X; W)}_{\text{Prediction}}, \underbrace{y}_{\text{True}})$$

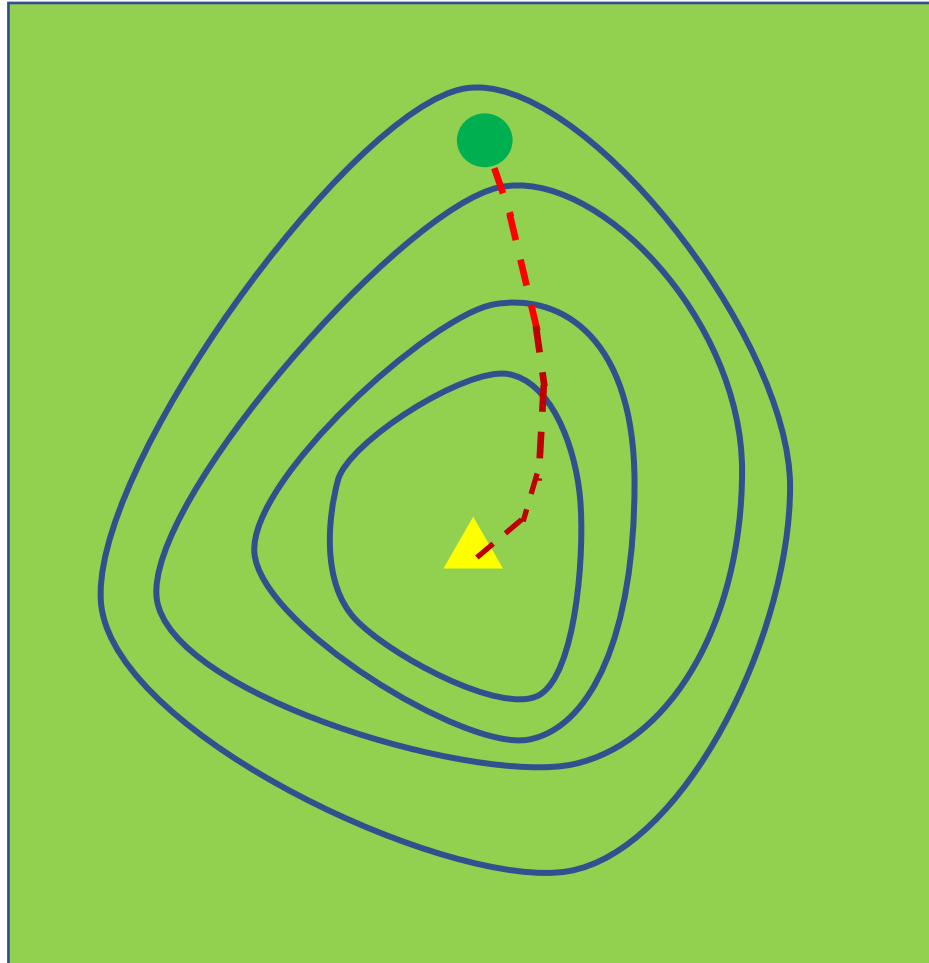
The loss function is a function of the network weights W .

$$W = [W^{(1)}, W^{(2)}, \dots]$$

contains all the weight vectors needed to be adjusted in the neural network



Gradient Descent



$$\hat{W} = \operatorname{argmin} \mathcal{L}(f(\mathbf{X}; \mathbf{W}), \mathbf{y})$$

A first-order iterative optimization algorithm for finding the minimum of a function.

Step 1. Compute the derivatives of the loss w.r.t. the parameters

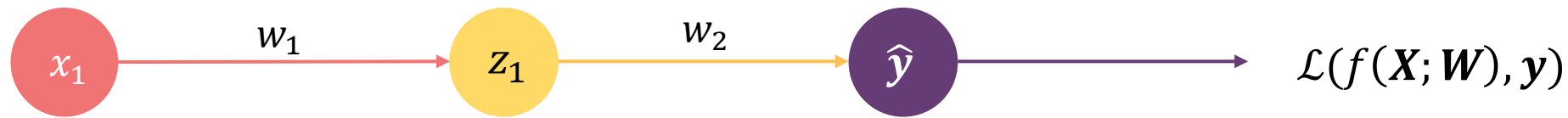
$$\frac{\partial \mathcal{L}(f(\mathbf{X}; \mathbf{W}), \mathbf{y})}{\partial \mathbf{W}}$$

Step 2. Update the parameters according to the rule:

$$\mathbf{w}_{new} = \mathbf{w} - \alpha \frac{\partial \mathcal{L}(f(\mathbf{X}; \mathbf{W}), \mathbf{y})}{\partial \mathbf{W}}$$

where the positive scalar α (learning rate) determines the magnitude of the change.

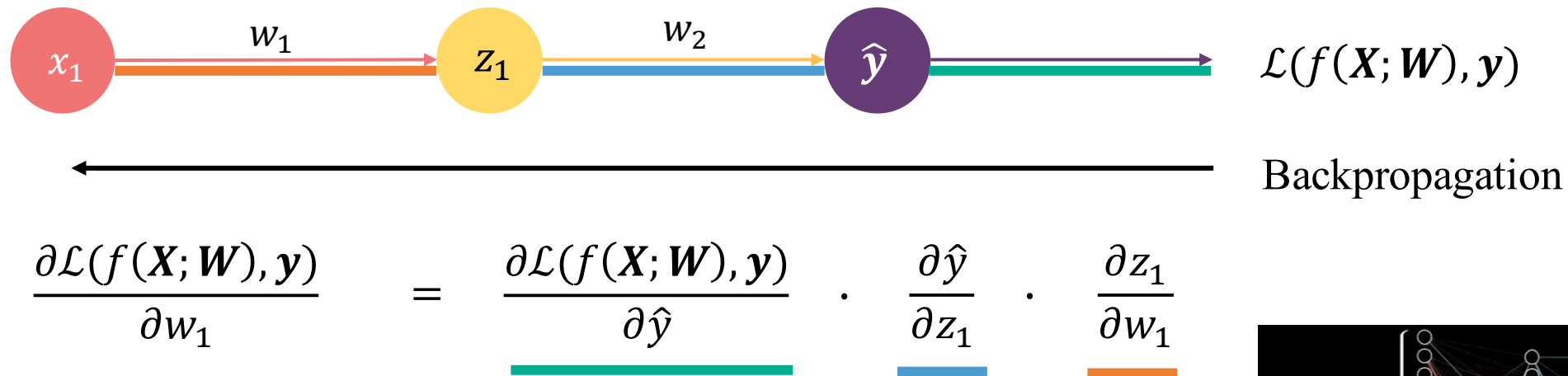
Gradient Computation: Backpropagation



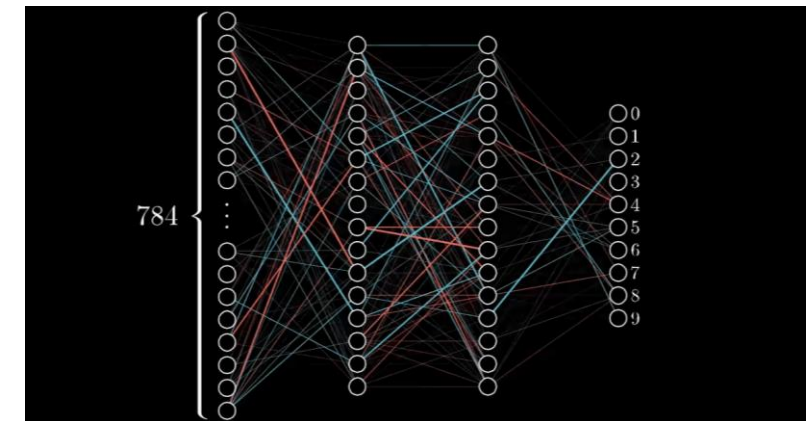
$$\frac{\partial \mathcal{L}(f(\mathbf{X}; \mathbf{W}), \mathbf{y})}{\partial w_1} = \frac{\partial \mathcal{L}(f(\mathbf{X}; \mathbf{W}), \mathbf{y})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1} \quad \text{Chain rule}$$

$$\frac{\partial \hat{y}}{\partial w_1} = \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1} \quad \text{Chain rule again}$$

Gradient Computation: Backpropagation



Repeat this process for each layer, see the visual on the right:



Optimization Algorithms in PyTorch

Stochastic Gradient Descent (SGD)

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

Gradient Descent with Momentum

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

AdaGrad (Adaptive Gradient Algorithm)

```
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.01)
```

Adam (Adaptive Moment Estimation)

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Stochastic Gradient Descent (SGD)

Characteristics:

- Basic form of gradient descent used in neural networks.
- Fixed learning rate.
- In each iteration, randomly select a single data point (or a batch of data points) from the training set to calculate the gradient of the loss function.
- Updates parameters for each training example, leading to frequent updates with high variance.

Batch Size:

Epoch:

Advantages:

- Simple and easy to understand.
- Can escape local minima due to its inherent noise.

Disadvantages:

- Slow convergence on large datasets and high variance in updates.
- Sensitive to learning rate and other hyperparameters.

Gradient Descent with Momentum

Characteristics:

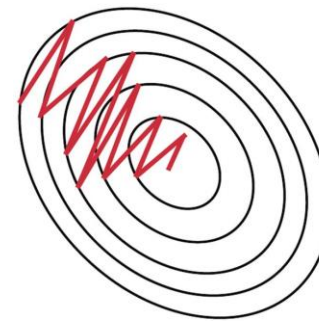
- Builds upon SGD by considering past gradients to smooth out the updates.
- Uses a momentum factor to accelerate SGD in the relevant direction.

Parameter update rule:

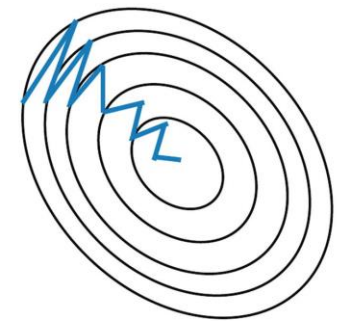
1. Update Velocity: $v = \gamma v - \alpha \nabla f(x)$.
2. Update Parameter: $x = x + v$

Advantages:

- Faster convergence than standard SGD.
- Reduces oscillations and improves stability.



Stochastic Gradient
Descent **without**
Momentum



Stochastic Gradient
Descent **with**
Momentum

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1 - \beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$
$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \mathbf{m}_{t+1},$$

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1 - \beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t - \alpha \beta \cdot \mathbf{m}_t]}{\partial \phi}$$
$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \mathbf{m}_{t+1},$$

AdaGrad (Adaptive Gradient Algorithm)

Parameter update rule:

1. Update accumulation: $G = G + g^2$, where g is the gradient of the loss function with respect to each parameter.
2. Adjust Learning Rate: Scale the learning rate for each parameter inversely proportional to the square root of G .
3. Update Parameters: Update the parameters using the adjusted learning rate, $x = x - \frac{\alpha}{\sqrt{G + \epsilon}} \cdot g$
where α is the initial learning rate, ϵ is a small constant added to improve numerical stability.

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \frac{\partial L[\phi_t]}{\partial \phi} \\ \mathbf{v}_{t+1} &\leftarrow \left(\frac{\partial L[\phi_t]}{\partial \phi} \right)^2.\end{aligned}$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1} + \epsilon}},$$

AdaGrad (Adaptive Gradient Algorithm)

Characteristics:

- Adjusts the learning rate to each parameter, decreasing it for parameters with large gradients.
- Each parameter has its own learning rate, which can be beneficial for datasets with features of varying importance or scale.

Advantages:

- The effective learning rate decreases over time for each parameter. Eliminates the need to manually tune the learning rate.
- Well-suited for dealing with sparse features or data with different scales.

Disadvantages:

- The continuously accumulating squared gradient can lead to an excessively reduced learning rate, causing the algorithm to stop learning too early.

Adam (Adaptive Moment Estimation)

Parameter update algorithm:

1. Moving averages: two vectors m and v are used to store moving averages of the gradients and squared gradients, both initialized to zero.

2. Hyperparameters: β_1 and β_2 , close to 1 (common defaults are 0.9 and 0.999).

3. Update Moving Averages: $m = \beta_1 m + (1 - \beta_1)g$ and $v = \beta_2 v + (1 - \beta_2)g^2$.

4. Correct Bias: $\hat{m} = \frac{m}{1 - \beta_1^t}$ and $\hat{v} = \frac{v}{1 - \beta_2^t}$.
$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1 - \beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

5. Adjust parameters: $x = x - \frac{\alpha}{\sqrt{\hat{v} + \epsilon}} \hat{m}$
$$\mathbf{v}_{t+1} \leftarrow \gamma \cdot \mathbf{v}_t + (1 - \gamma) \left(\sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi} \right)^2$$

where α is the initial learning rate, ϵ is a small constant added to improve numerical stability.

$$\begin{aligned} \tilde{\mathbf{m}}_{t+1} &\leftarrow \frac{\mathbf{m}_{t+1}}{1 - \beta^{t+1}} & \phi_{t+1} &\leftarrow \phi_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1} + \epsilon}} & \phi_{t+1} &\leftarrow \phi_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1} + \epsilon}}, \\ \tilde{\mathbf{v}}_{t+1} &\leftarrow \frac{\mathbf{v}_{t+1}}{1 - \gamma^{t+1}}. \end{aligned}$$

Adam (Adaptive Moment Estimation)



Diederik P. Kingma

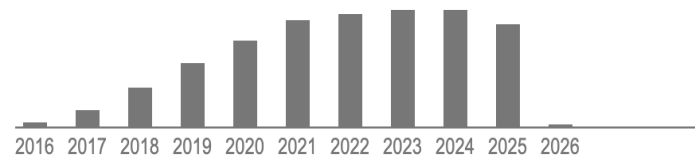
Adam: A method for stochastic optimization

Authors Diederik P Kingma, Jimmy Ba

Publication date 2014/12/22

Journal arXiv preprint arXiv:1412.6980

Total citations Cited by 239290



Announcing the Test of Time Award Winners from ICLR 2015

CARL VONDRICK / ICLR 2025

We are honored to announce the Test of Time awards for ICLR 2025. This award recognizes papers published ten years ago at ICLR 2015 that have had a lasting impact on the field. The 2025 program chairs and general chair reviewed the papers published at ICLR 2015, and selected the two papers below for their profound influence and impact on machine learning today.

Congratulations to the authors of the Test of Time winner and runner up!

Test of Time

Adam: A Method for Stochastic Optimization

Diederik P. Kingma, Jimmy Ba

<https://arxiv.org/abs/1412.6980>

Adam (Adaptive Moment Estimation)

Characteristics:

- Designed to combine the advantages of two other popular optimizers: the adaptive learning rate feature of AdaGrad and the momentum feature of RMSprop.
- Different learning rates for different parameters and adjusts them throughout training.
- Corrects the bias in moving averages, especially important in the initial training phase.

Advantages:

- Combines the benefits of AdaGrad and RMSprop.
- Performs well in practice and across a wide range of non-convex optimization problems and large dataset.

Disadvantages:

- Can be memory-intensive due to storing moving averages for each parameter.
- Might not converge to the optimal solution in certain theoretical cases.

Optimization

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

➤ Batch SGD

➤ Momentum

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1 - \beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi} \\ \phi_{t+1} &\leftarrow \phi_t - \alpha \cdot \mathbf{m}_{t+1}, \end{aligned}$$

$$\begin{aligned} \mathbf{f}_0 &= \beta_0 + \Omega_0 \mathbf{x}_i \\ \mathbf{h}_k &= \mathbf{a}[\mathbf{f}_{k-1}] \\ \mathbf{f}_k &= \beta_k + \Omega_k \mathbf{h}_k \end{aligned}$$

Forward Pass

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1 - \beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi} & \tilde{\mathbf{m}}_{t+1} &\leftarrow \frac{\mathbf{m}_{t+1}}{1 - \beta^{t+1}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1 - \gamma) \left(\sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi} \right)^2 & \tilde{\mathbf{v}}_{t+1} &\leftarrow \frac{\mathbf{v}_{t+1}}{1 - \gamma^{t+1}} \end{aligned}$$

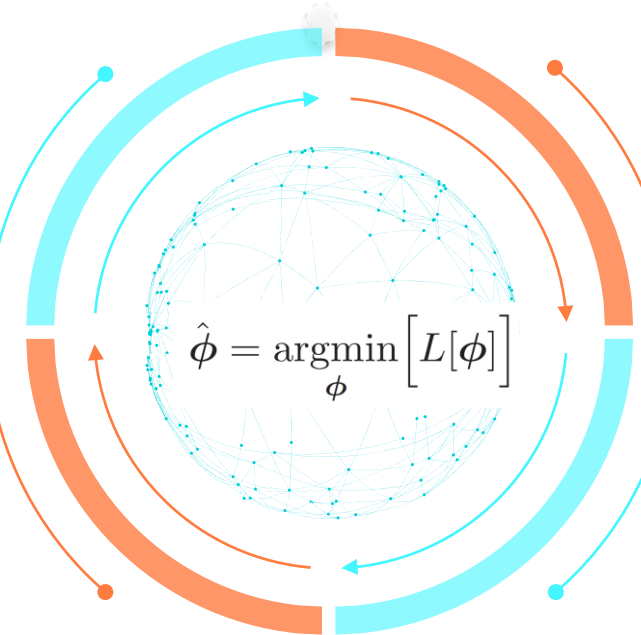
$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}$$

➤ Adaptive Moment Estimation (Adam)

➤ Backpropagation algorithm

$$\begin{aligned} \frac{\partial \ell_i}{\partial \beta_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} & \frac{\partial \ell_i}{\partial \beta_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \\ \frac{\partial \ell_i}{\partial \Omega_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T & \frac{\partial \ell_i}{\partial \Omega_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \mathbf{x}_i^T \\ \frac{\partial \ell_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\Omega_k^T \frac{\partial \ell_i}{\partial \mathbf{f}_k} \right) \end{aligned}$$

Backward Passes



Backpropagation Algorithm

2 Steps:

1. Forward Propagation

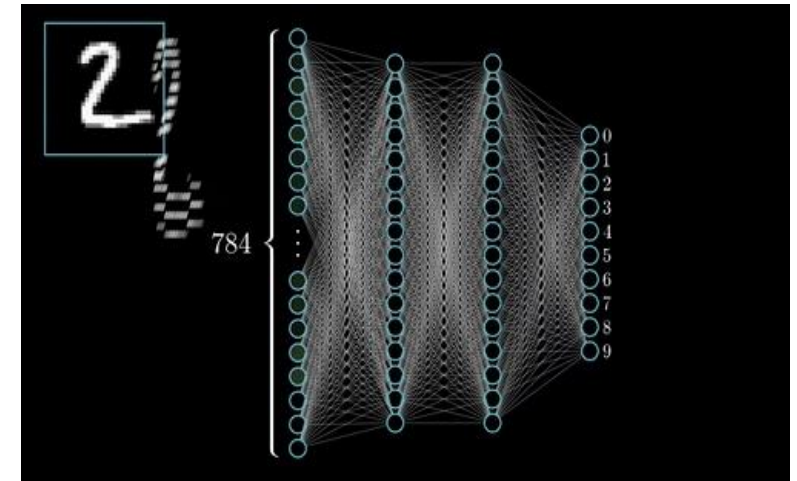
- Forward propagation is **how neural networks make predictions**.
- Involves passing input data through the network layer by layer to the output.

2. Backpropagation

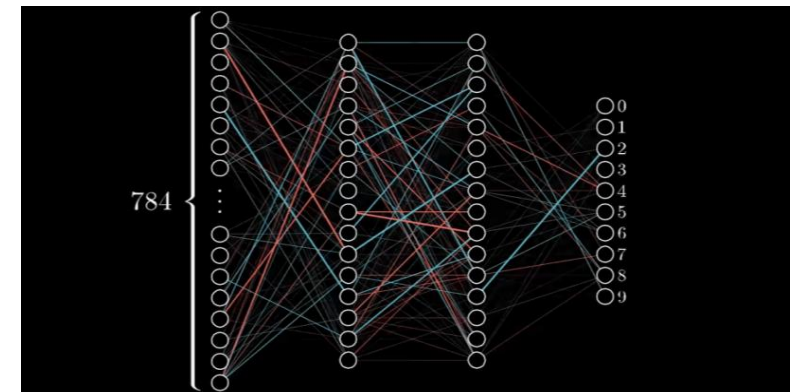
- Backpropagation is the process of adjusting the weights of the network by propagating through the neural network **backward**.
- Involves calculating the gradient of the loss function with respect to each weight by the chain rule.
- The weights are adjusted in the direction that reduces the loss.

Both steps are iteratively repeated for several epochs to minimize the loss and improve the model's accuracy.

Forward Propagation



Backpropagation

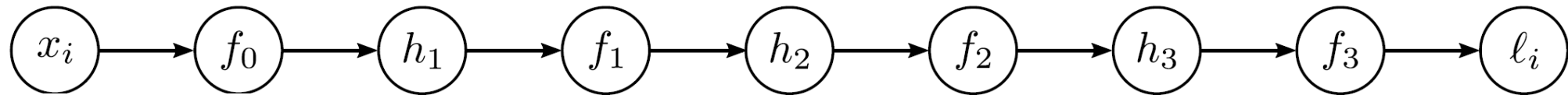


Backpropagation Algorithm

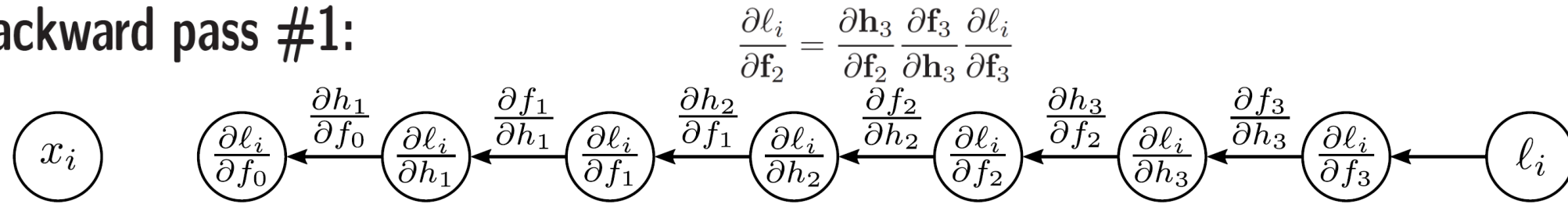
$$\ell_i = (f[x_i, \phi] - y_i)^2 \quad f[x, \phi] = \beta_3 + \omega_3 \cdot \cos[\beta_2 + \omega_2 \cdot \exp[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x]]]$$

Forward pass:

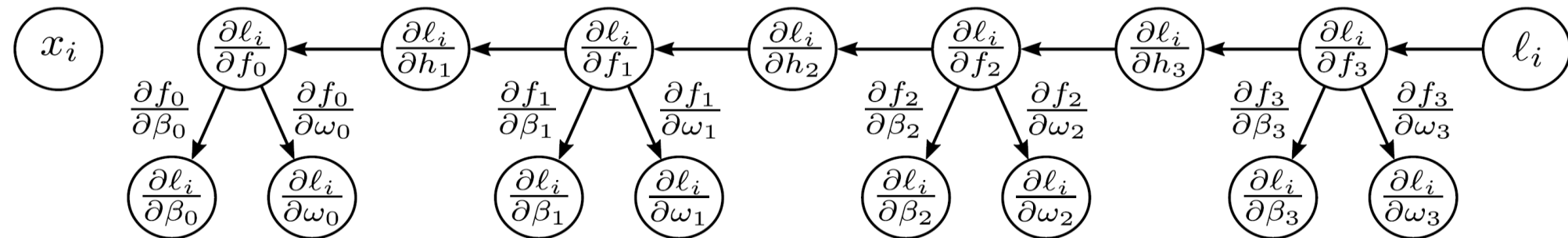
$$\begin{array}{llll} f_0 = \beta_0 + \Omega_0 x_i & f_1 = \beta_1 + \Omega_1 h_1 & f_2 = \beta_2 + \Omega_2 h_2 & f_3 = \beta_3 + \Omega_3 h_3 \\ h_1 = a[f_0] & h_2 = a[f_1] & h_3 = a[f_2] & \ell_i = l[f_3, y_i], \end{array}$$



Backward pass #1:



Backward pass #2:



Batch Normalization

- **Definition:** Batch Normalization (BN) is a technique used in deep learning to normalize the inputs to each layer within a neural network. It ensures that the inputs have a consistent distribution, which stabilizes and accelerates training.
- **Purpose:** **Reduce internal covariate shift:** This occurs when the distribution of inputs to a layer changes during training.
- **Benefits:**
 - a) **Improved Stability:** Keeps activations in a stable range, mitigating vanishing/exploding gradients.
 - b) **Faster Convergence:** Allows for higher learning rates and reduces sensitivity to initialization.
 - c) **Regularization Effect:** Adds noise due to batch statistics, reducing overfitting.
 - d) **Enhanced Generalization:** Produces better results on unseen data.

<https://kharshit.github.io/blog/2018/12/28/why-batch-normalization>

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

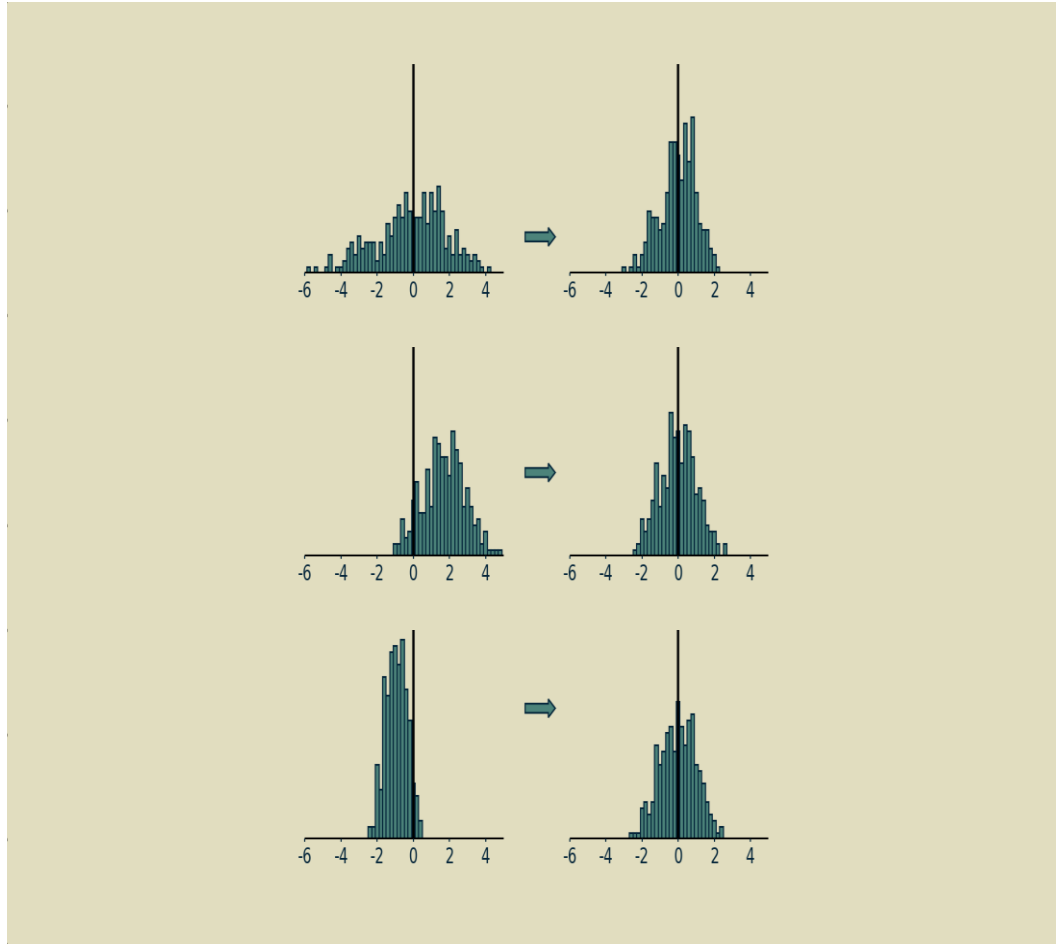
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch Normalization

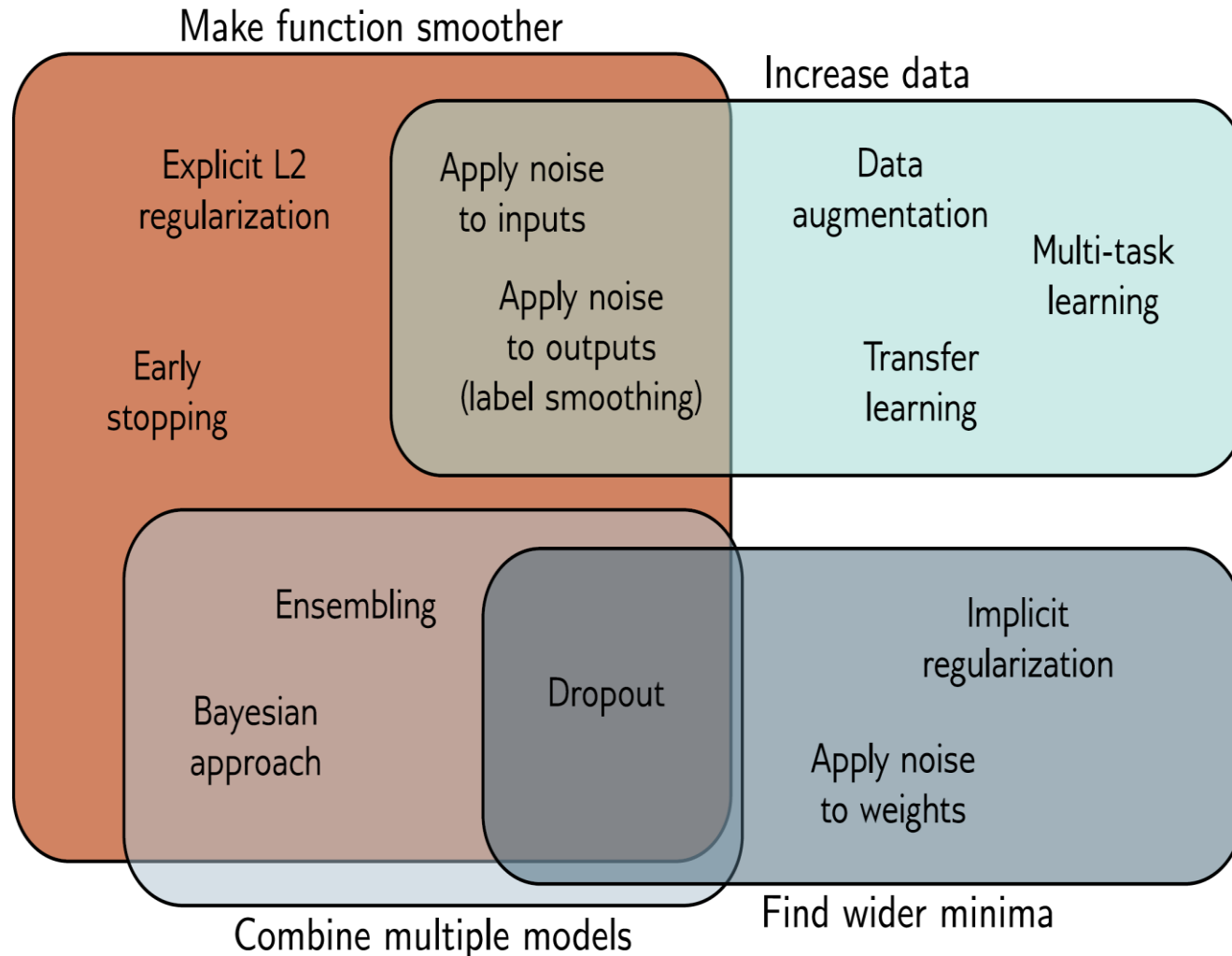


https://e2eml.school/batch_normalization



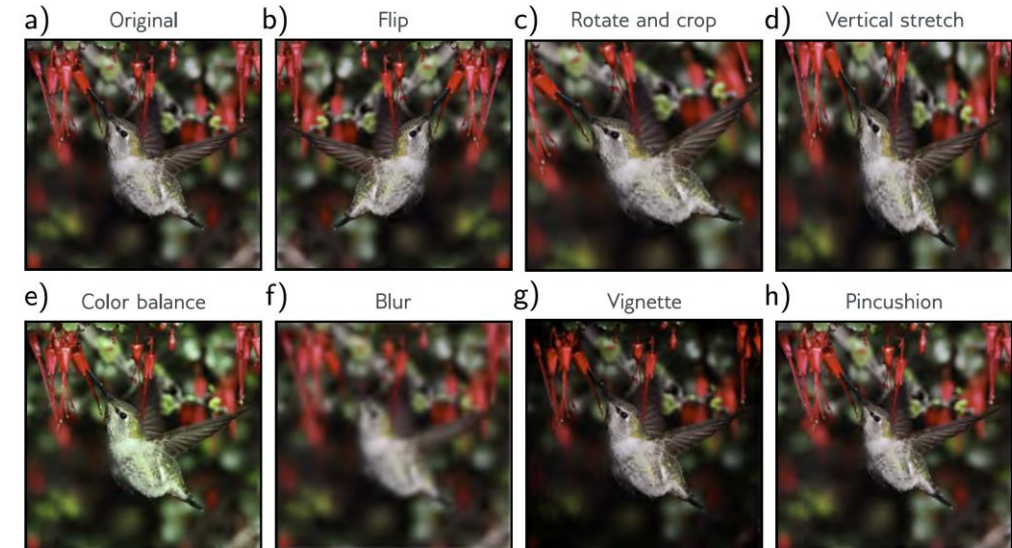
<https://kharshit.github.io/blog/2018/12/28/why-batch-normalization>

Regularization Methods



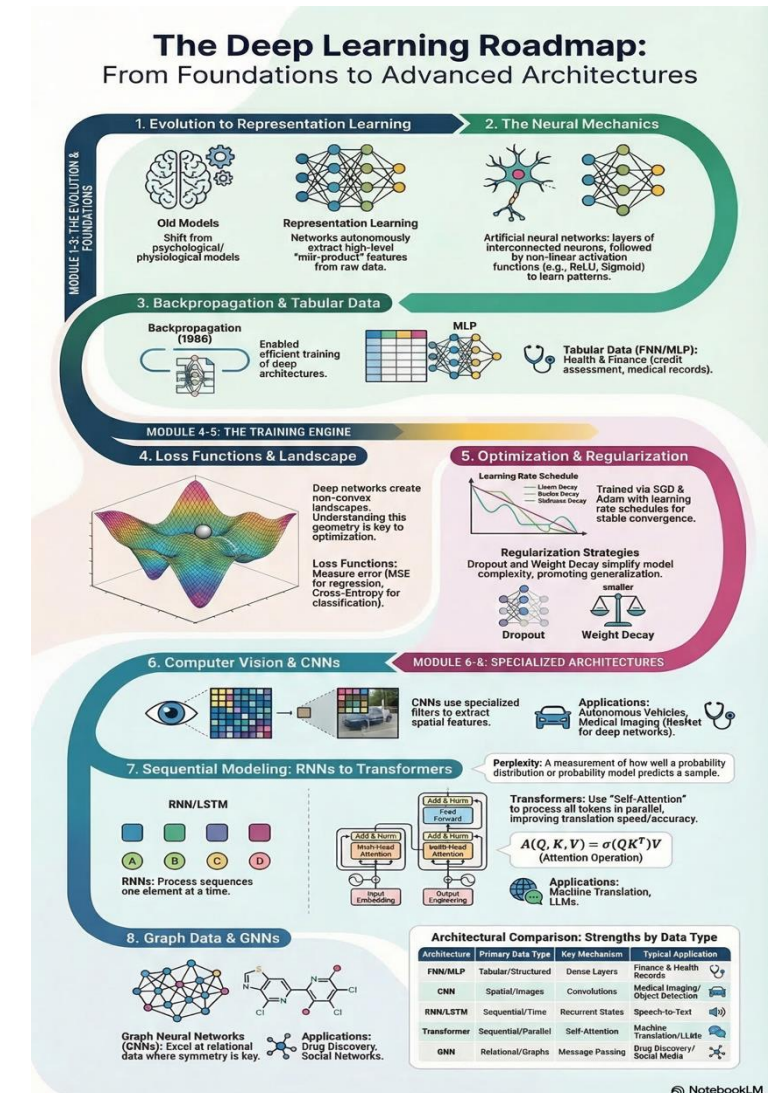
Four Mechanisms:

- ❖ Make the modeled function smoother.
- ❖ Increase the effective amount of data.
- ❖ Combine multiple models to mitigate uncertainty in the fitting process.
- ❖ Encourages the training process to converge to a wide minimum, where small errors in the estimated parameters are less important.

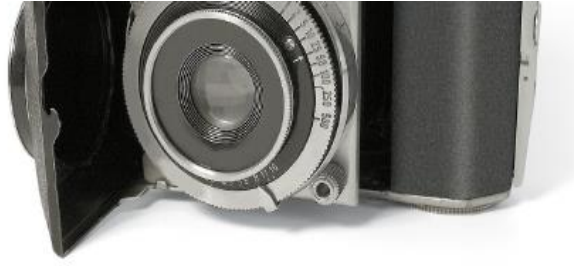


Content

5 Computer Vision and Convolutional Neural Networks (CNN)



Nature Image Data is Everywhere



Major CV Tasks

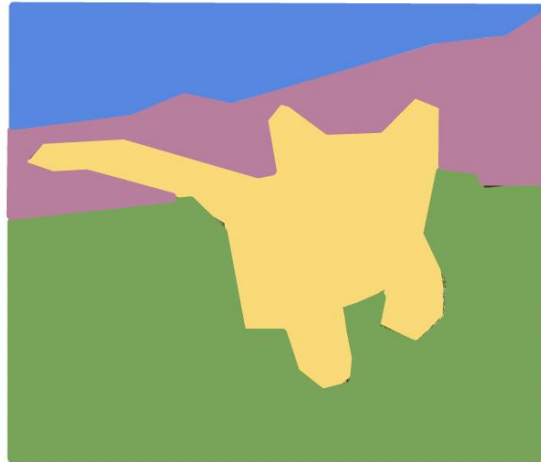
Classification



CAT

No spatial extent

Semantic Segmentation



GRASS, CAT, TREE,
SKY

No objects, just pixels

Object Detection



DOG, DOG, CAT

Multiple Object

Instance Segmentation



DOG, DOG, CAT

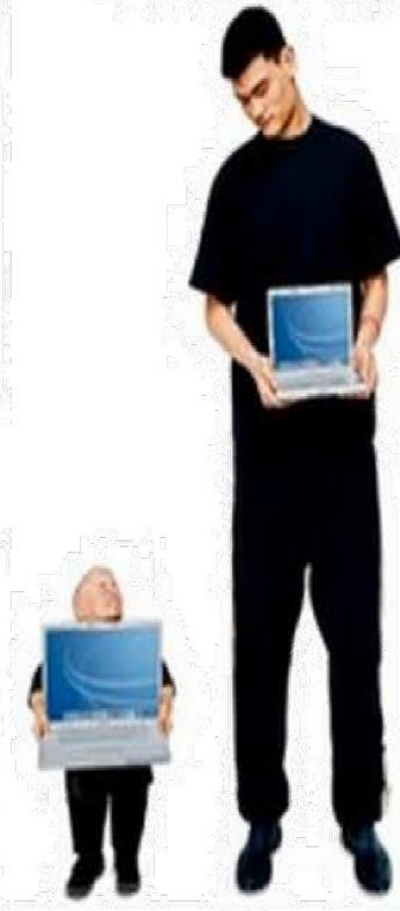
[This image is CC0 public domain](#)

Scenario Challenges

Viewpoint variation



Scale variation



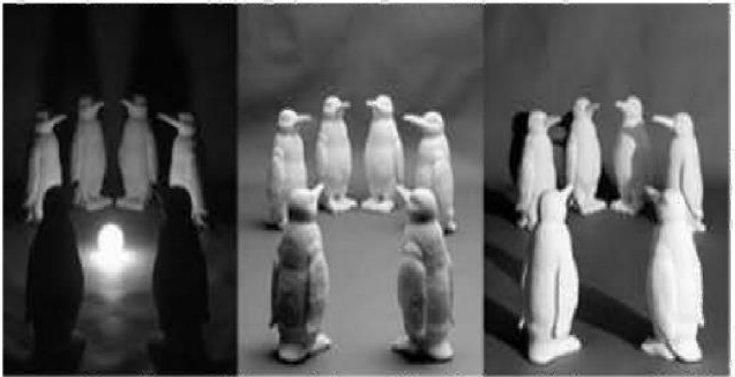
Deformation



Occlusion



Illumination conditions



Background clutter



Intra-class variation



Why CNNs? Limitations of Fully Connected Networks for Images

1. High Dimensionality

A $224 \times 224 \times 3$ image has 150,528 inputs; a shallow FC layer needs ~22 billion weights → overfitting, huge memory, slow training.

CNN fix: shared weights (convolutional filters) applied to local regions (kernels).

2. Spatial Relationships

Nearby pixels form local patterns (edges, corners, textures), but FC networks treat all pixels equally and ignore locality.

CNN fix: local receptive fields preserve spatial coherence and capture local patterns.

3. Geometric Transformations

A shifted, rotated, or flipped object keeps its identity, yet FC networks must relearn patterns at every position → inefficient, data-hungry.

CNN fix: translation invariance via shared filters, plus data augmentation in training.

4. Noise, Scale & Hierarchy

Real images are noisy and contain features at multiple scales (edges → textures → objects); FC networks cannot represent this hierarchy.

CNN fix: stacked convolutional layers with increasing receptive fields; robust local features.

Takeaway: CNNs leverage **shared weights, local receptive fields, and hierarchical feature extraction** to process high-dimensional images efficiently and robustly.

ImageNet

What is ImageNet?

- **Definition:** ImageNet is a large-scale visual database designed to advance research in object detection, classification, and other computer vision tasks.
- **Dataset Size:** It contains over **14 million labeled images** spanning **20,000+ categories**, with the most commonly used subset having **1,000 object categories**.

Key Features of ImageNet

a) Diversity of Classes:

Includes both broad categories (e.g., "dog," "car") and fine-grained subcategories (e.g., "golden retriever," "sports car").

b) Real-World Images:

Images collected from the internet represent real-world complexity, including cluttered backgrounds, occlusions, and multiple objects.

c) Hierarchical Organization:

Based on the WordNet hierarchy, where classes are semantically related, providing meaningful relationships between categories.

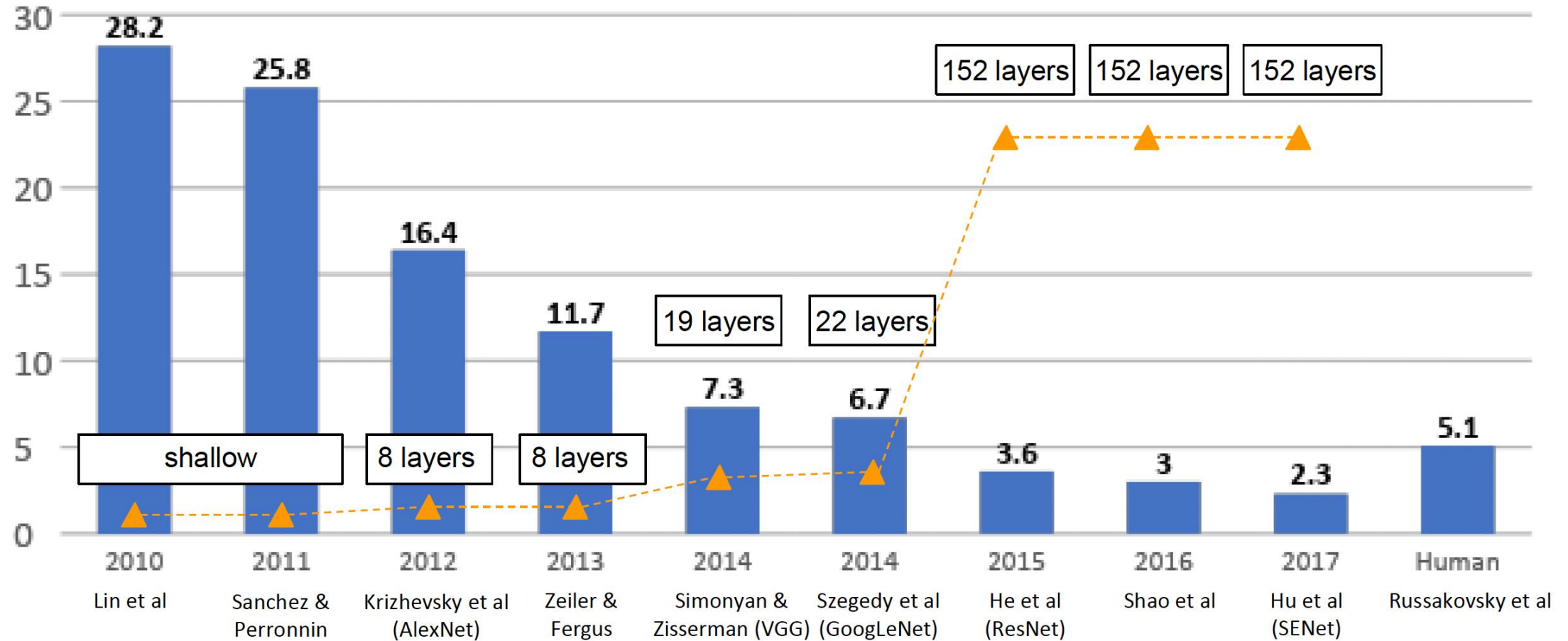


[This image is CC0 1.0 public domain](#)



[This image is CC0 1.0 public domain](#)

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Modern DL Model Architectures

Convolutional Neural Networks (CNNs)

- **Key Features:** Utilizes convolutional layers to process data in a grid pattern (like images).
- **Key Components:**
 - Convolutional Layers: Extract features from input images using filters.
 - Pooling Layers: Reduce dimensions and computational load, retaining key information.
 - Fully Connected Layers: Classify images based on extracted features.
- **Example Models:** LeNet-5, AlexNet, VGGNet.

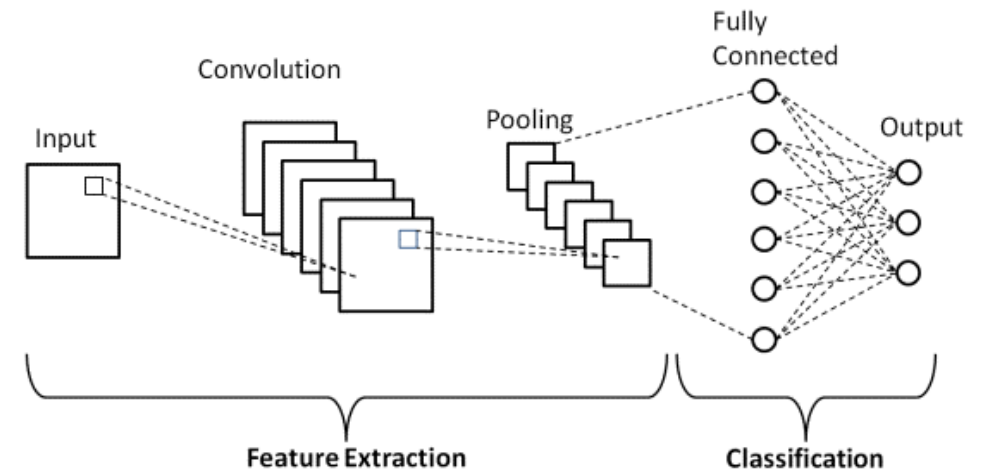
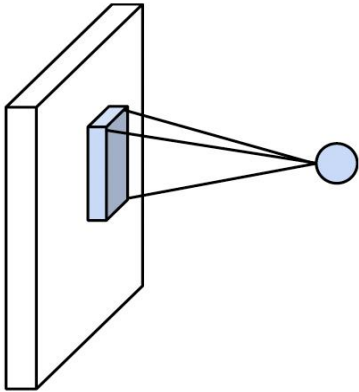


Figure. Basic CNN structure.

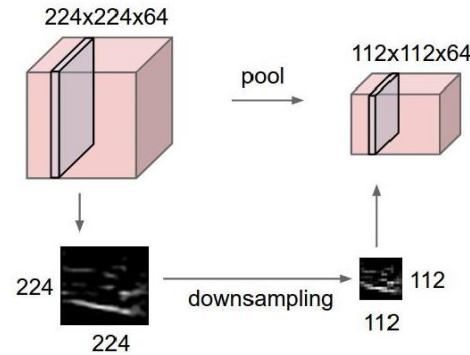
- **Applications in Biomedicine:**
 - Image classification in diagnostics (e.g., cancer detection from scans).
 - Image segmentation for identifying regions of interest in medical images.

Key Components of CNNs

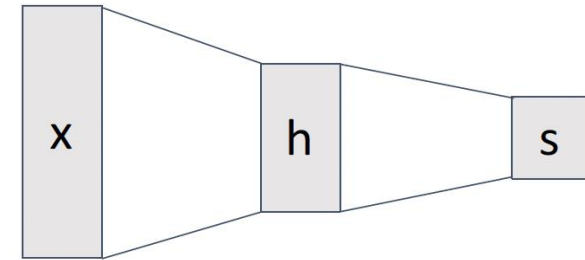
Convolution Layers



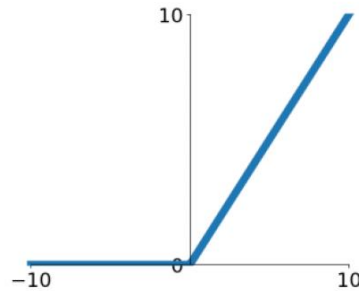
Pooling Layers



Fully-Connected Layers



Activation Function



Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Key Components of CNN

- Convolutional layers
- Rectified Linear Unit (ReLU)
- Pooling layers
- Fully connected layers

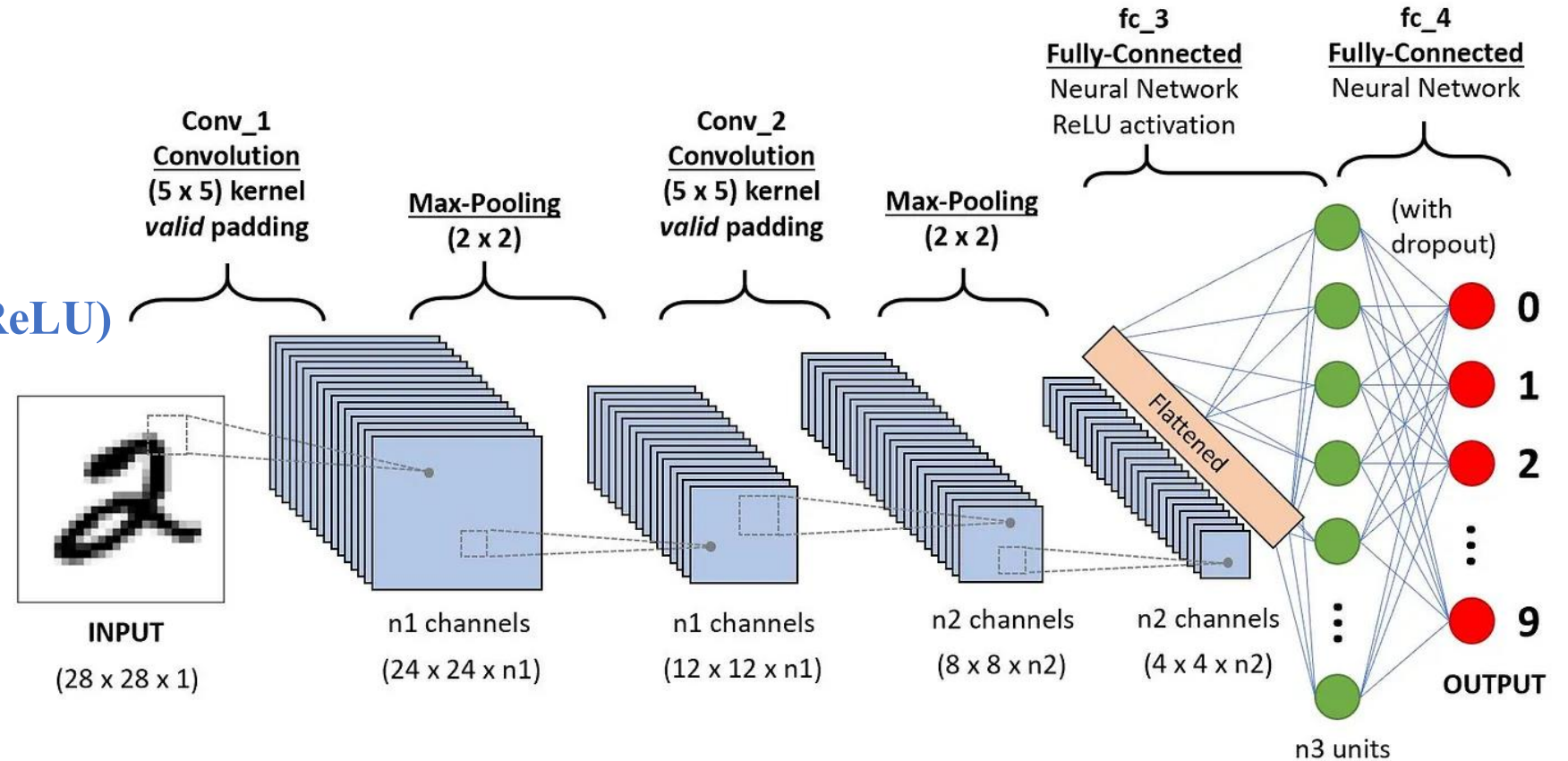


Illustration of architecture of CNNs applied to digit recognition ([source](#))

Feature Extraction Using Convolution

- **Input, kernel, and output (right figure)**
- **Fully Connected Networks**
- **Locally Connected Networks**
 - A simple solution to this problem is to limit connections between hidden and input units, allowing each hidden unit to connect to only a small subset of input units, such as a contiguous region of pixels. For other data types different than images like audio, hidden units can be connected to specific time spans. This concept of local connections is inspired by the visual cortex, where neurons respond to stimuli in specific locations.

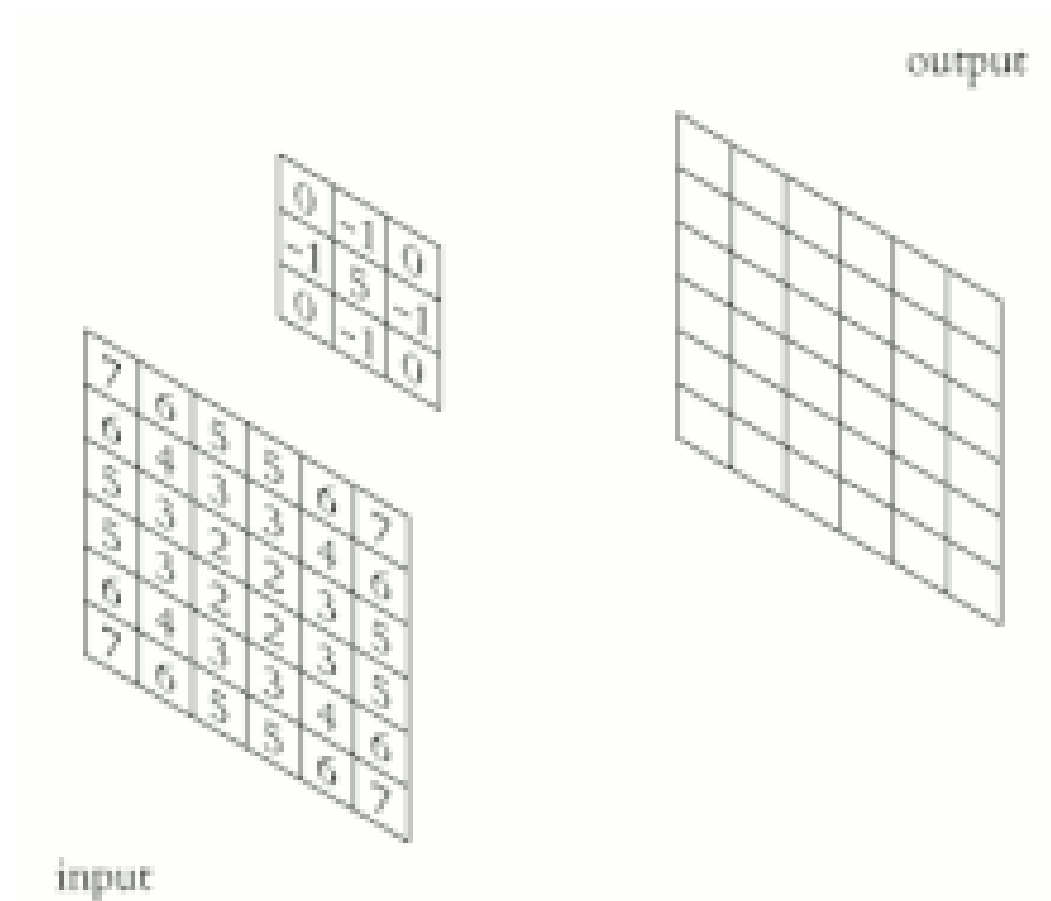


Illustration of Discrete 2D Convolution ([source](#))

Understanding the Convolution Operation

What is convolution?

Mathematically, Convolution is defined as $f, g: \mathbb{R}^n \rightarrow \mathbb{R}$:

$$(f * g)(x) = \int f(z)g(x - z)dz$$

Whenever we have discrete objects, the integral turns into a sum. For instance, in CNN, we used discrete convolution for vectors from the set of square-summable infinite-dimensional vectors defined as:

$$(f * g)(i) = \sum_a f(i)g(i - a)$$

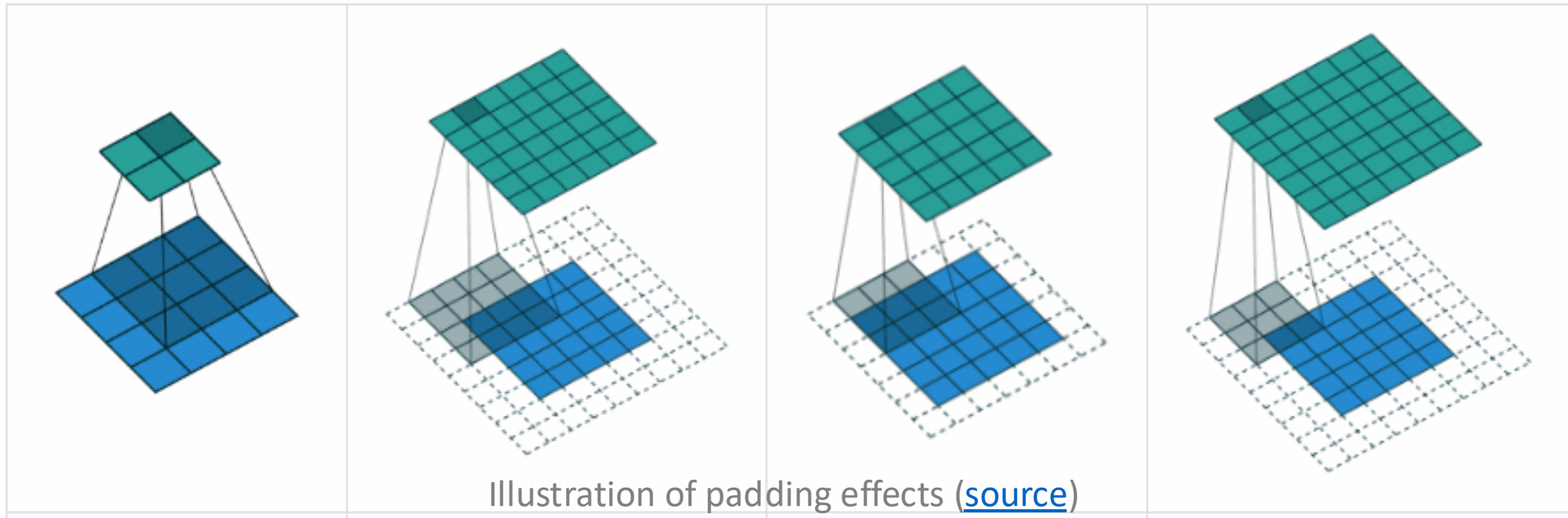
For two-dimensional tensors, we have a corresponding sum with (a,b) for f $(i-a,j-b)$ for g , respectively:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i - a, j - b)$$

Padding, Stride, and Pooling

- **Padding**

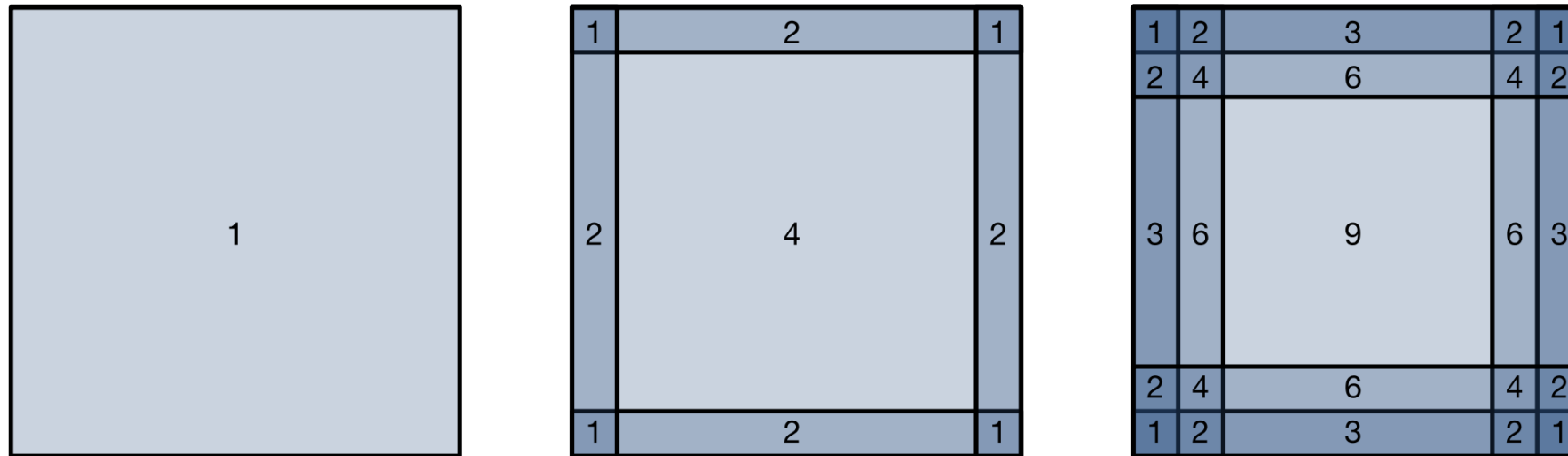
- Zero-padding and why it's necessary (The pixels at the corner in the previous images are less counted than those in the middle)
- How padding affects the dimensions of the output



Padding

One tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image. The following figure depicts **the pixel utilization as a function of the convolution kernel size and the position within the image**.

We can see that the pixels in the corners are hardly used at all.

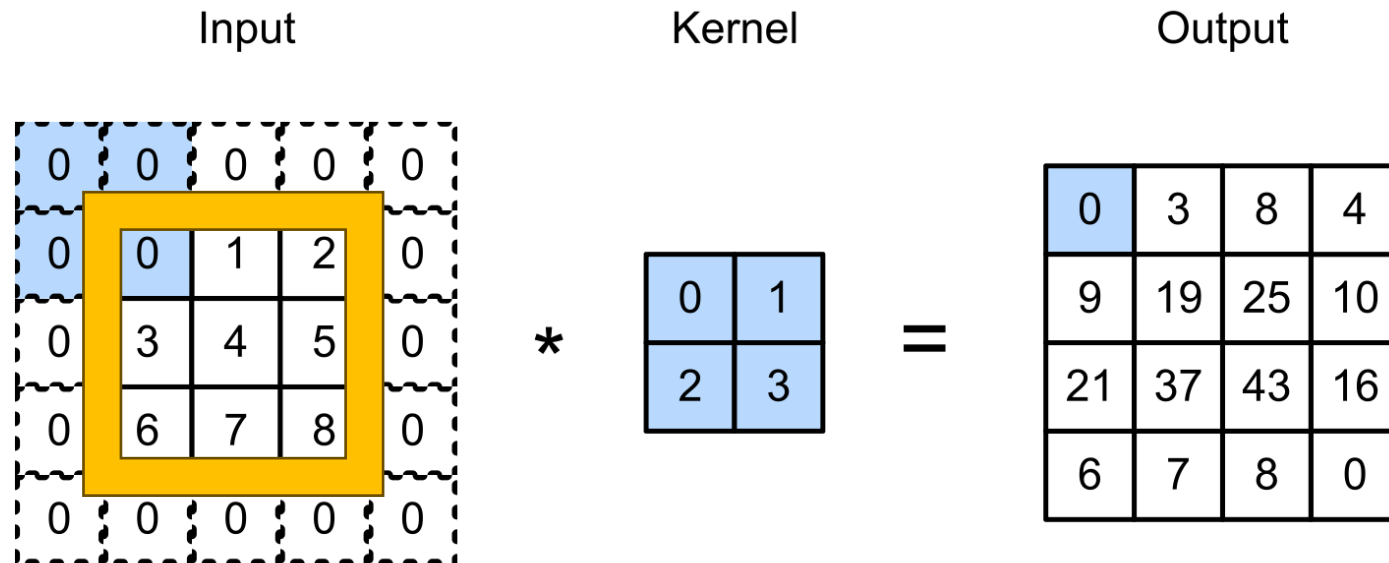


Pixel utilization for convolutions of 1x1, 2x2, and 3x3 respectively.

Padding

One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Typically, we set the values of the extra pixels to zero.

Example on padding 3x3 input to 5x5 matrix:



Padding, Stride, and Pooling

Stride

- Example with stride of 1 vs. 2

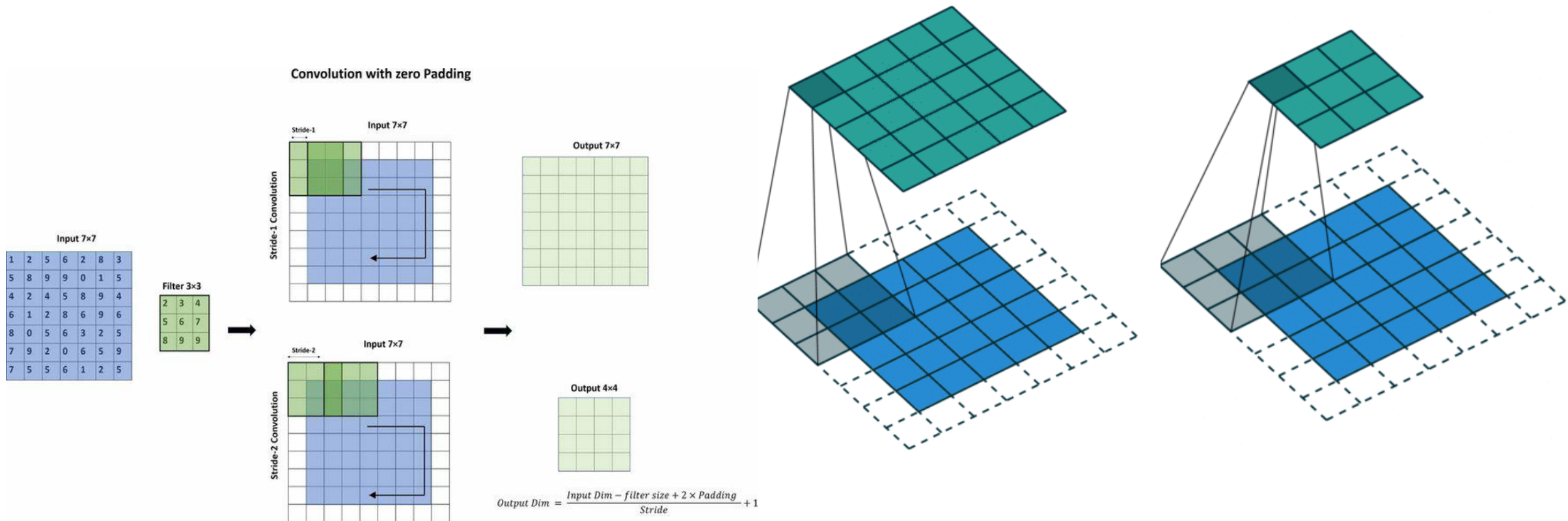
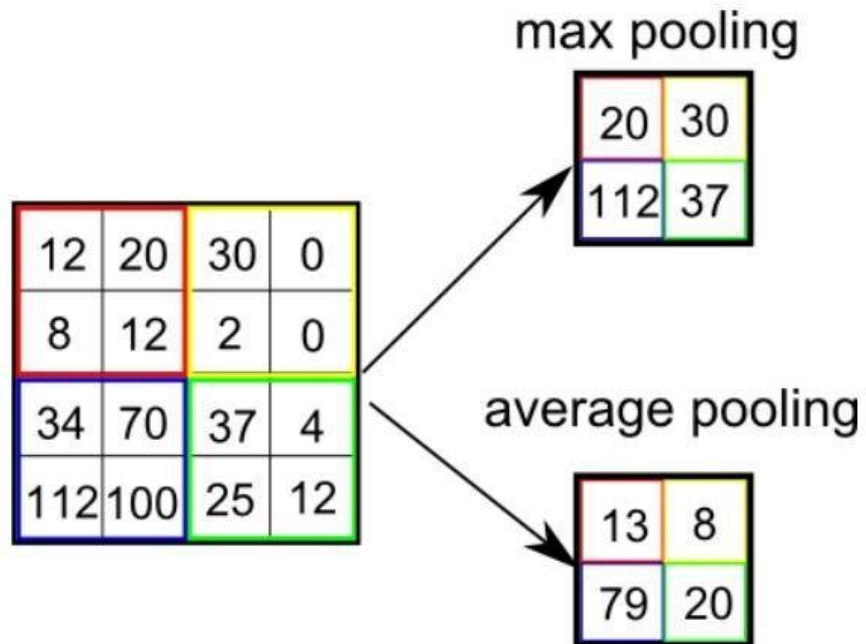


Illustration Convolution Operation with Stride Length = 1 Vs 2 ([source](#))

Padding, Stride, and Pooling

- **Pooling**

- Types: Max pooling, average pooling
- Role in reducing dimensionality
- Example: Pooling on an image



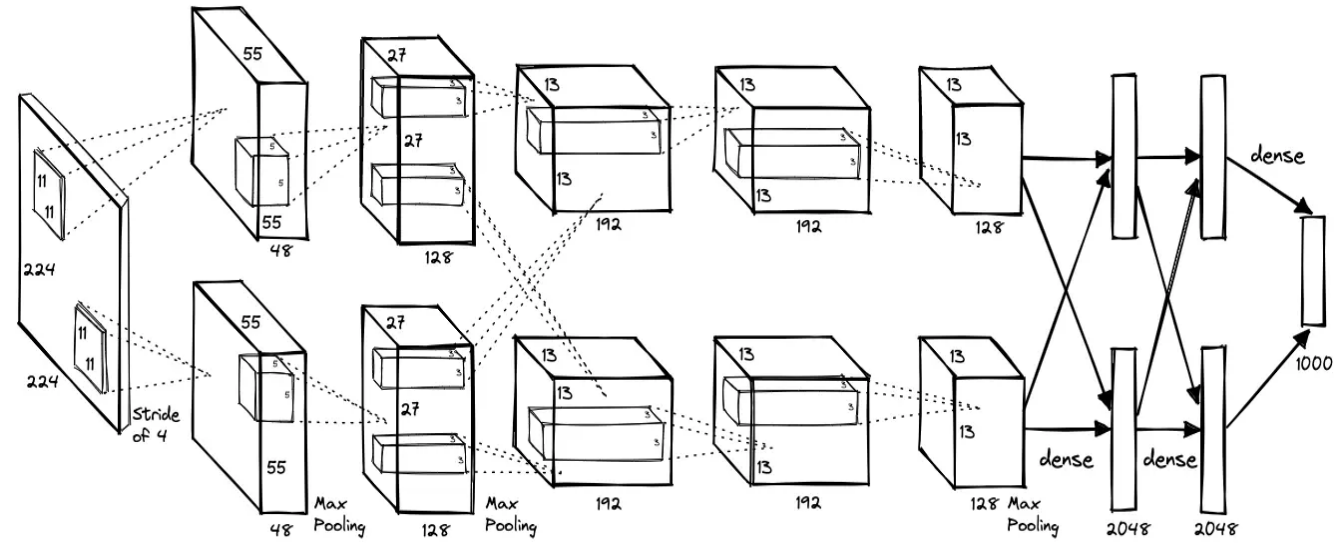
3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

Illustration of 3x3 pooling over 5x5 convolved feature ([source](#))

AlexNet

- With high performance hardware (GPUs from Nvidia) and sufficiently rich data-set, Krizhevsky et al. proposed AlexNet (Alom et al. 2018), which consists of **five convolution layers and three fully connected layers**.
- Each convolution layer contains a convolution kernel, a bias term, a ReLU activation function, and a local response normalization (LRN) module.
- In the 2012 ILSVRC, AlexNet won the competition with a Top-5 classification error rate of **16.4%**, became the dividing line between traditional and deep learning algorithms, and was the first deep CNN model in modern times.



Layer	Type	Filter/Kernel Size	Number of Filters	Stride	Output Size
Input	RGB Image Input	-	-	-	$227 \times 227 \times 3$
Layer 1	Convolution + Max Pooling	11×11	96	4	$55 \times 55 \times 96$
Layer 2	Convolution + Max Pooling	5×5	256	1	$27 \times 27 \times 256$
Layer 3	Convolution	3×3	384	1	$13 \times 13 \times 384$
Layer 4	Convolution	3×3	384	1	$13 \times 13 \times 384$
Layer 5	Convolution + Max Pooling	3×3	256	1	$13 \times 13 \times 256$
Layer 6	Fully Connected	-	4096	-	4096
Layer 7	Fully Connected	-	4096	-	4096
Layer 8	Fully Connected (Output)	-	1000	-	1000 (class probabilities)

Table 1: Architecture of AlexNet (Rotated Table).

Residual network (ResNet)

- **Degradation Problem:** Deeper networks (e.g., >20 layers) suffered from degradation of accuracy, not just overfitting, but actual performance decline.
- **Key Idea:** Instead of learning the direct mapping ($H(x)$), ResNet learns the **residual mapping** ($F(x)=H(x)-x$). This simplifies optimization and allows gradients to flow through skip connections, improving convergence.
- **Impact:**
 - **Ease of Optimization:** Learning residuals is simpler than learning direct mappings.
 - **Deeper Architectures:** ResNet-152 outperforms shallower networks while maintaining high accuracy.
 - **State-of-the-art Results:** Top-5 error dropped to **~3.6%** on ImageNet (ILSVRC).
- **Connection to Highway Networks (Srivastava et al., 2015):** ResNet can be seen as a special, simplified case of highway layers where gates are mostly open.
- Residual connections enable building much deeper and more powerful networks by addressing gradient vanishing and “degradation” issues.

Residual Block

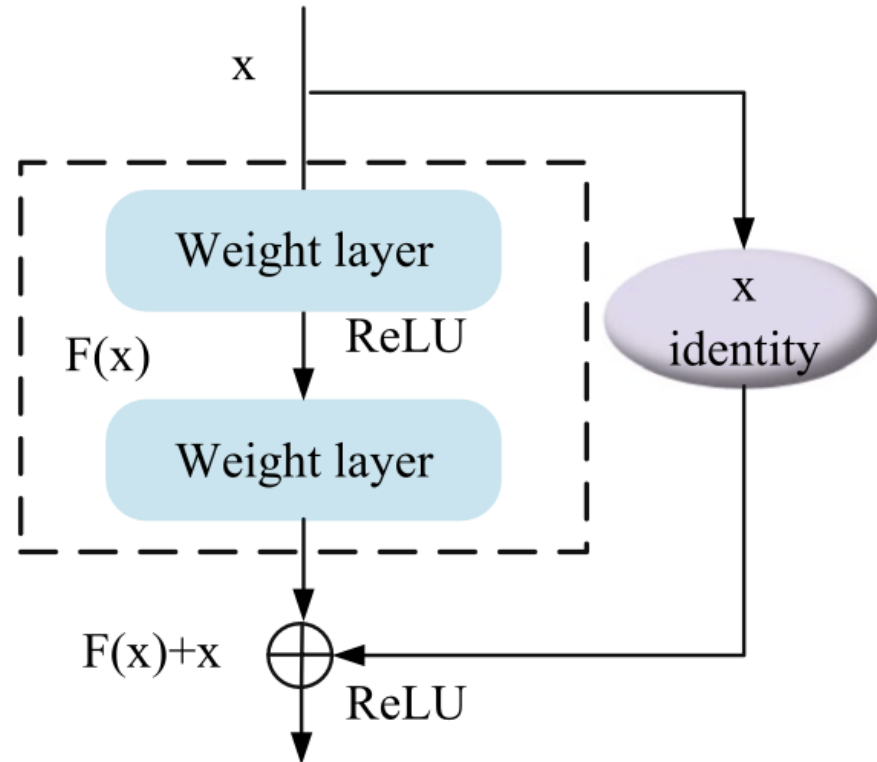


Illustration of a residual block

Zhao et al. *Artificial Intelligence Review* (2024)

Building Block:

- a) **Input:** x (feature map from the previous layer).
- b) **Path 1 (Residual Function):**
 - i. 3×3 convolution $\rightarrow$ Batch Normalization $\rightarrow$ ReLU.
 - ii. 3×3 convolution $\rightarrow$ Batch Normalization.
- c) **Path 2 (Skip Connection):**
 - i. Identity mapping: Directly passes the input x .
- d) **Addition:**
 - i. Output: $F(x) + x$ (summation of the two paths).
- e) **Activation:**
 - i. Apply ReLU to the combined output.
- f) **Output:**
Final feature map retains the same dimensions as the input.

Why is ResNet so popular?



Kaiming He

Deep Residual Learning for Image Recognition

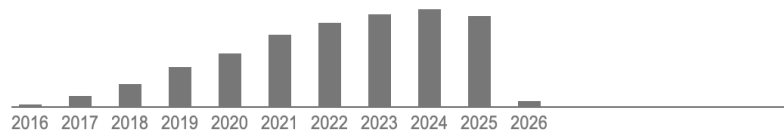
Authors Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

Publication date 2016

Conference Computer Vision and Pattern Recognition (CVPR), 2016

Description Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate residual nets with a depth of up to 152 layers---8x deeper than VGG nets but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test set. This result won the 1st place on the ILSVRC 2015 classification task. We also present analysis on CIFAR-10 with 100 and 1000 layers. The depth of representations is of central importance for many visual recognition tasks. Solely due to our extremely deep representations, we obtain a 28% relative improvement on the COCO object detection dataset. Deep residual nets are foundations of our submissions to ILSVRC & COCO 2015 competitions, where we also won the 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.

Total citations Cited by 310651



❖ Solving the "Depth Barrier"

❖ Residual Learning Philosophy

❖ Foundations of Modern Architectures

➤ The "Industry Backbone Inherited by Transformers

➤ Stable Convergence

❖ Efficiency & Scalability

➤ **Parameter Economy:** offering higher accuracy with fewer weights.

➤ **Transfer Learning:** Pre-trained ResNet weights are the most widely available and reliable starting points for custom AI tasks.

CNN Optimization Techniques

CNN optimization involves techniques to improve the performance, efficiency, and generalization of Convolutional Neural Networks during training and inference.

- **Goals:**

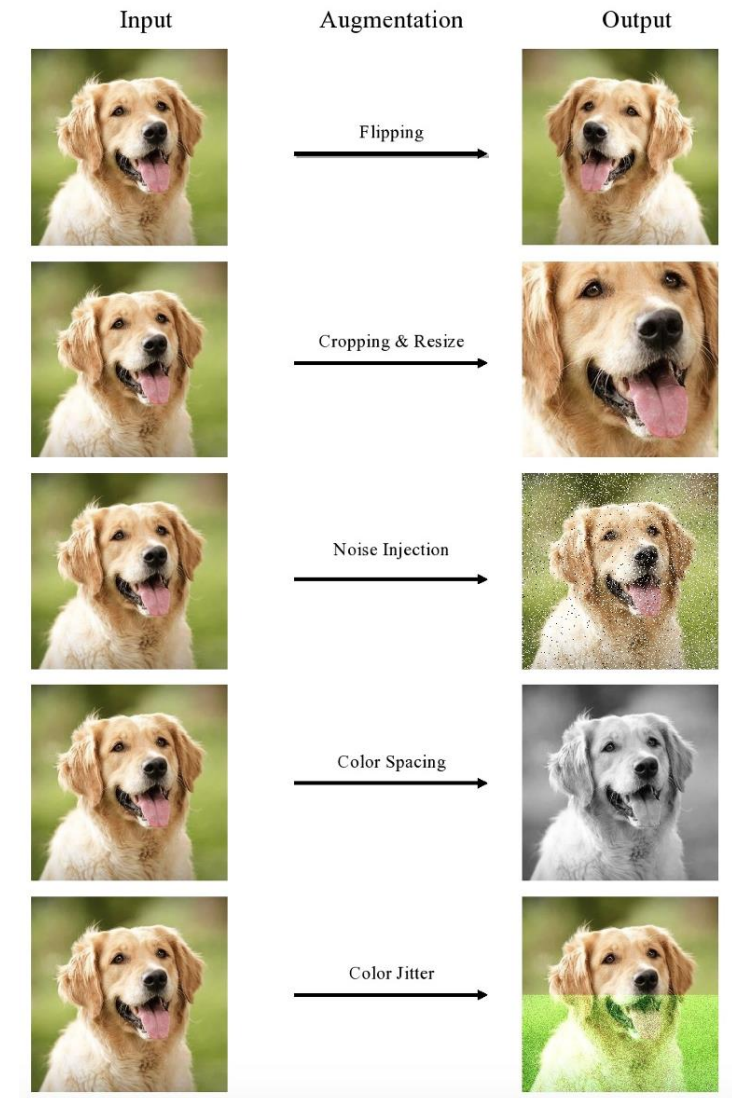
- a) Reduce overfitting.
- b) Improve convergence speed.
- c) Optimize computational resources.

- **Common Strategies:**

- a) Data Augmentation
- b) Regularization (L1, L2, Elastic Net)
- c) Dropout & Early Stopping
- d) Transfer Learning

Data Augmentation

- Data augmentation is a strategy used to artificially increase the size and diversity of a training dataset by applying transformations to the existing data.
- **Purpose:** Improve model generalization. Prevent overfitting. Compensate for limited training data.
- *Why?* Increases effective training set size without extra data collection.
- **Common Methods:**
 - Color jittering, cropping, flipping, rotations, scaling.
 - PCA-based color augmentation (as in AlexNet) (Krizhevsky et al., *Commun. ACM*, 2017).
- Transfer learning approach using well-known CNN models (GoogleNet, AlexNet, VGG16, VGG19, DenseNet, etc.) along with data augmentation techniques can be used to accelerate the training and testing process while yielding good results and performance.
- He et al. implemented data augmentation along with regularization techniques such as dropouts and weight decay (CVPR, 2016).



Teerath et al. *IEEE Access* (2024)

Data Augmentation

- **Geometric Transformations:**

Flipping: Horizontal and vertical flips.

Rotation: Rotates images by a specified angle.

Scaling: Resizes images while preserving aspect ratio.

Cropping: Extracts subregions from the image.

- **Color Transformations:**

Brightness Adjustment: Alters image brightness.

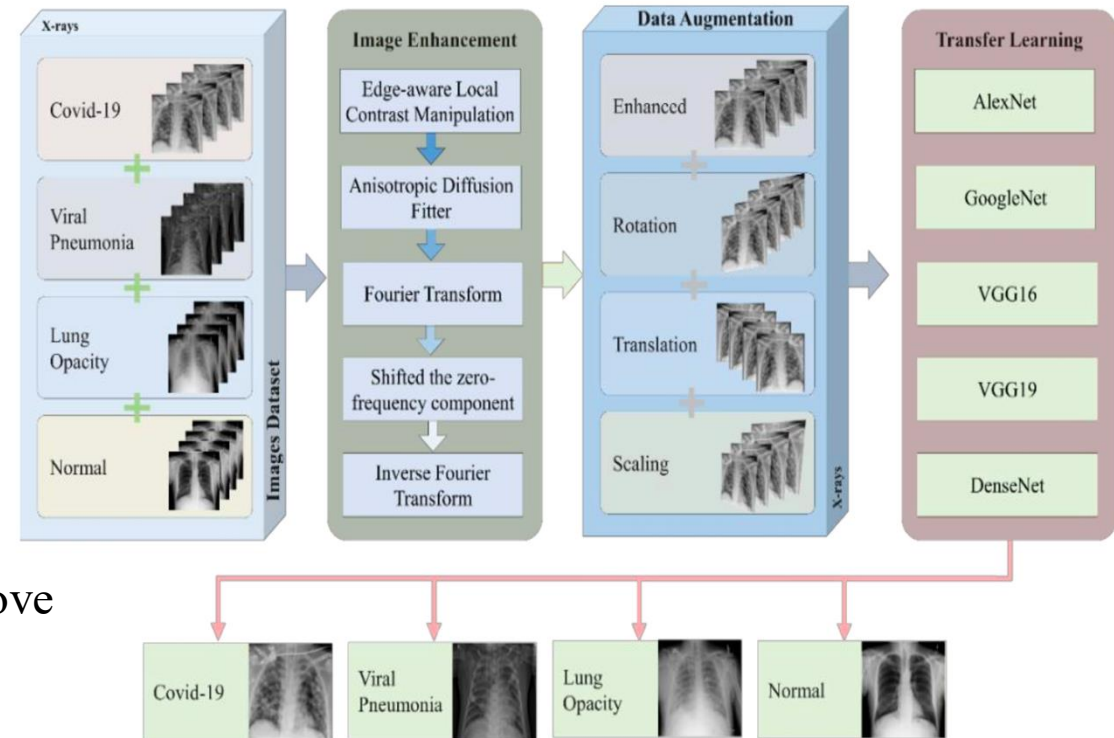
Contrast Adjustment: Modifies contrast levels.

Saturation Adjustment: Changes color saturation.

Hue Adjustment: Shifts color hues.

- **Noise Injection:** Adds random noise to images to improve robustness.

- **Affine Transformations:** Applies scaling, shearing, or translation to the images.

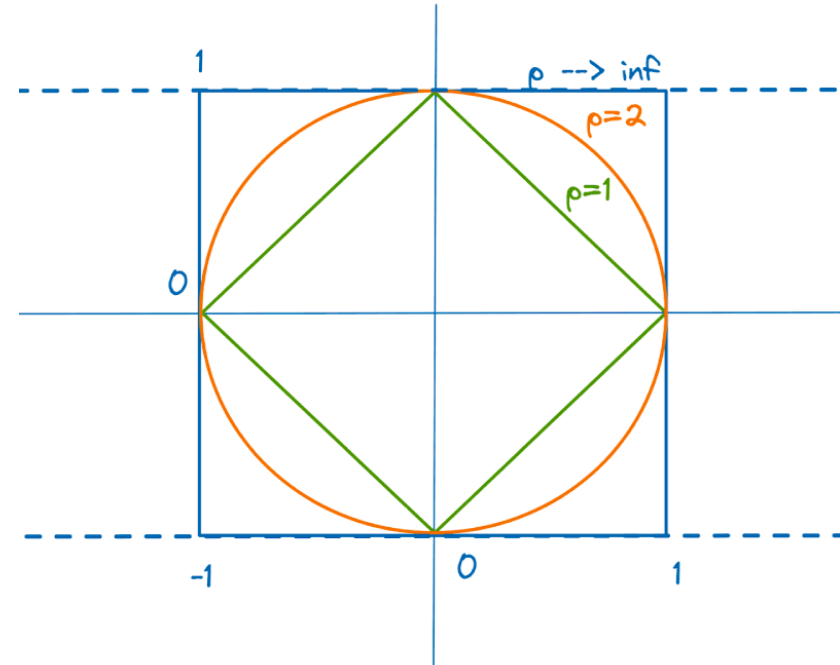


Example of using preprocessing techniques along with the well-known CNN models for COVID-19 and Lungs Pneumonia detection using transfer learning.

Latif et al. *AIMS Mathematics* (2024)

Regularization methods

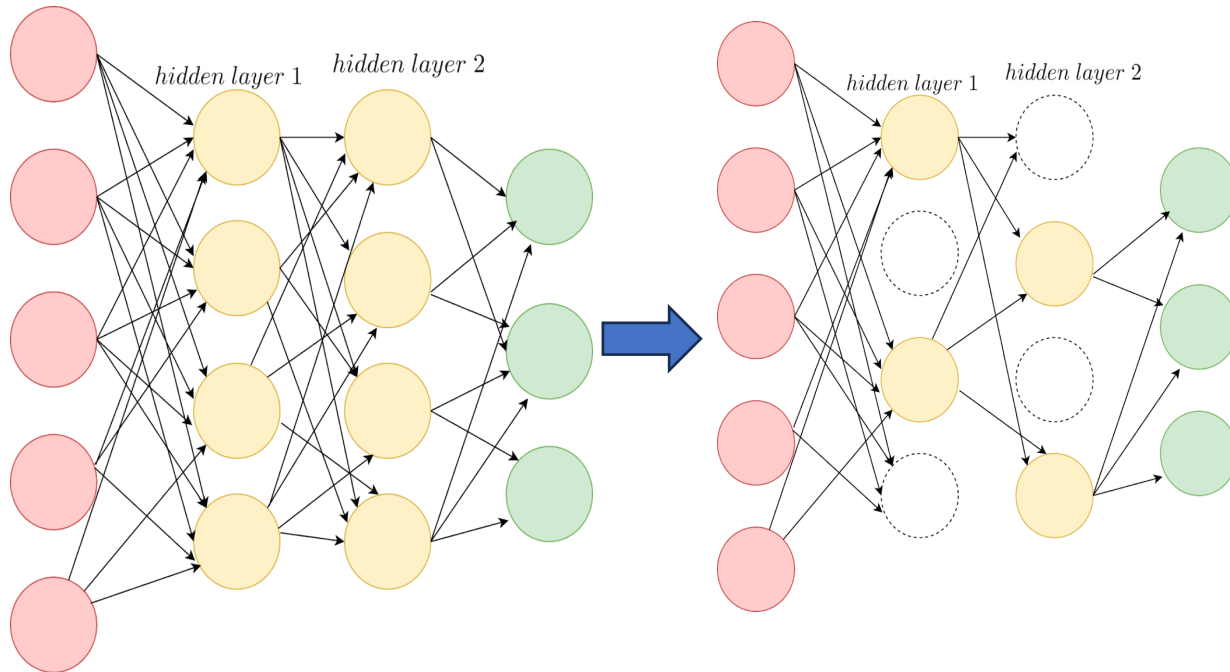
- **Definition:** Regularization refers to techniques that improve a model's generalization by reducing overfitting to the training data.
- **Why Regularization?** Deep learning models are prone to overfitting due to high capacity and complex structures. Regularization helps balance the trade-off between model complexity and performance.
- **L2 Regularization (Weight Decay)**
 - Penalizes the square of weights $\rightarrow$ discourages large weight values, helps smooth solutions.
- **L1 Regularization (Lasso)**
 - Penalizes the absolute value of weights $\rightarrow$ encourages sparsity (some weights become zero).
- **Elastic Net**
 - Combines L1 and L2 $\rightarrow$ can both shrink weights and promote sparsity.



Dropout & Early Stopping

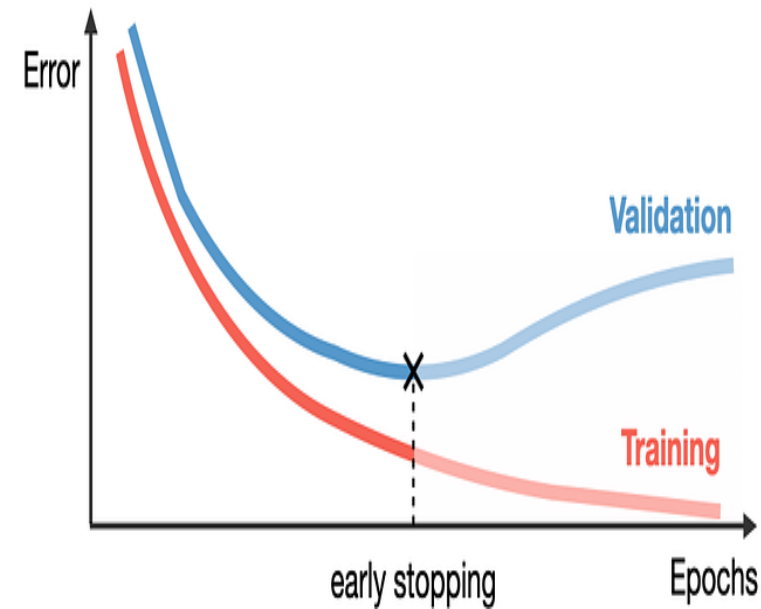
- **Dropout**

- Randomly “drops” neurons during training.
- Reduces co-adaptations among neurons → mitigates overfitting.



- **Early Stopping**

Monitors validation performance and halts training before overfitting sets in. Balances bias/variance by stopping at the optimal point.

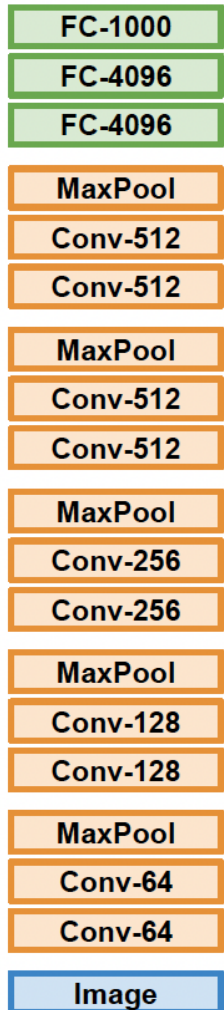


<https://www.pinecone.io/learn/regularization-in-neural-networks/>

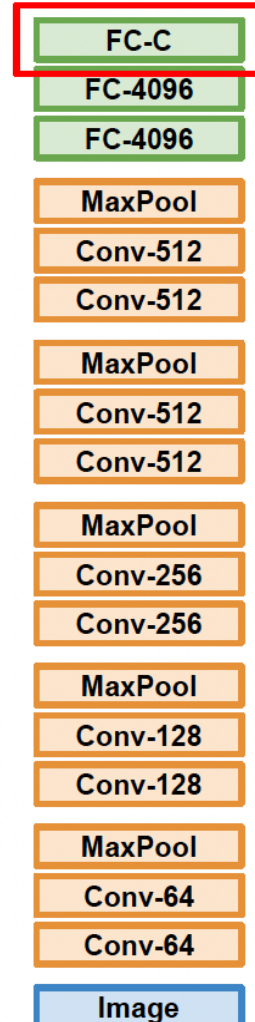
<https://www.comet.com/site/blog/4-techniques-to-tackle-overfitting-in-deep-neural-networks/>

Transfer Learning

Train on ImageNet



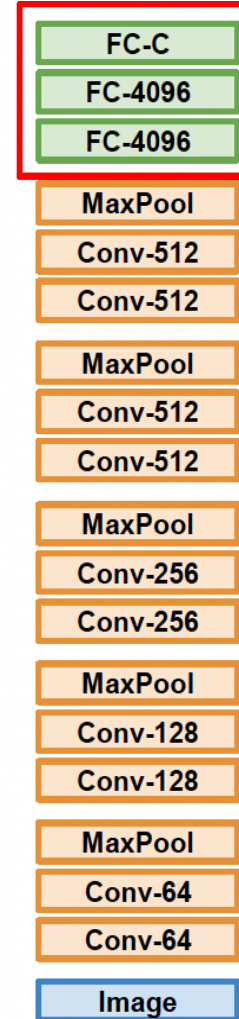
Small Datasets



Reinitialize
this and train

Freeze these

Bigger Datasets



Train these

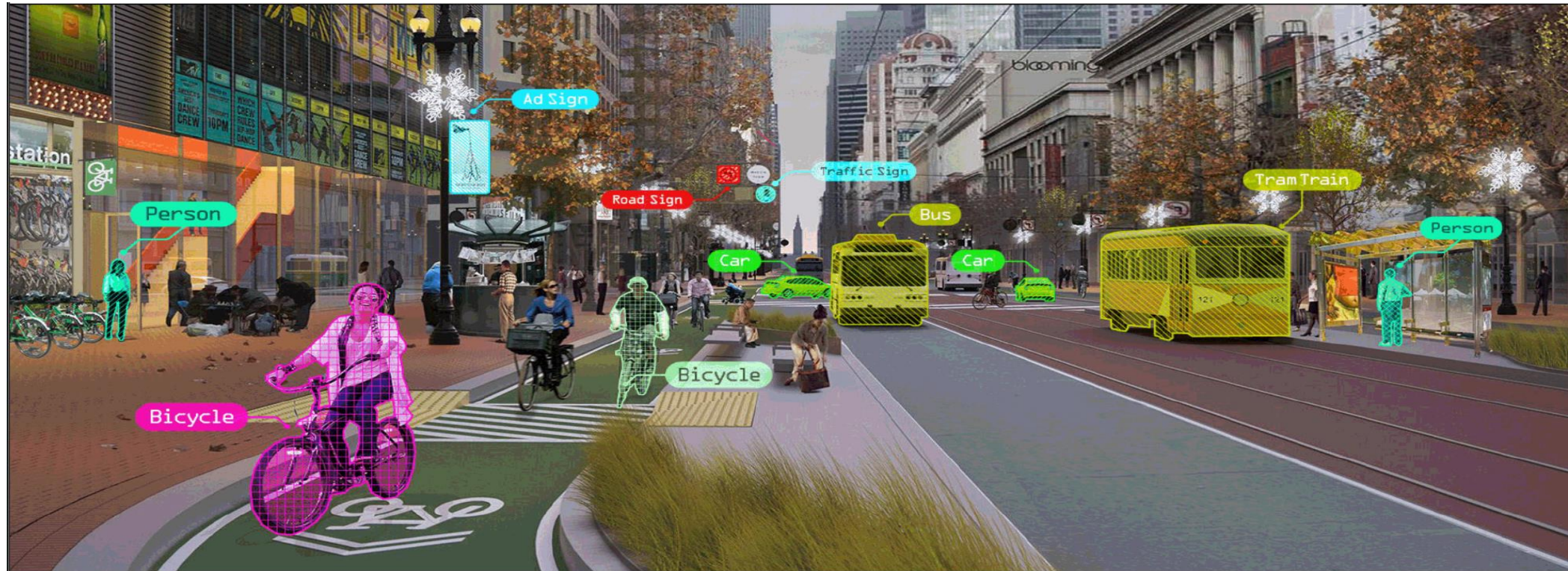
With bigger
dataset, train
more layers

Freeze these

Lower learning rate
when finetuning;
1/10 of original LR
is good starting
point

*Donahue et al, ICML
2014*
*Razavian et al, CVPR
Workshops 2014*

Object Detection



Challenge:

- Objects can be anywhere in the scene, in any orientation, rotation, color hue, etc.
- How can we overcome this challenge?

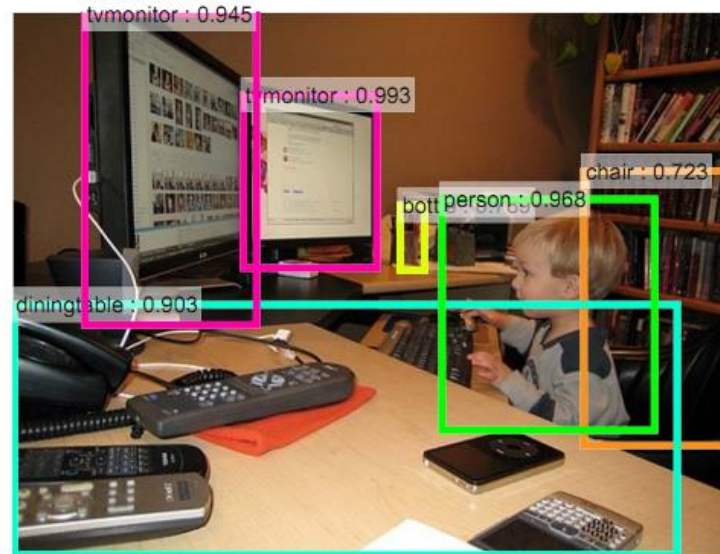
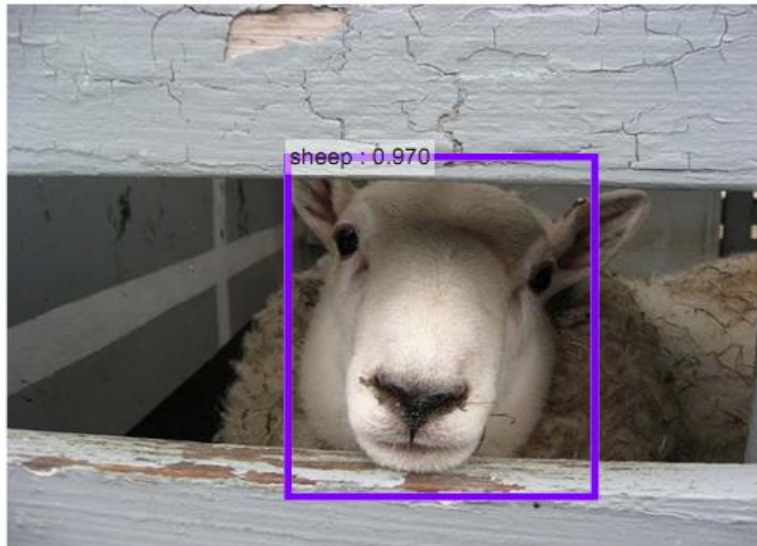
Answer:

- Learn a ton of features (millions) from the bottom up
- Learn the convolutional filters, rather than pre-computing them

What is Object Detection?

To determine: *What objects are where?*

- Object bounding box: location and size
- Object category.



(a)

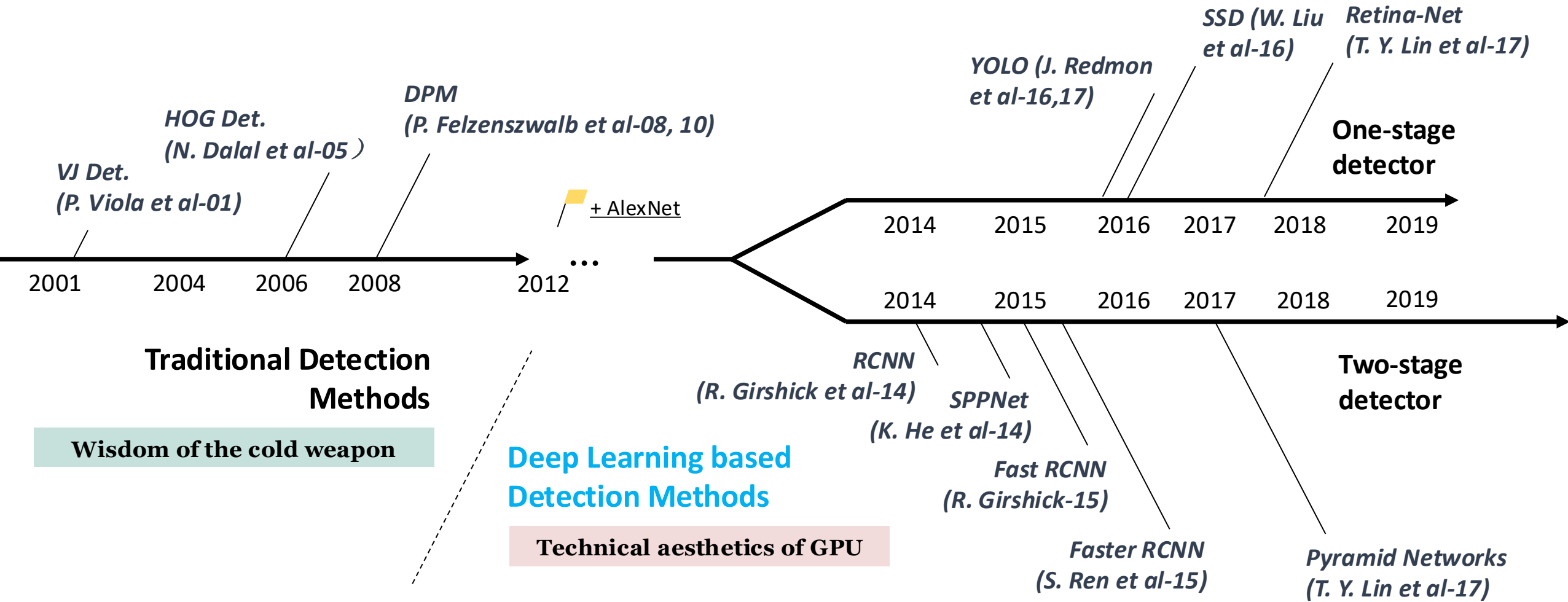
(b)



(c)

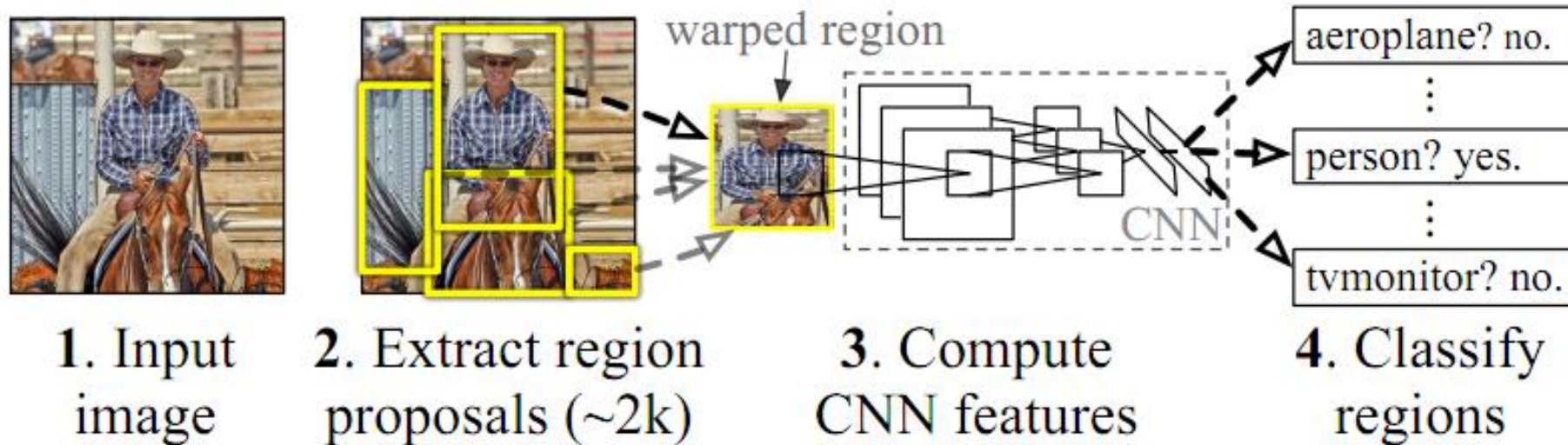
By NIPS15-Faster RCNN

Object Detection Milestones



R-CNN: Regions with CNN features

Ross B. Girshick et al., (CVPR2014)



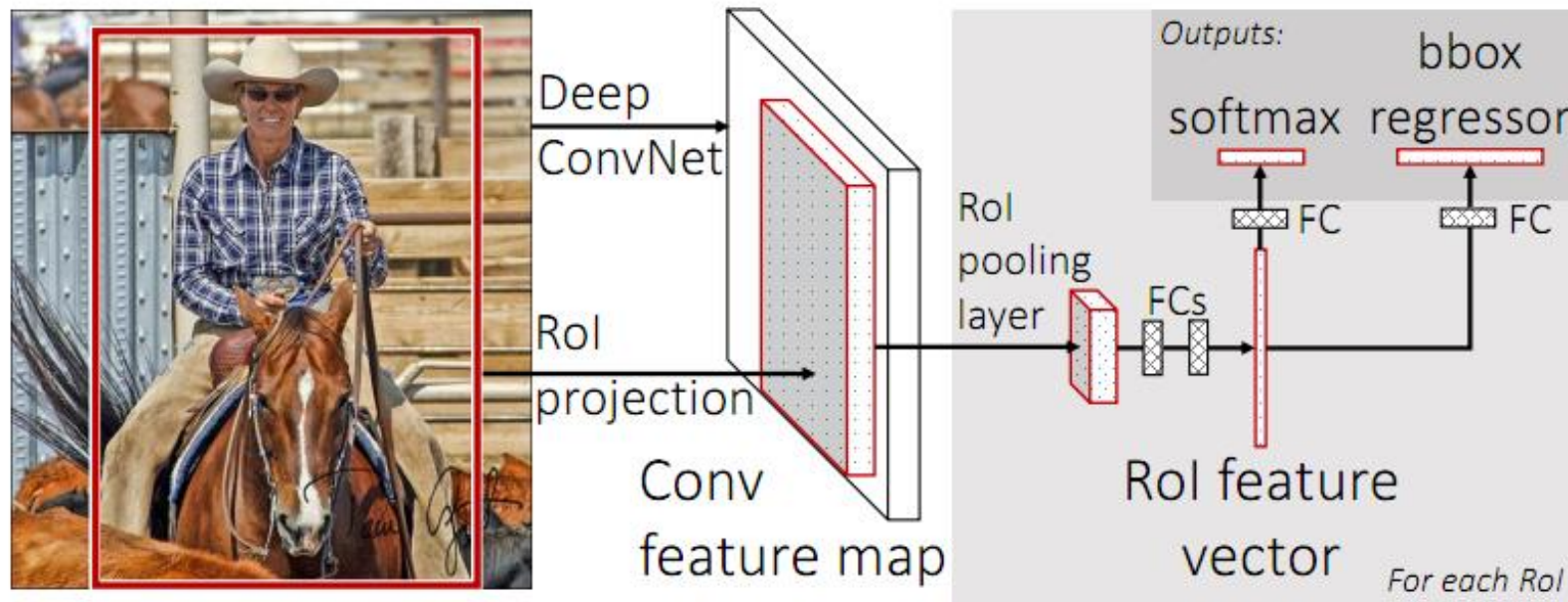
- Object Proposal+CNN features
- Bounding Box Regression
- Fine tuning
- VOC07 mAP: 33.7→58.5

Time: 14s/image on a GPU

Drawbacks

- The redundant feature computations on a large number of overlapped proposals (>2000 boxes/img) leads to an extremely slow detection speed (14s per image with GPU).

Fast RCNN



Fast R-CNN is an object detection framework introduced by Ross Girshick in 2015. It improves upon the inefficiencies of R-CNN by introducing Region of Interest (ROI) Pooling and enabling shared computation, leading to faster and more accurate object detection.

- ROI Pooling
- Multi-task loss (Cls. + BB Reg.)
- BP through ROI pooling layers
- VOC07 mAP: 58.5→70.0

Time:
0.32s/image on a GPU

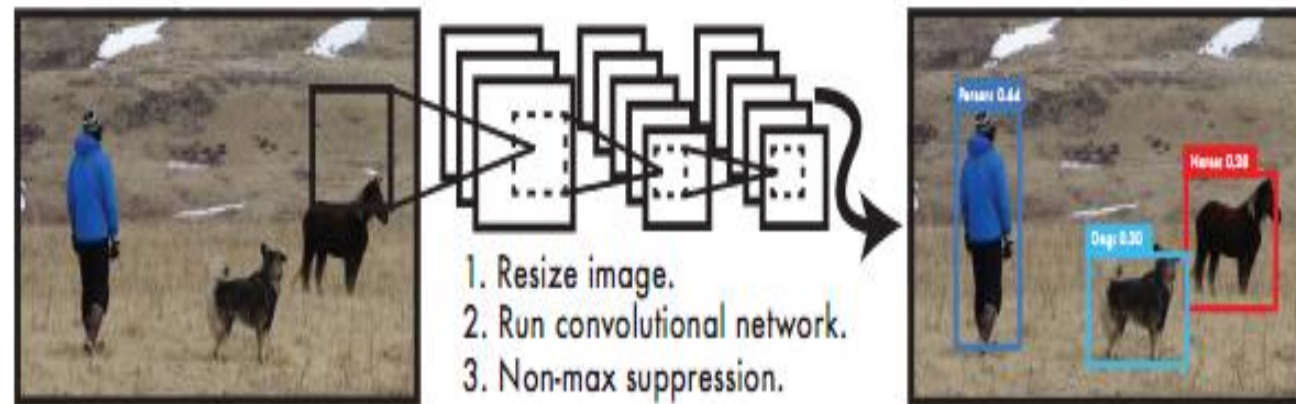
Ross B. Girshick (ICCV15)

You Only Look Once (YOLO)

YOLO treats object detection as a single regression problem, predicting both class probabilities and bounding box coordinates in one forward pass.
J. Redmon et al., (CVPR2016)

Key Contributions:

- ❖ Introduces a unified framework for object detection, enabling real-time performance.
- ❖ Processes the entire image in a single forward pass, improving efficiency.
- ❖ Balances speed and accuracy, making it suitable for real-world applications.

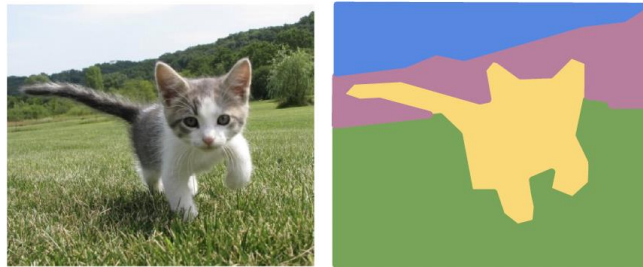


Workflow of YOLO

- **Input Image:** The input image is divided into an SXS grid (e.g., 7X7).
- **Feature Extraction:** A CNN processes the image to extract features.
- **Bounding Box Prediction:** Each grid cell predicts:
 - Bounding boxes (coordinates and dimensions).
 - Confidence scores for each bounding box.
- **Classification:** Each grid cell predicts class probabilities for the objects it contains.
- **Post-Processing:** Non-Maximum Suppression (NMS) removes duplicate detections and retains the most confident predictions.

- Runs at 45fps with VOC07 mAP=63.4% and VOC12 mAP=57.9%.
- A fast version runs at 155fps with VOC07 mAP=52.7%.

Semantic Segmentation: The Problem



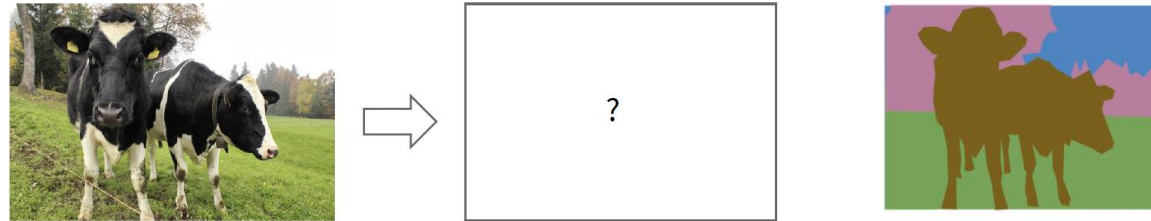
GRASS, CAT, TREE,
SKY, ...

Paired training data: for each training image, each pixel is labeled with a semantic category.

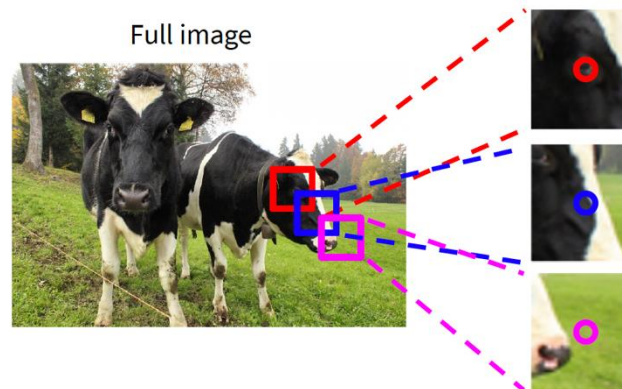


Impossible to classify without context

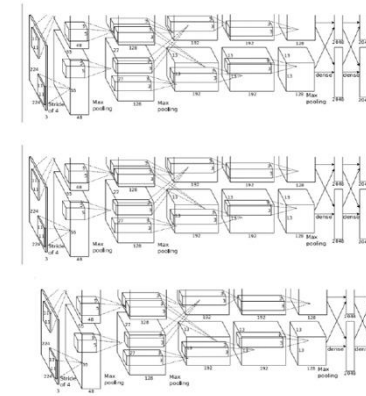
Q: how do we include context?



At test time, classify each pixel of a new image.



Classify center pixel with CNN



Cow

Cow

Grass

Q: how do we model this?

Farabet et al, "Learning Hierarchical Features for Scene Labeling," TPAMI 2013

Pinheiro and Collobert, "Recurrent Convolutional Neural Networks for Scene Labeling", ICML 2014

U-Net: Motivation

In CNNs, different layers learn different feature levels:

- **Lower layers:** Learn low-level, fine-grained details (e.g., edges, textures)
- **Higher layers:** Capture high-level, coarse-grained semantic features (e.g., shape, structure)
- This hierarchy is ideal for classification tasks but introduces limitations for pixel-level tasks like segmentation

Challenges in Medical Image Segmentation

- Medical images often suffer from:
 - Noise
 - Low contrast
 - Blurred or unclear boundaries
- Relying only on low-level features results in poor object recognition
- Relying only on high-level semantic features leads to inaccurate boundary detection

Need for Multi-Level Feature Integration

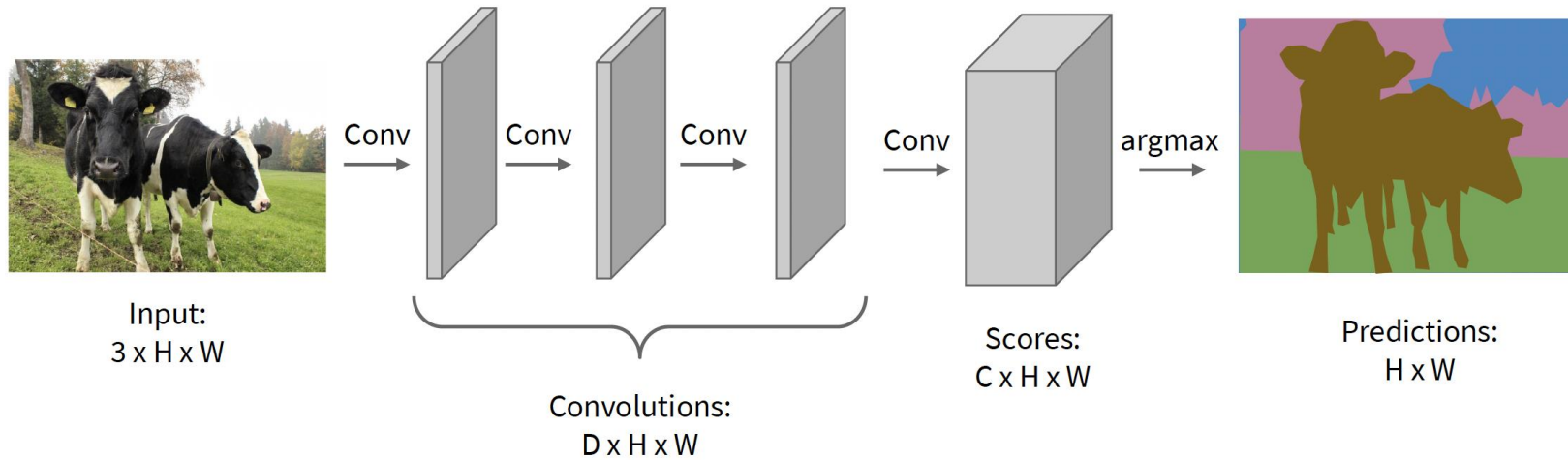
- Effective segmentation requires a combination of:
 - High-level semantic understanding (context)
 - Low-level spatial precision (details)
- General CNNs lack explicit mechanisms to combine both effectively

Encoder-Decoder Architectures

- Designed to combine high-level and low-level features
- Consist of:
 - **Encoder:** Downsamples and extracts abstract features
 - **Decoder:** Upsamples to recover spatial resolution and integrates detail
- Enables pixel-level prediction with semantic awareness

Semantic Segmentation Idea

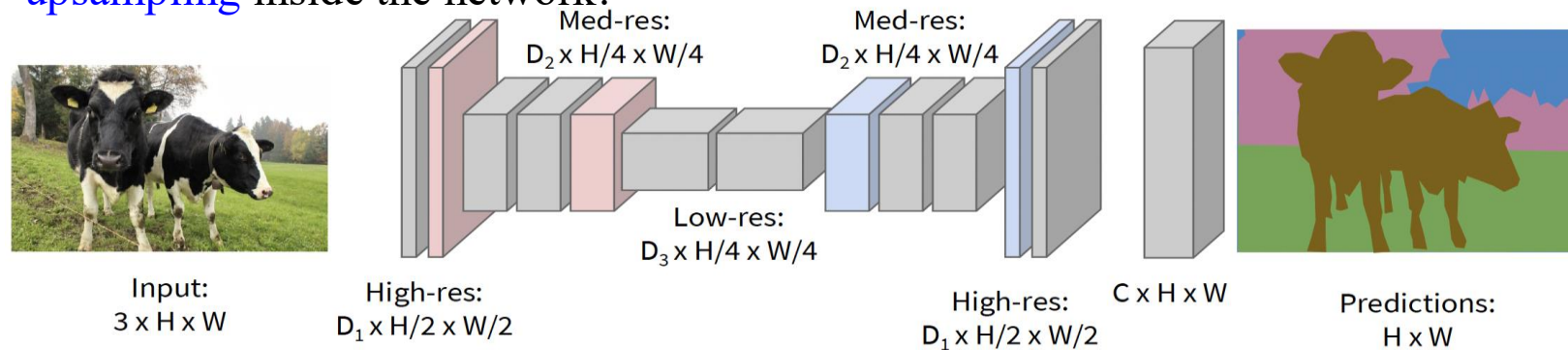
Design a network with only convolutional layers without downsampling operators to make predictions for pixels all at once!



Problem: convolutions at original image resolution will be very expensive ...

Design network as a bunch of convolutional layers, with **downsampling** and **upsampling** inside the network!

Downsampling:
Pooling, strided
convolution

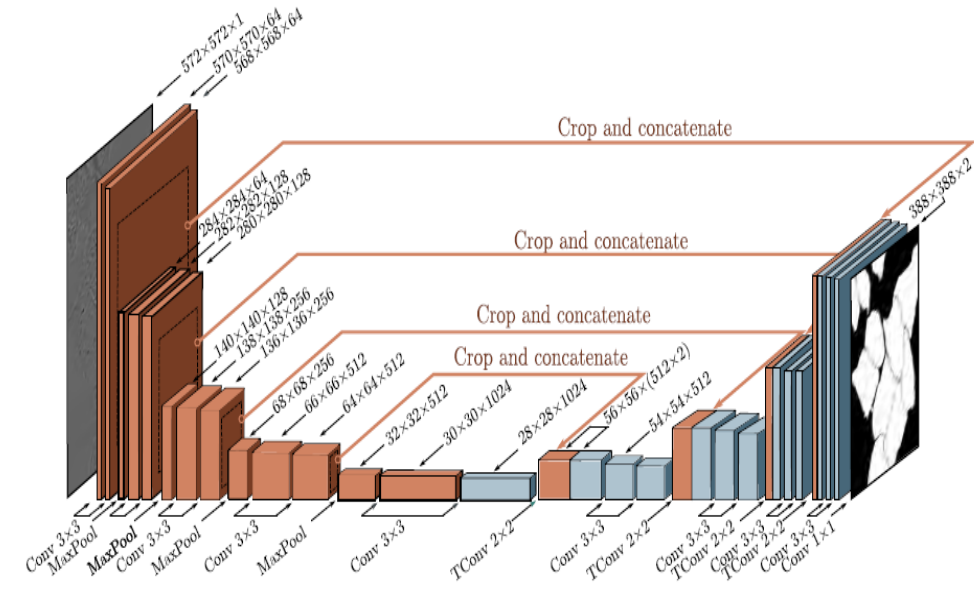


Upsampling:
Unpooling or strided
transposed convolution

Modern DL Model Architectures

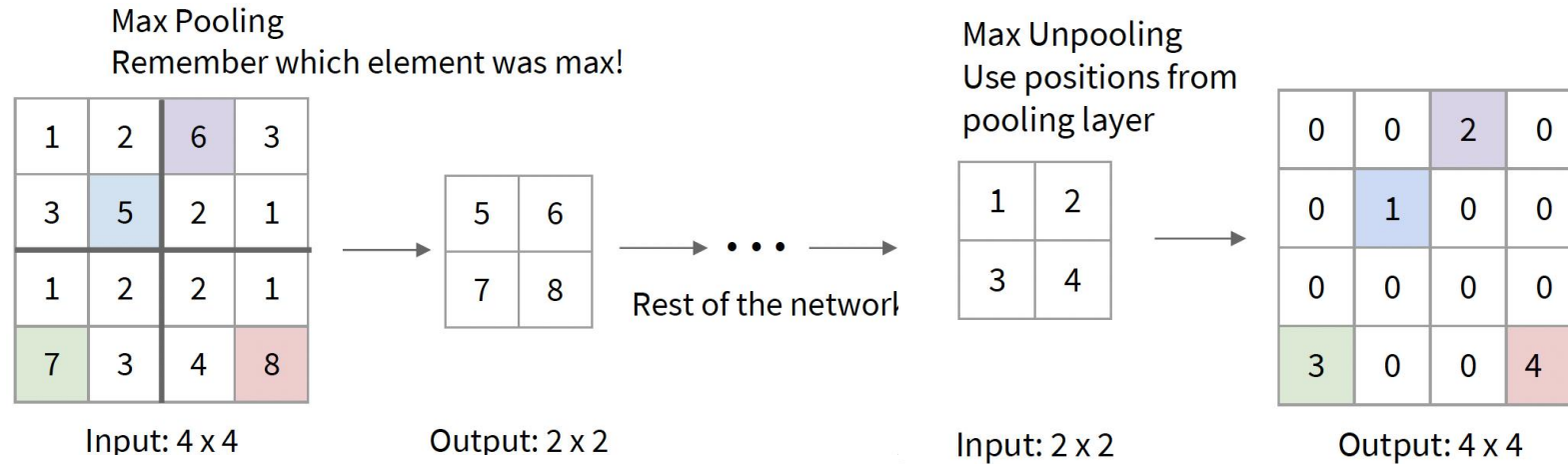
U-Net

- **Key Features:** U-shaped architecture with symmetric encoder and decoder paths. Skip connections that concatenate feature maps from encoder to decoder
- **Structure:** Encoder: Series of convolutional and max-pooling layers that capture context. Bottleneck: Intermediate layer connecting encoder and decoder. Decoder: Series of up-convolution and concatenation layers that restore resolution. Final Layer: Convolutional layer that maps features to the desired output.
- **Types:** 2D/3D U-Net, Attention U-Net.
- **Applications in Biomedicine:** Medical image segmentation. Satellite image segmentation. Biomedical image analysis. Autonomous driving. General image segmentation tasks.

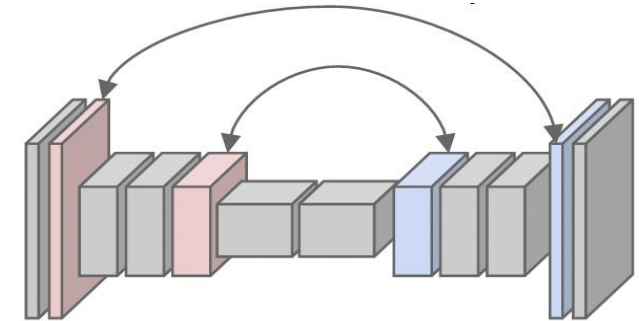


U-Net for segmenting HeLa cells. The U-Net has an encoder-decoder structure, in which the representation is downsampled (orange blocks) and then re-upsampled (blue blocks). The encoder uses regular convolutions, and the decoder uses transposed convolutions. Residual connections append the last representation at each scale in the encoder to the first representation at the same scale in the decoder (orange arrows).

Downsampling and Upsampling



Corresponding pairs of downsampling and upsampling layers



Common Downsampling types:

- **Max pooling:** Takes the maximum value in each window
- **Average pooling:** Computes the average value
- **Stochastic pooling:** Randomly selects an activation based on a probability distribution
- **LP-pooling:** Generalized pooling that uses the p-norm over each region
- **Global pooling:** Applies pooling over the entire feature map to reduce to a single value per channel

• **Purpose:** (i) Reduce computation; (ii) Increase receptive field; (iii) Achieve spatial invariance; (iv) Introduce regularization

Common unpooling strategies:

- **Max-unpooling with indices:**
- **Fixed-position unpooling:** inserts values at top-left corner of window
- **Interpolation-based unpooling:** uses nearest-neighbor or bilinear interpolation to expand feature maps
- **Learnable unpooling:** introduces parameters to learn where and how to upsample

Often followed by convolutional layers to refine outputs

U-Net: Vanilla Version

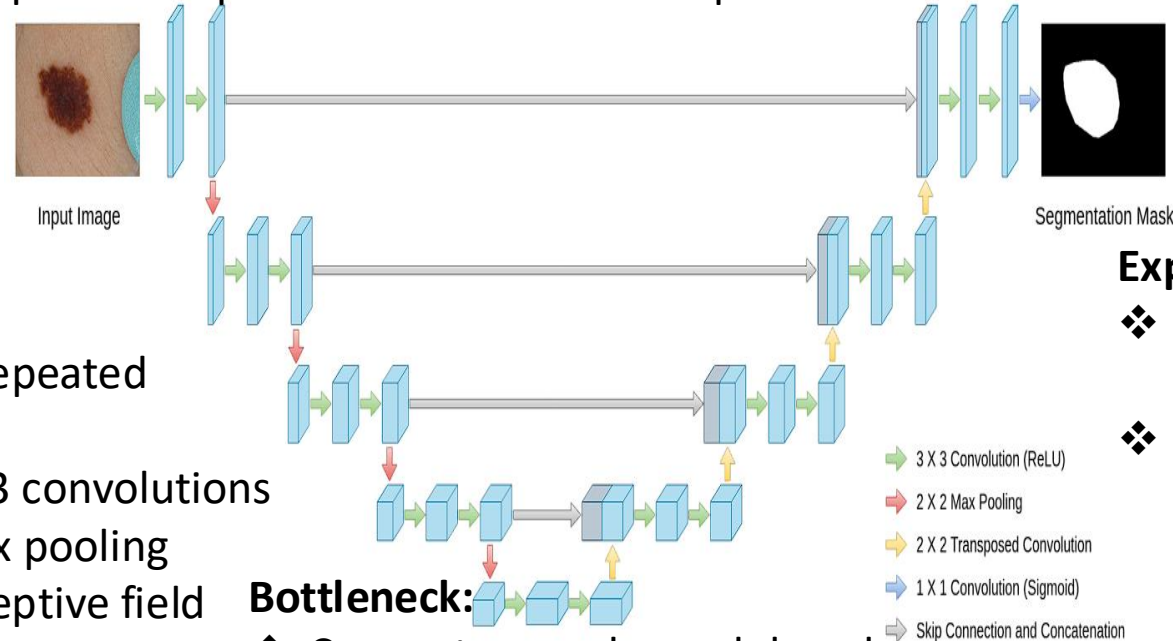
❖ U-Net is a neat end-to-end neural network with a characteristic "U" shape

Contracting Path (Encoder):

- ❖ Captures context through repeated downsampling blocks
- ❖ Each block includes two 3×3 convolutions + ReLU, followed by 2×2 max pooling
- ❖ Gradually increases the receptive field without heavy computation

Skip Connections:

- Link encoder and decoder layers at the same depth level
- Concatenate encoder feature maps with decoder inputs to combine detailed and contextual information
- Help restore spatial resolution and sharpen boundaries



Bottleneck:

- ❖ Connects encoder and decoder
- ❖ Two 3×3 convolutions + ReLU
- ❖ Reduces spatial resolution and increases depth for high-level abstraction

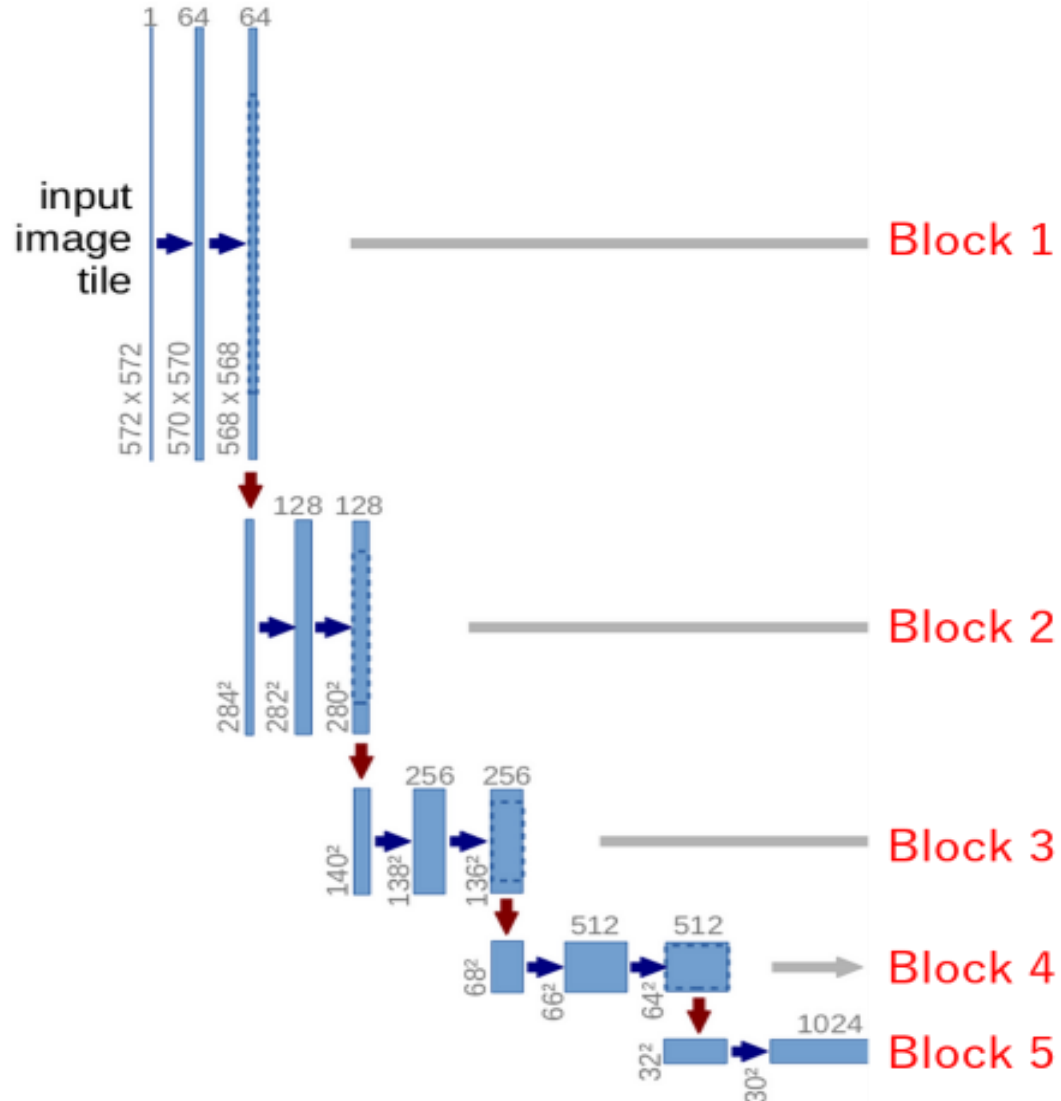
Final Output:

- ❖ A 1×1 convolution maps the final feature map to the number of target classes
- ❖ Produces a pixel-level classification map (e.g., segmentation mask)

Expanding Path (Decoder):

- ❖ Upsamples feature maps to match input resolution
- ❖ Each block includes one 2×2 transposed convolution (up-conv), two 3×3 convolutions + ReLU

Contracting Path (Encoder)



❖ Block 1:

- ❖ Input: 572x572x1 (grayscale image)
- ❖ Two 3x3 unpadded convolutions + ReLU → 64 channels
- ❖ 2x2 max pooling (stride 2) → downsampled to 284x284

❖ Block 2:

- ❖ Two 3x3 convolutions + ReLU → 128 channels
- ❖ 2x2 max pooling → 140x140

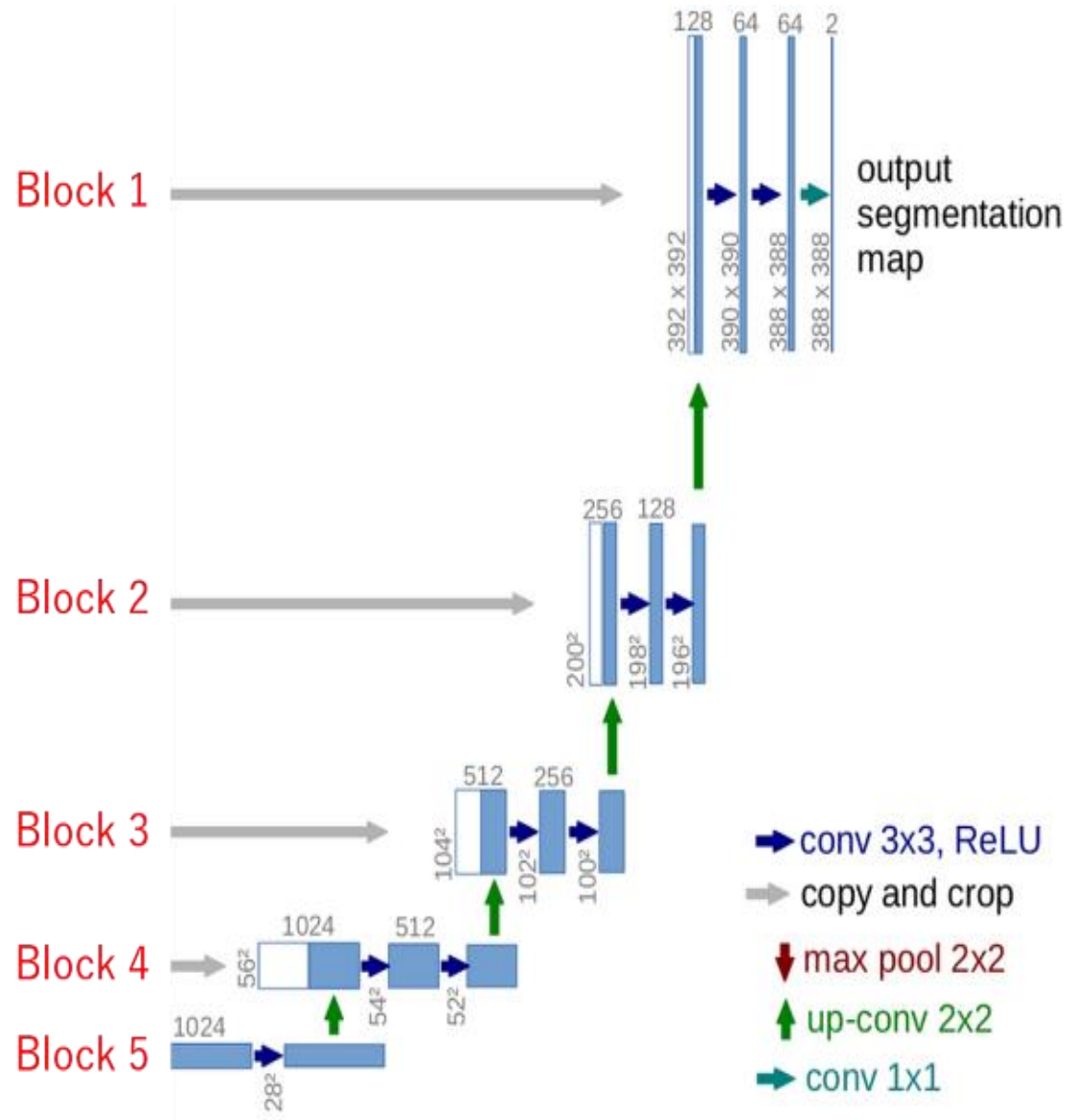
❖ Block 3 & Block 4:

- ❖ Same as previous blocks with doubled channels (256, 512)
- ❖ Max pooling after each block halves spatial dimensions

❖ Block 5 (Bottom):

- ❖ Two 3x3 convolutions + ReLU → 1024 channels
- ❖ First conv in this block included here, second used in expanding path for symmetry

Expanding Path (Decoder)



•Block 5:

- Continues from the bottom block with a second 3×3 convolution + ReLU
- Followed by a 2×2 up-convolution $\rightarrow$ doubles spatial resolution, reduces channels to 512

•Block 4:

- Skip connection: concatenate encoder feature map (cropped to match size) $\rightarrow$ 1024 channels
- Two 3×3 convolutions + ReLU $\rightarrow$ reduce to 512 channels
- 2×2 up-convolution $\rightarrow$ upsample and reduce channels to 256

•Block 3 & Block 2:

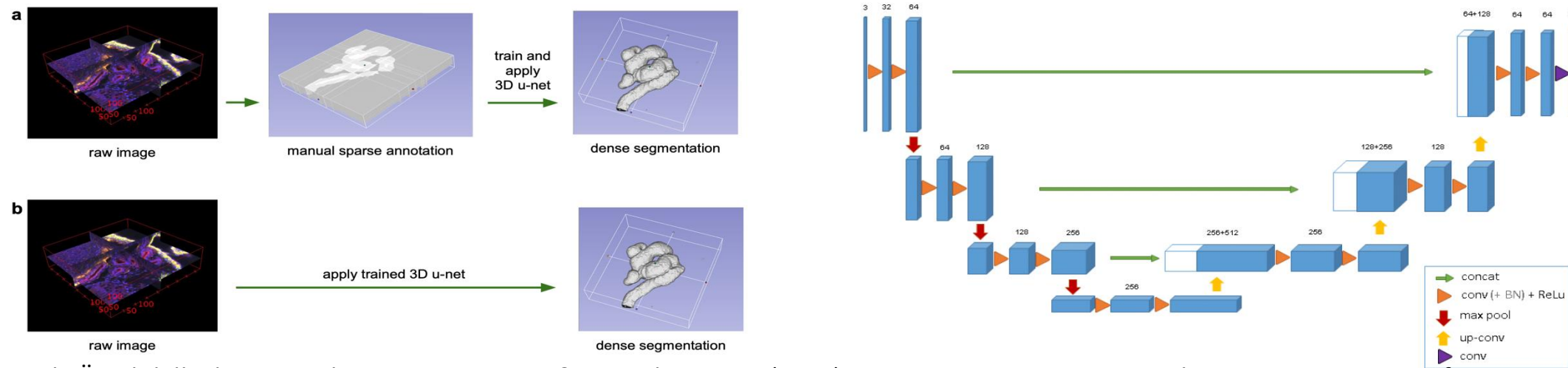
- Same as Block 4, with halved channels: $256 \rightarrow 128 \rightarrow 64$

•Block 1 (Final Block):

- After skip connection: 128 channels
- Two 3×3 convolutions + ReLU $\rightarrow$ reduce to 64 channels
- Final 1×1 convolution $\rightarrow$ maps to number of classes (e.g., 2 for binary)
- Followed by activation function (e.g., sigmoid for binary classification)

3D U-Net

- Due to the abundance and representation power of volumetric data, most medical image modalities are three-dimensional. 3D U-Net was commonly used in Brain tumor segmentation (e.g., BraTS dataset), Lung nodule detection, and liver and pancreas segmentation.
- 3D U-Net is proposed to deal with 3D medical data directly. It replaces all 2D operations with their 3D counterparts. The users can annotate some slices in the volume to be segmented. The model then learns from these sparse annotations and provides a dense 3D segmentation.
- However, due to the limitation of computational resources, it only includes three down-sampling, which cannot effectively extract deep-layer image features, leading to limited segmentation accuracy for medical images.



Çiçek, Ö., Abdulkadir, A., Lienkamp, S. S., Brox, T., & Ronneberger, O. (2016). *3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation* (No. arXiv:1606.06650). arXiv. <https://doi.org/10.48550/arXiv.1606.06650>

U-Mamba

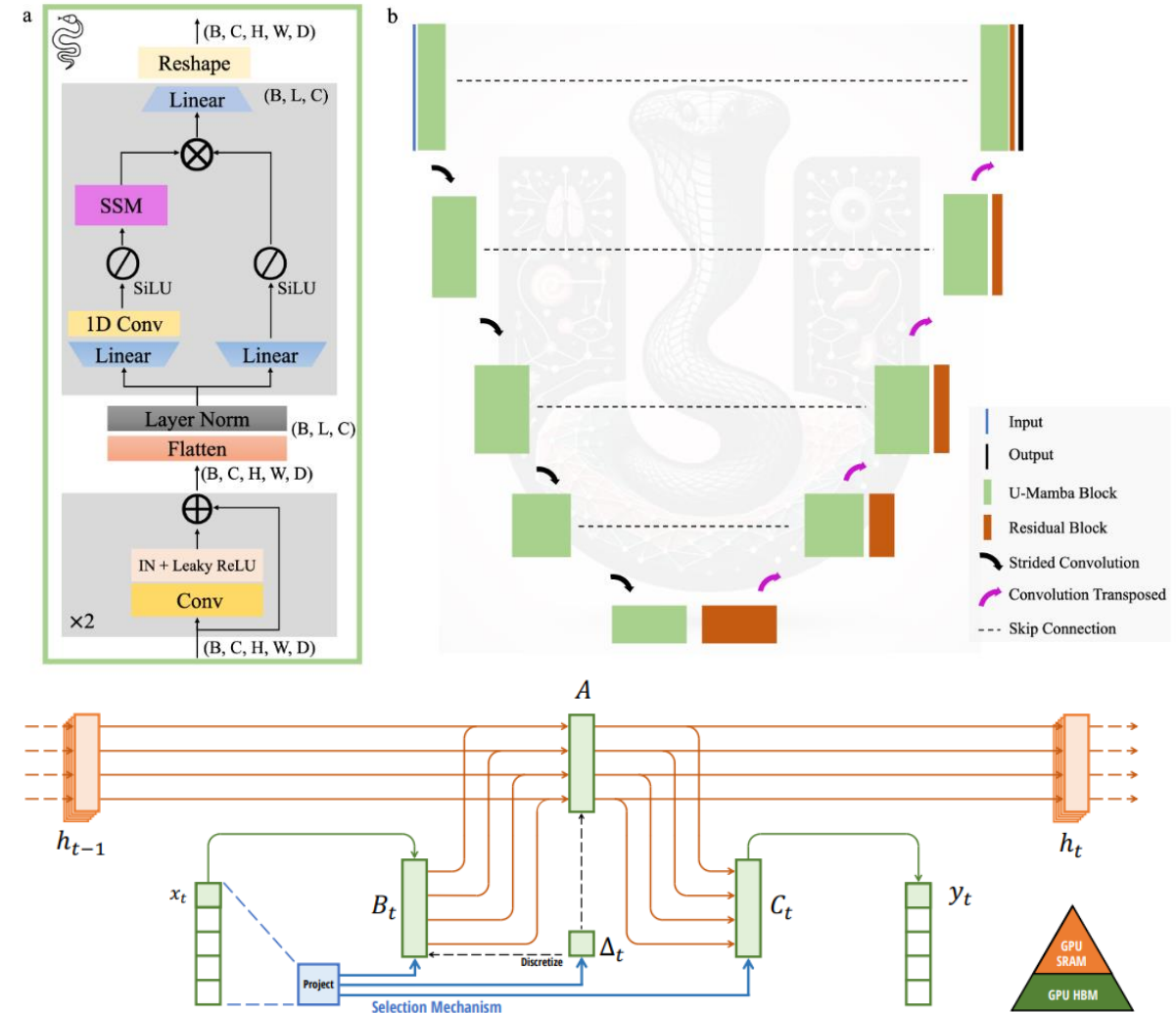
Motivation: CNNs (e.g., U-Net) suffer from **limited receptive fields**. Transformers offer global context but are **computationally expensive**. There is a need for models that **efficiently capture long-range dependencies**.

Key Contributions:

- **U-Mamba** combines:
 - **CNN Residual blocks**: for local feature extraction.
 - **Mamba blocks (SSMs)**: for scalable long-range dependency modeling.
- Employs a **hybrid encoder-decoder** architecture with skip connections.
- **Self-configuring** like nnU-Net: adapts automatically to various datasets.

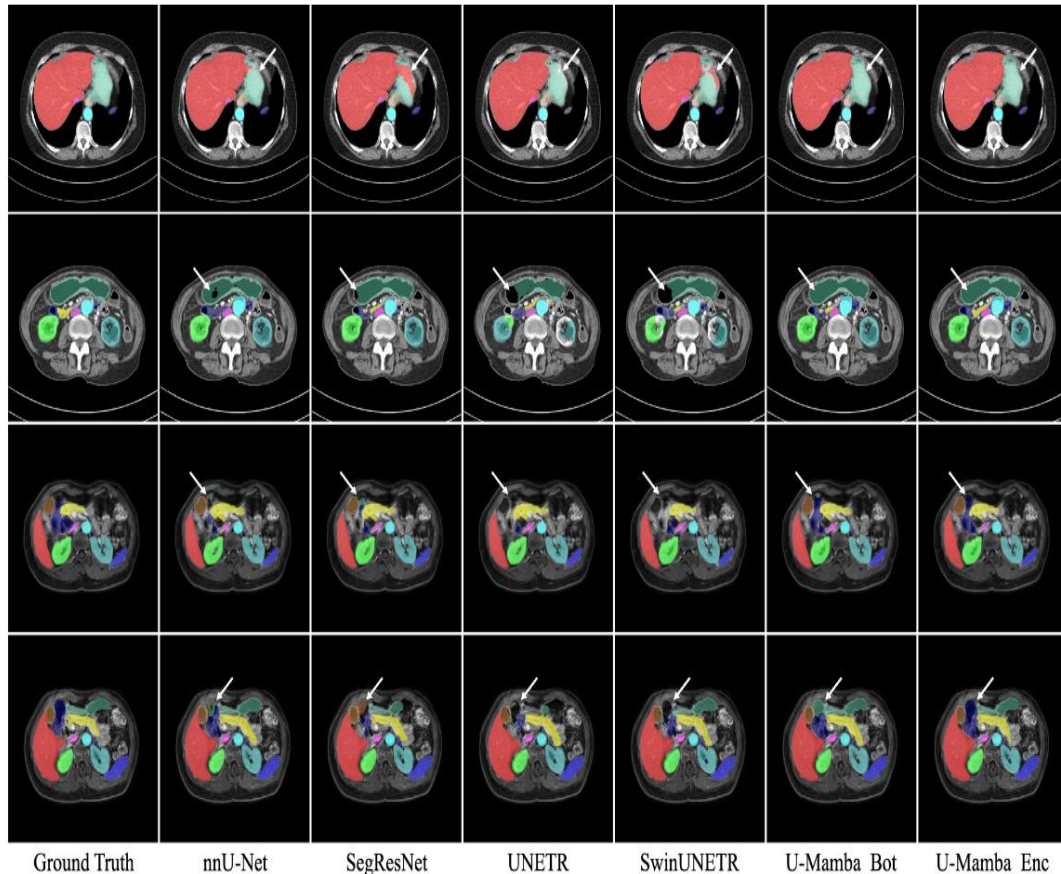
Architecture Overview:

- Encoder: Residual + Mamba blocks.
- Decoder: Residual blocks with transposed convolution.
- Variants: U-Mamba Bot (bottleneck only) and U-Mamba_Enc (full encoder).



U-Mamba

- U-Mamba also enjoys a self-configuration mechanism, as it is implemented within the nnU-Net framework.
- It outperforms STOA CNN-based and Transformer-based segmentation networks across all tasks.

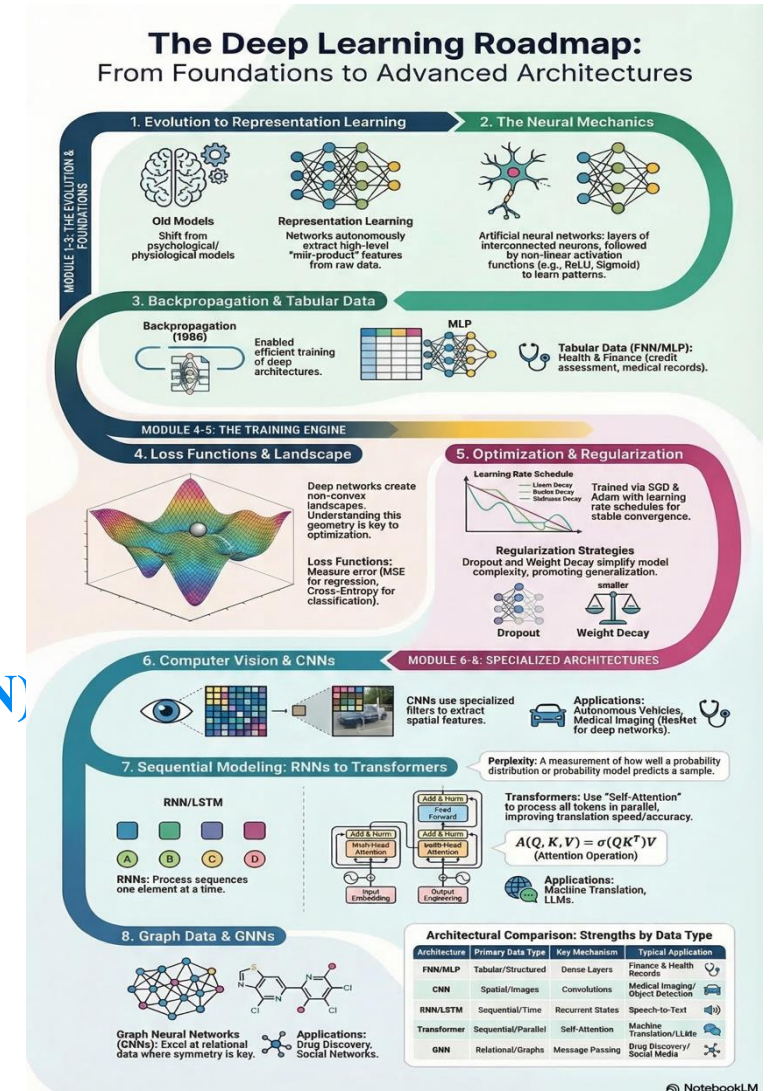


Methods	Organs in Abdomen CT		Organs in Abdomen MRI	
	DSC	NSD	DSC	NSD
nnU-Net	0.8615±0.0790	0.8972±0.0824	0.8309±0.0769	0.8996±0.0729
SegResNet	0.7927±0.1162	0.8257±0.1194	0.8146±0.0959	0.8841±0.0917
UNETR	0.6824±0.1506	0.7004±0.1577	0.6867±0.1488	0.7440±0.1627
SwinUNETR	0.7594±0.1095	0.7663±0.1190	0.7565±0.1394	0.8218±0.1409
U-Mamba_Bot	0.8683±0.0808	0.9049±0.0821	0.8453±0.0673	0.9121±0.0634
U-Mamba_Enc	0.8638±0.0908	0.8980±0.0921	0.8501±0.0732	0.9171±0.0689

Methods	Organs in Abdomen MRI		Instruments in Endoscopy		Cells in Microscopy
	DSC	NSD	DSC	NSD	F1
nnU-Net	0.7450±0.1117	0.8153±0.1145	0.6264±0.3024	0.6412±0.3074	0.5383±0.2657
SegResNet	0.7317±0.1379	0.8034±0.1386	0.5820±0.3268	0.5968±0.3303	0.5411±0.2633
UNETR	0.5747±0.1672	0.6309±0.1858	0.5017±0.3201	0.5168±0.3235	0.4357±0.2572
SwinUNETR	0.7028±0.1348	0.7669±0.1442	0.5528±0.3089	0.5683±0.3123	0.3967±0.2621
U-Mamba_Bot	0.7588±0.1051	0.8285±0.1074	0.6540±0.3008	0.6692±0.3050	0.5389±0.2817
U-Mamba_Enc	0.7625±0.1082	0.8327±0.1087	0.6303±0.3067	0.6451±0.3104	0.5607±0.2784

Content

6 Natural Language Processing, Recurrent Neural Networks (RNN) and Transformer



Motivation

Recurrent Neural Networks (RNNs) are motivated by their ability to address challenges inherent in sequential data.



Weather
forecasting



Stock market
trends



Autocomplete
for texting



Genetic
sequencing

Many real-world datasets are inherently sequential, where **the order of data points is crucial**.

- ❖ **Time series:** Stock prices, weather forecasts, and sensor data require capturing patterns over time.
- ❖ **Text:** The meaning of a sentence depends on word order ("The cat chased the dog" vs. "The dog chased the cat").
- ❖ **Speech:** Phonemes and intonation must be processed sequentially to understand spoken language.
- ❖ **Video:** Frames in a video sequence have temporal relationships that determine the flow of events.



Speech
recognition



Video frame
prediction



Music
composition



...

Sequences

A **sequence** is an **ordered list of elements**, where the order of the elements matters. They are fundamentally different from unordered data because each element is **dependent** or **influenced** by the previous elements.

Examples:

- Letters (words)
- Words (sentences)
- Sentences (documents)
- Frames (video)
- Amino-acids (genetic code)
- fMRI/ECG signals

"Hello, how are you?" (chatbot input).

e.g., A security camera capturing a person walking.

e.g., "ACGTAGCTAGT" represents a biological sequence.

Why Are Sequences Important?

Unlike independent data points, sequences contain **temporal or contextual dependencies**:

- **Future values depend on past values** (e.g., predicting tomorrow's weather).
- **Words in a sentence rely on context** (e.g., in "bank deposit" vs. "river bank").
- **Biological sequences determine genetic functions.**

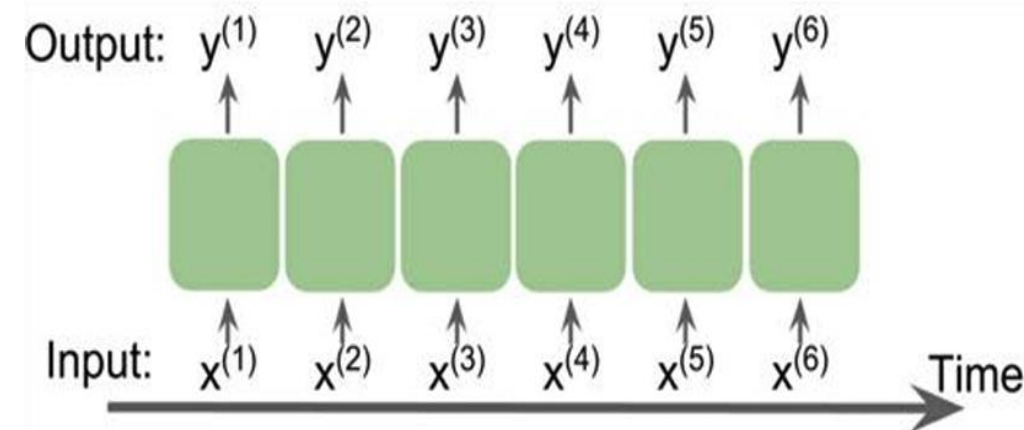
Sequences

Challenges in modeling sequential data

- **Infinite number of possible sequences:**
 - ❖ Sequences can vary in **length** (short vs. long sequences).
 - ❖ Sequences can have **variable patterns** (e.g., DNA sequences, language models).
 - ❖ Order **matters**, meaning different orders of the same elements can have different meanings.
- **Need for Probability Distributions Over Sequences:**
 - Since an infinite number of sequences exist, we **cannot store all possible sequences explicitly**.
 - Instead, we model a **probabilistic function** that assigns a likelihood to each possible sequence.
 - Example: Given a sequence $S=(x_1, x_2, \dots, x_T)$, we want to learn a **probability distribution** $P(S)$.

RNNs are designed for modeling sequences

- Sequences of any length in the input, in the output, or in both
- They can remember past information
- Apply the same weights on each step



Machine Translation

Machine Translation (MT) is the task of translating a sentence x from one language (the **source language**) to a sentence y in another language (the **target language**).

Goal: Produce translations that are both fluent and faithful to the meaning of the source text.

Applications: Global communication, localization cross-lingual information retrieval, etc.

x : *L'homme est né libre, et partout il est dans les fers*

English:

y : Man is born free, but everywhere he is in chains

Chinese:

“人生而自由，但无处不在被枷锁束缚。”

Japanese:

「人間は自由に生まれるが、どこにいても鎖に縛られている。」

NMT: the first big success story

Neural Machine Translation (NMT): Uses deep learning and end-to-end training to model translation and offers improved fluency and the ability to capture complex dependencies.

Neural Machine Translation went from a **fringe research attempt** in **2014** to the **leading standard method** in **2016**

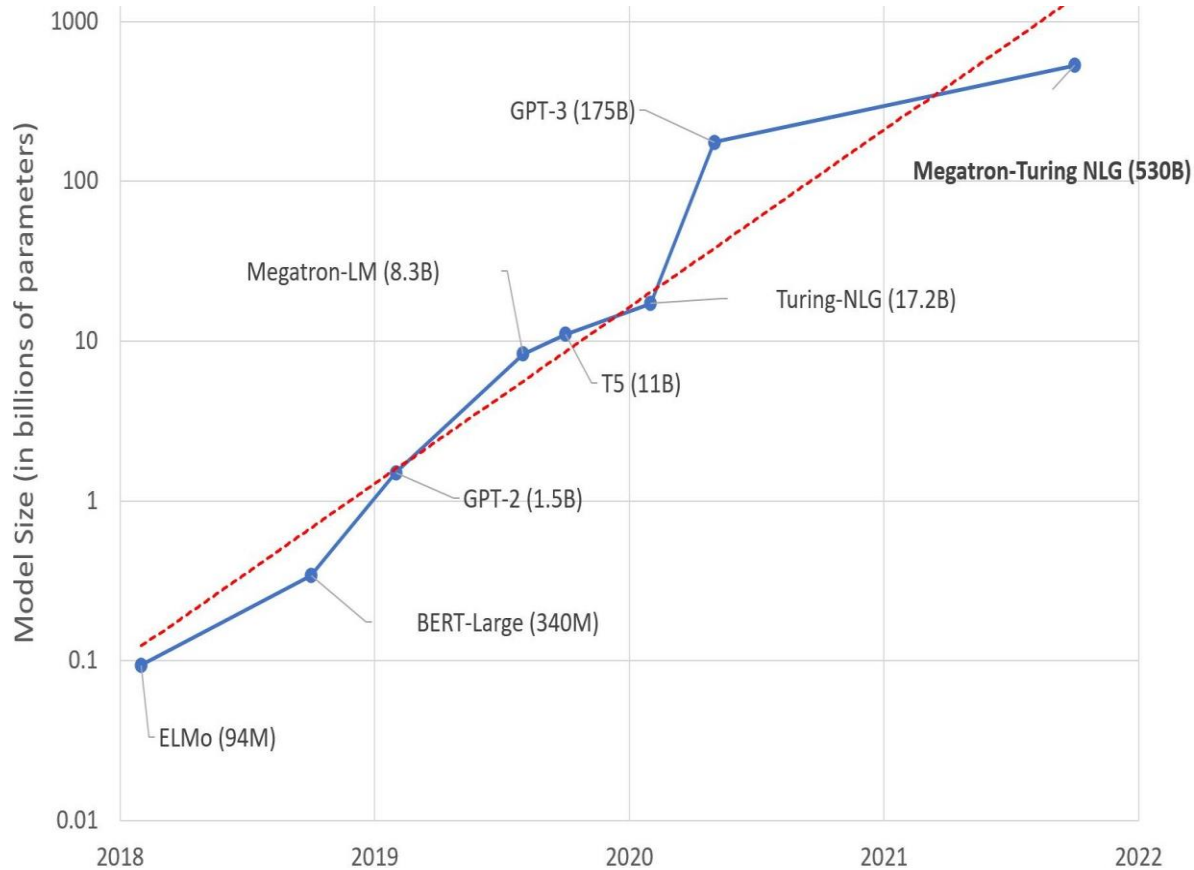
- **2014:** First seq2seq paper published [Sutskever et al. 2014]
- **2016:** Google Translate switches from SMT to NMT – and by 2018 everyone had
 - <https://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html>



- **This was amazing!**
 - **SMT** systems, built by **hundreds** of engineers over many **years**, were outperformed by NMT systems trained by **small groups** of engineers in a few **months**

StanfordCS224n

Deepseek v.s. OpenAI



Modern NLP Systems

<https://ai.plainenglish.io/deepseek-r1-vs-chatgpt-01-my-experience-ddbe09e80aa9>

<https://huggingface.co/blog/large-language-models>

Why RNNs? Challenges of Sequential Data for Standard Networks

1. Fixed Size, No Memory

MLPs/CNNs map a fixed-size input to a fixed-size output, with no recurrence — no memory of past computations, so patterns across time steps cannot be learned.

RNN fix: recurrence: a hidden state carries memory from step to step.

2. Temporal Dependencies

Context over time is essential (speech, NLP, time series, DNA): the next word depends on previous words; distant elements influence the current state.

RNN fix: hidden states model dependencies; LSTM/GRU gates capture long-range ones (vanishing gradients).

3. Variable-Length Inputs & Outputs

Sentences, audio, and irregularly sampled series differ in length; fixed-dimension models cannot handle them.

RNN fix: step-by-step processing handles any sequence length dynamically.

4. Complexity, Noise & Missing Data

High-dimensional sequences have intricate interdependencies (text, multivariate series, biology), plus sensor noise and gaps in collection.

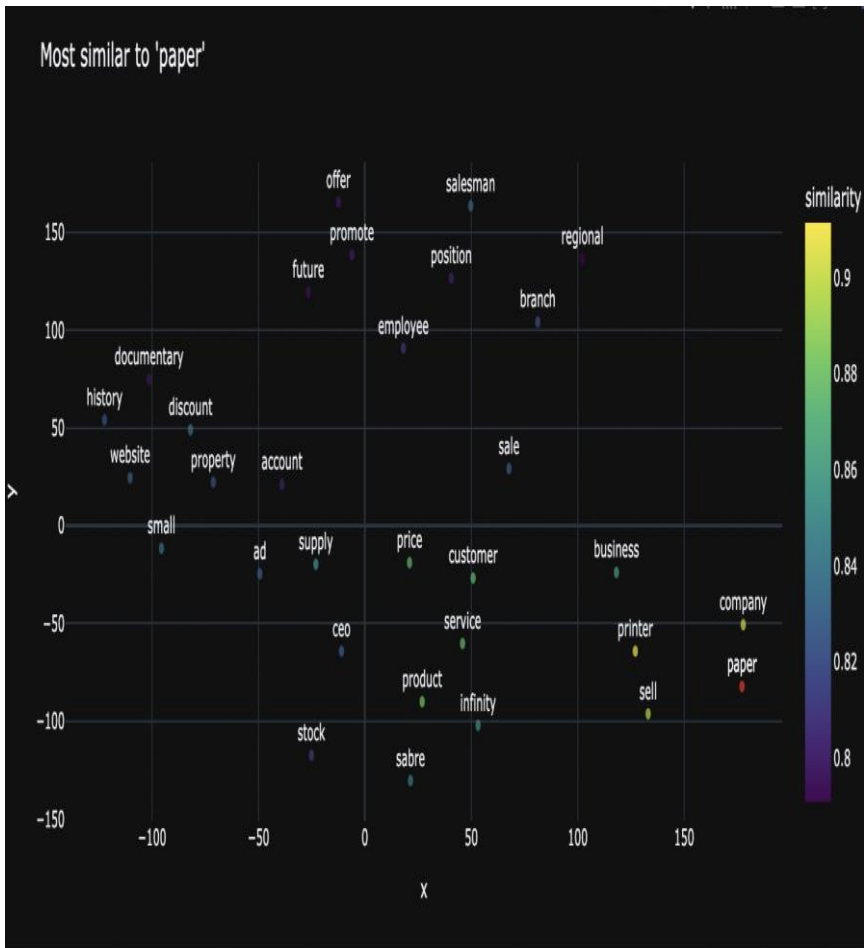
RNN fix: hierarchical representations; aggregation over time is robust to noise and interpolates gaps.

Takeaway: even a trivial task — output YES iff the sequence has an even number of 1s — requires memory. **Recurrence + hidden-state memory** is what lets RNNs capture context over time.

Text Processing for RNN

Step	Task	Description
1	Preprocessing Text	Clean text, tokenize, remove stopwords, and normalize case using NLTK.
2	Build Vocabulary	Assign a unique index to each word using a frequency-based vocabulary.
3	Convert Text to Sequences	Map tokenized words to their corresponding integer indices.
4	Padding Sequences	Standardize input sequence lengths by adding padding tokens where necessary.
5	Dataloader Preparation	Create PyTorch dataset and dataloader for mini-batch training.
6	Load Pretrained Word Embeddings	Use GloVe embeddings (100D) for better semantic representation.
7	Define RNN Model	Construct an RNN with an embedding layer, hidden layers, and output layer.
8	Loss and Optimization	Use Binary Cross-Entropy Loss ('BCEWithLogitsLoss') and Adam optimizer.
9	Train Model	Train the RNN model using mini-batches from the dataloader.
10	Make Predictions	Preprocess new text, convert it to sequences, and run inference using the trained model.

Word Embeddings



Word embeddings are a fundamental technique in NLP. They convert words into dense, continuous vector representations. Word embeddings place similar words closer in vector space. Unlike traditional one-hot encoding, embeddings preserve:

- ❖ **Semantic relationships** between words.
- ❖ **Contextual meaning** of words in sentences.
- ❖ **Word similarity and analogies.**

One-hot encoding.

For vocabulary size V , represent word w by $x(w) \in \{0, 1\}^V$.

Problem 1: No graded similarity.

Thus “cat” vs “dog” has the same similarity as “cat” vs “microwave”.

$$x(w_1)^T x(w_2) = \begin{cases} 1, & w_1 = w_2, \\ 0, & w_1 \neq w_2. \end{cases}$$

Problem 2: High dimensionality. V can be 10^5 – 10^7 .

Embedding matrix:

$$E \in \mathbb{R}^{V \times d}, \quad d \ll V.$$

For token id $i \in \{0, \dots, V - 1\}$, the embedding is the i -th row:

$$e_i = E[i] \in \mathbb{R}^d.$$

Word Embeddings

- We will learn a dense vector for each word such that words appearing in similar contexts have similar vectors, where similarity is measured by the dot (inner) product. These word vectors are also called **word embeddings** or **neural word representations**, and they provide a **distributed representation** of meaning.

$$\cos(e(w_1), e(w_2)) = \frac{e(w_1)^\top e(w_2)}{\|e(w_1)\| \|e(w_2)\|}.$$

$$\text{NN}(w) = \arg \max_{u \in \mathcal{V}} \cos(e(w), e(u)).$$

$\cos(\text{banking}, \text{monetary}) \approx 0.8570$.

banking =

$$\begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

monetary =

$$\begin{pmatrix} 0.413 \\ 0.582 \\ -0.007 \\ 0.247 \\ 0.216 \\ -0.718 \\ 0.147 \\ 0.051 \end{pmatrix}$$

Close (high similarity)

Near-synonyms:

doctor–physician; car–automobile

Same category (co-hyponyms):

cat–dog; apple–orange

Strong association:

hospital–nurse; coffee–cup

Distant (low similarity)

Unrelated concepts (low similarity):

- ❖ banana–democracy;
- ❖ microscope–basketball;
- ❖ piano–chlorine;
- ❖ volcano–spreadsheet;
- ❖ neuron–suitcase

Teaching moments (quirks)

Antonyms can be close:

hot–cold; good–bad

Polysemy mixes senses: “bank” may blend finance + river neighbors

Tip: contextual models (BERT/GPT) resolve meaning using sentence context

Types of Word Embeddings

A. Count / Frequency-based (Distributional, “classical”)

- ❖ **TF-IDF** (sparse weighting; mainly for retrieval/features)
 - ❖ **LSA / LSI** (SVD on term–document or term–context matrix → dense vectors)
 - ❖ **PPMI + SVD** (common “explicit-to-dense” pipeline)
- GloVe** (factorizes a global co-occurrence objective → count-based/matrix factorization)

B. Predictive Neural (Static, context-free)

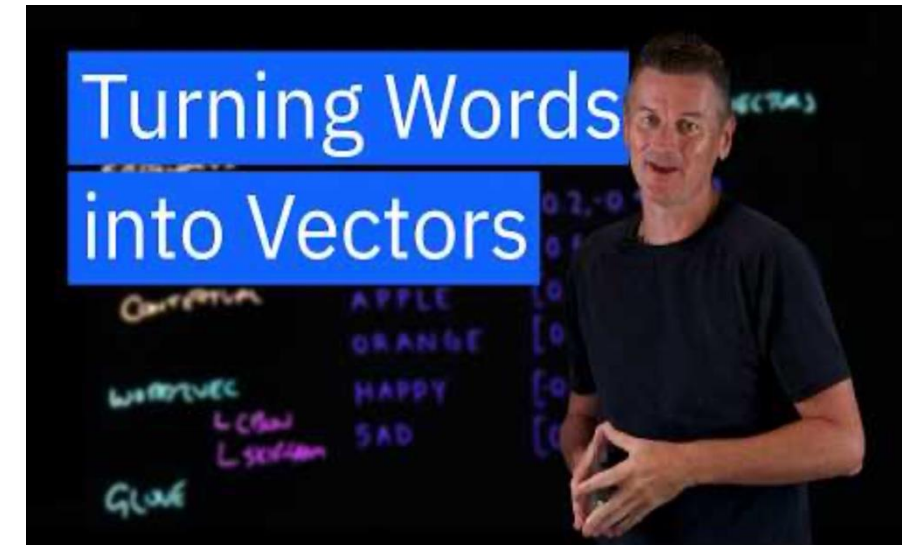
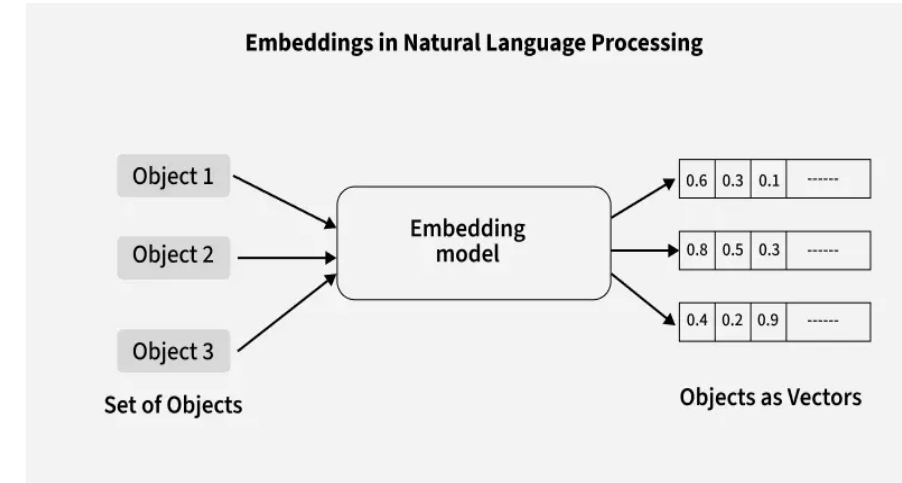
- **Word2Vec**
- **CBOW** (predict center word from context)
- **Skip-gram** (predict context from center word)

C. Subword / Character-aware (Static, better for Out-Of-Vocabulary (OOV))

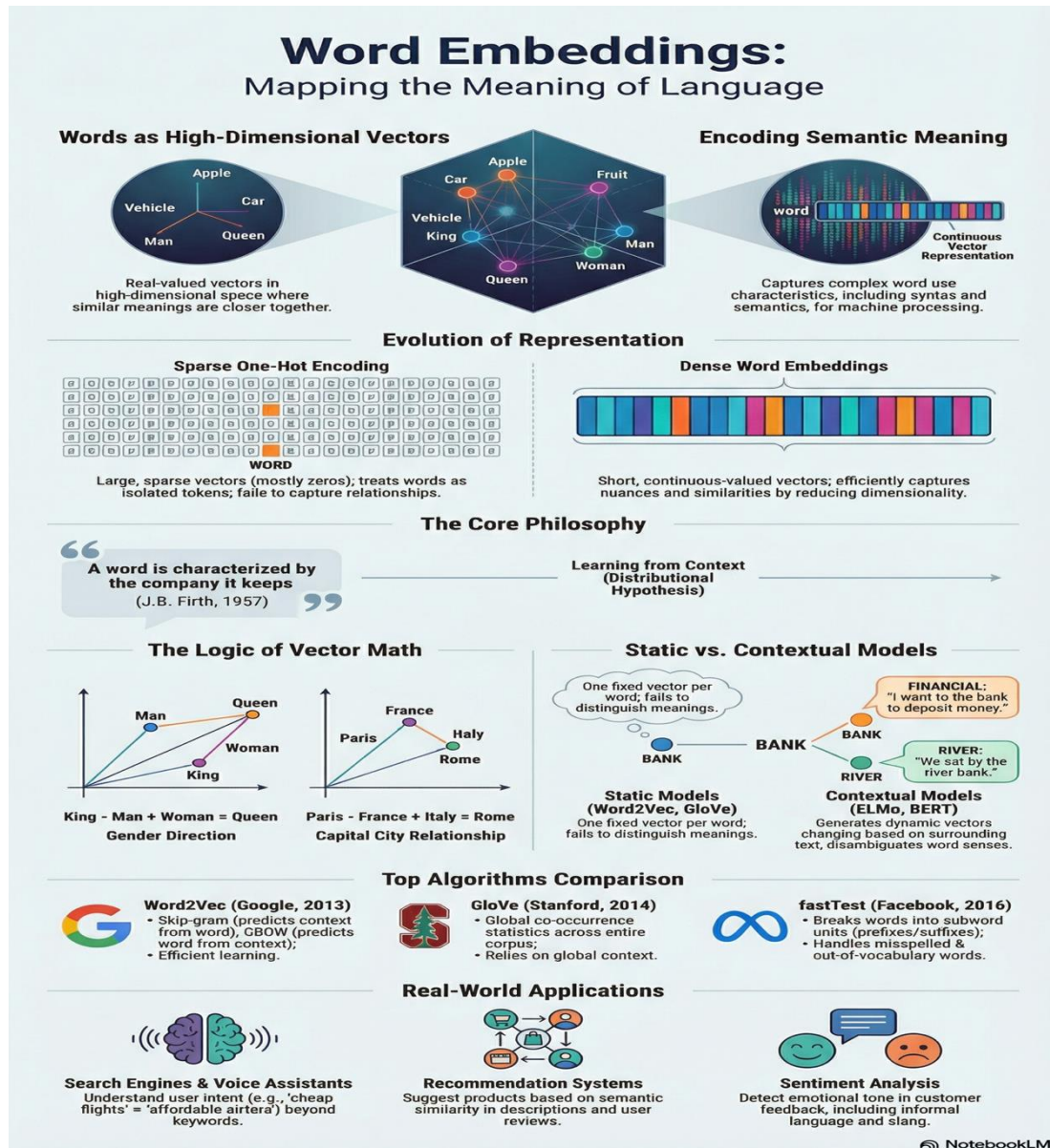
- ❖ **fastText** (word vectors built from character n-grams)

D. Contextual / Transformer-based (Dynamic, context-dependent)

- **BERT / RoBERTa** (masked language modeling; bidirectional context)
- **GPT** (causal language modeling; left-to-right context)



Word Embeddings Dimensions



Word Embeddings	Dimension	Key Features
Word2Vec	50-300	Trained on large corpora like Google News
GloVe	50-300	Uses word co-occurrence statistics
FastText	50-300	Handles subword information
ELMo	1024	Contextual embeddings from bidirectional LSTMs
BERT (base)	768	Transformer-based, context-aware
BERT (large)	1024	More parameters than BERT base
GPT-2 (small)	768	Transformer-based generative model
GPT-2 (medium)	1024	More layers and parameters
GPT-3	12288	High-dimensional transformer model

- ◆ For small models or mobile applications → Use **50-300 dimensions** (Word2Vec, GloVe).
- ◆ For NLP applications with context-awareness → Use **512-1024 dimensions** (BERT, ELMo).
- ◆ For large-scale generative AI → Use **1024+ dimensions** (GPT-3, Transformers).

Word2Vector

Word2Vec is a framework for learning dense word vectors (embeddings) from large text corpora.

Core Idea:

- ▶ Start with a large corpus: a long sequence of tokens $w_1, \dots, w_T$.
- ▶ Assign each vocabulary word a trainable vector representation.
- ▶ For each position t , choose a *center* word $c = w_t$ and its surrounding *context* (*outside*) words

$$o \in \{w_{t-m}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+m}\},$$

where m is the context-window radius.

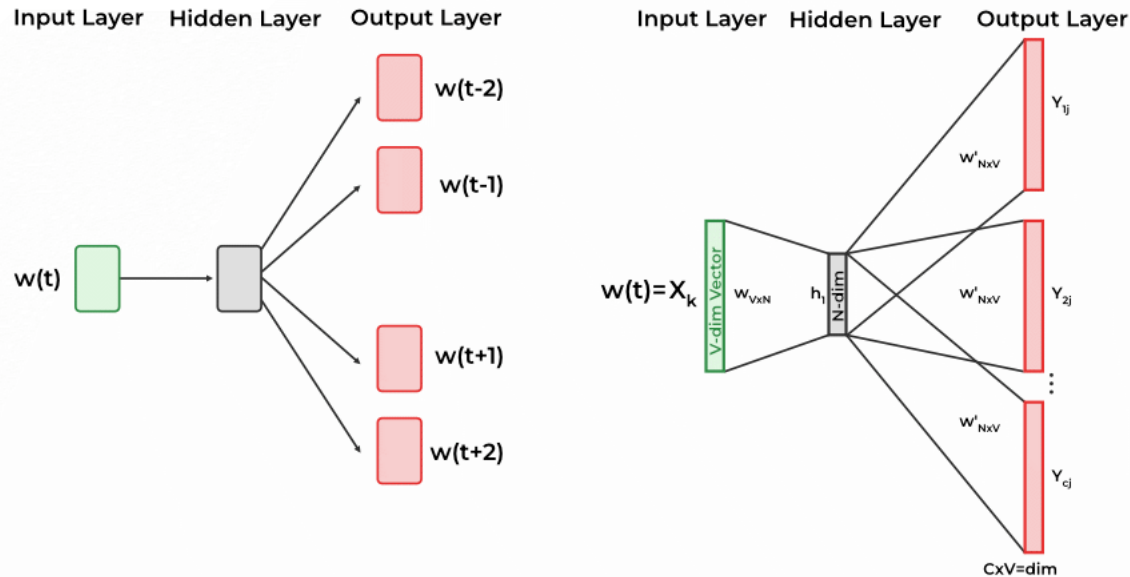
- ▶ Use a similarity score (typically a dot product) between the vectors of c and o to define a conditional probability, e.g., $p(o | c)$ (Skip-gram) or $p(c | \text{context})$ (CBOW).
- ▶ Update the vectors to *maximize the likelihood* of observed center–context pairs across the corpus (often implemented efficiently via negative sampling or hierarchical softmax).

Goldberg, Y., and Levy, O. (2014). "word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method." *arXiv preprint arXiv:1402.3722*.

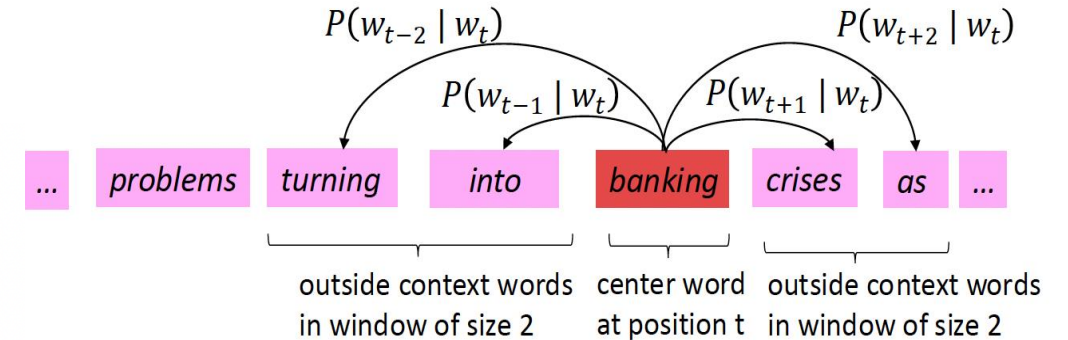
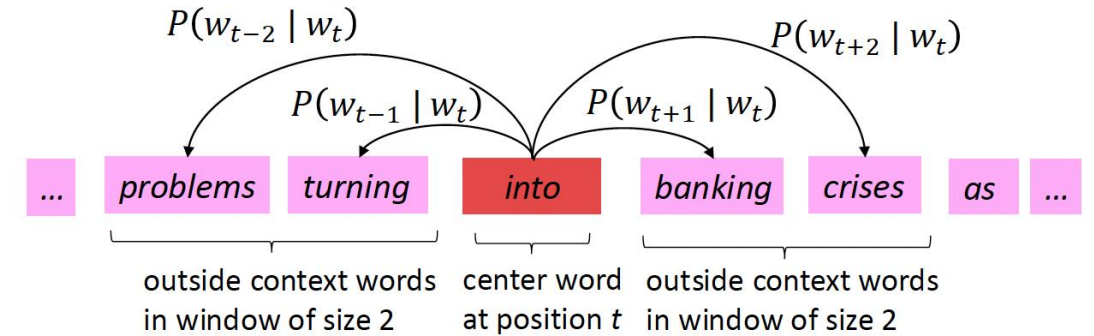
Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). "Distributed Representations of Words and Phrases and their Compositionality." *Advances in Neural Information Processing Systems (NeurIPS)*, 26.

Word2Vector: Skip-Gram Architecture

SKIP-GRAM ARCHITECTURE



Example windows and process for computing



For each position $t = 1, \dots, T$, predict context words within a window of fixed size m , given center word w_t . Data likelihood:

$$\text{Likelihood} = L(\theta) = \prod_{t=1}^T \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta)$$

θ is all variables to be optimized

We want to minimize the objective function:

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^T \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j} | w_t; \theta)$$

StanfordCS224n

Word2Vector

Question: How to calculate $P(w_{t+j} | w_t; \theta)$?

Answer: We will use two vectors per word w :

- v_w when w is a center word
- u_w when w is a context word

Softmax Function

We typically take v_w as the embedding; u_w is the context embedding; optionally use $(v_w + u_w)/2$.

② Exponentiation makes anything positive

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

① Dot product compares similarity of o and c .
 $u^T v = u \cdot v = \sum_{i=1}^n u_i v_i$
Larger dot product = larger probability

③ Normalize over entire vocabulary to give probability distribution

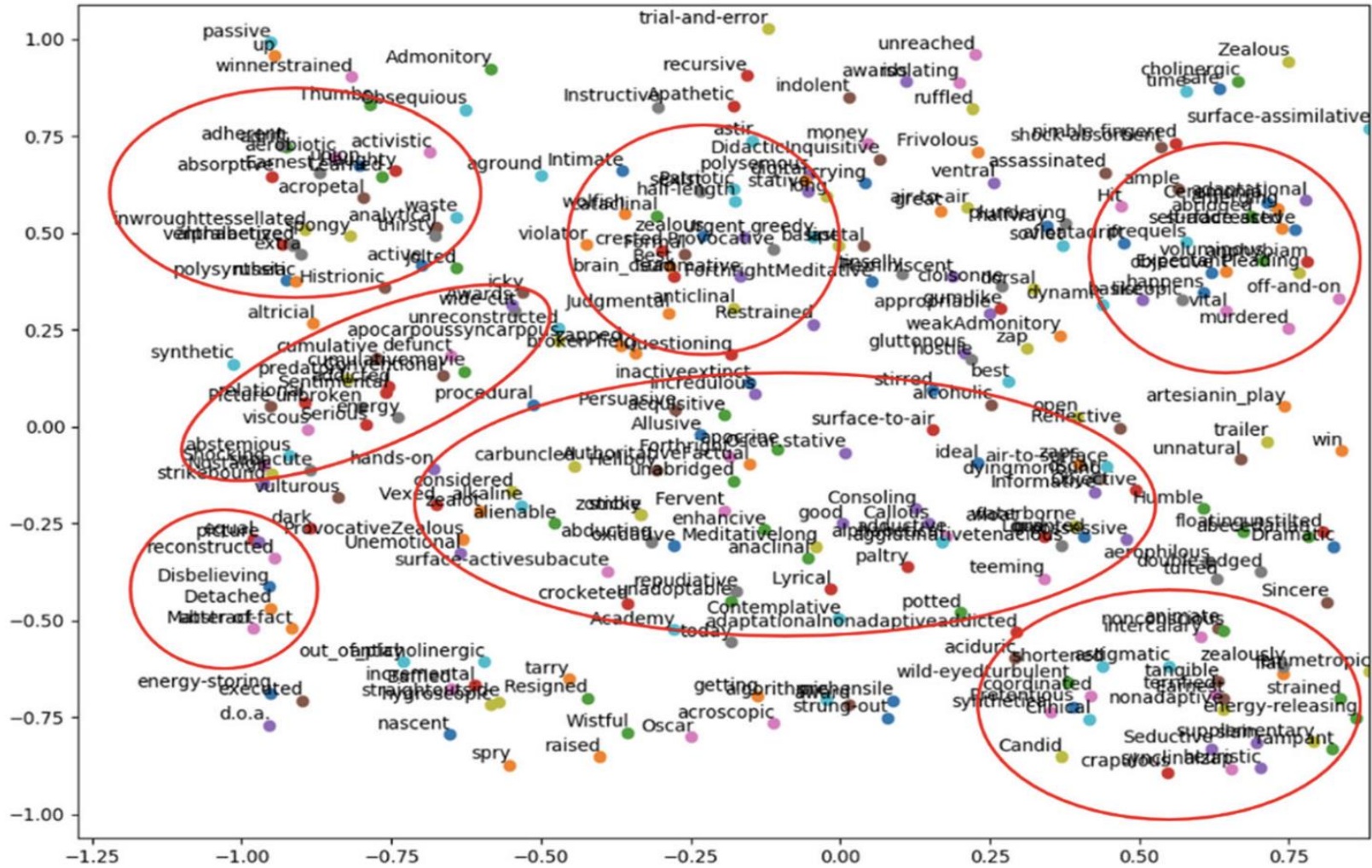
$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \vdots \\ v_{zebra} \\ u_{aardvark} \\ u_a \\ \vdots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$

Why Two Vectors?

- **Modeling Flexibility:** The distribution of contexts given a word ("word as center") does not necessarily match the distribution of words appearing around it ("word as context").
- **Performance:** Having separate v_w and u_w allows the model to capture more nuance, typically leading to higher-quality embeddings.

StanfordCS224n

Word2Vector results

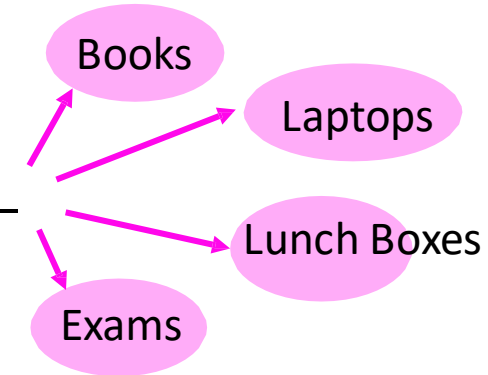


Baek, J. W., & Chung, K. Y. (2021). Multimedia recommendation using Word2Vec-based social relationship mining. *Multimedia Tools and Applications*, 80(1), 1-18.

Formulation: Language Modeling (LM)

A **language model (LM)** is a statistical or machine learning model that **predicts the next word in a sequence** or assigns **probabilities** to sequences of words.

the students opened their



- ✓ Predicts the likelihood of a sequence of words
- ✓ Generates human-like text (e.g., GPT models)
- ✓ Understands context and meaning
- ✓ Enables AI systems to process and generate natural language

More formally: given a sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$: $P(x^{(t+1)} | x^{(t)}, \dots, x^{(1)})$ where $x^{(t+1)}$ can be any word in the vocabulary $V = \{w_1, \dots, w_{|V|}\}$

Example:

- **Input:** "I am going to the"
- **Model prediction:** "store" (80%), "beach" (15%), "moon" (5%)
- The model assigns probabilities and selects the most likely next word.

Language Modeling

- You can also think of a Language Model as a system that **assigns a probability to a piece of text**
- For example, if we have some text $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$ then the probability of this text (according to the Language Model) is:

$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}) = P(\mathbf{x}^{(1)}) \times P(\mathbf{x}^{(2)} | \mathbf{x}^{(1)}) \times \dots \times P(\mathbf{x}^{(T)} | \mathbf{x}^{(T-1)}, \dots, \mathbf{x}^{(1)})$$

$$= \prod_{t=1}^T P(\mathbf{x}^{(t)} | \mathbf{x}^{(t-1)}, \dots, \mathbf{x}^{(1)})$$

This is what our LM provides

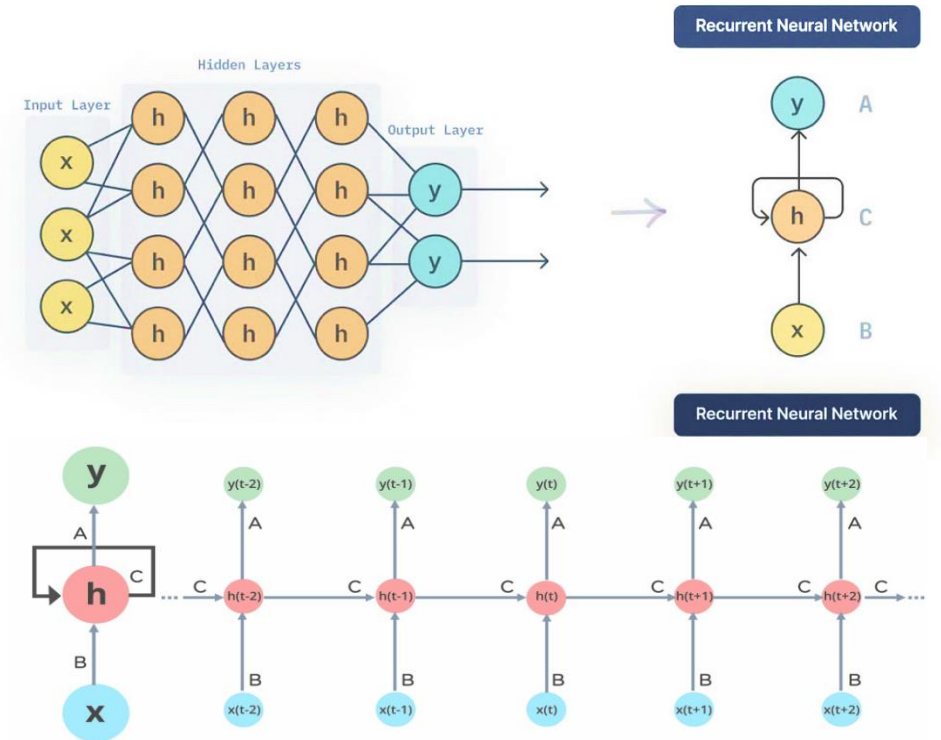
History of Deep RNNs

- **The Rise of Deep RNNs (2010s - Present)**
- **◆ RNNs in NLP and AI**
- **2013** – Google used LSTM for **speech recognition**.
- **2014** – **Seq2Seq Models** (Sutskever et al.) used LSTMs for **machine translation**.
- **2015** – Google Translate adopted LSTMs for **neural machine translation (NMT)**.
- **◆ Attention and Transformers Change the Game**
- **2015** – **Bahdanau et al.** introduced **Attention Mechanisms**, improving Seq2Seq models.
- **2017** – **Vaswani et al.** introduced **Transformers**, replacing RNNs with a more parallelizable model.
- **🔍 Key Concept:**
Transformers like **BERT (2018)**, **GPT-3 (2020)**, and **ChatGPT (2022)** outperformed RNNs, leading to their decline in NLP.

Modern DL Model Architectures

Recurrent Neural Networks (RNNs)

- **Key Features:** Processes sequences of data (time-series data), with memory of previous inputs, capturing temporal dynamics.
- **Unique Feature:** Loop-like architecture allowing previous outputs to be used as inputs while having hidden states, enabling information persistence.
- **Challenges & Solutions:** Problem of vanishing gradients; solved by advanced RNNs, e.g. LSTM and GRU.
- **Example Models:** LSTM (Long Short-Term Memory), GRU (Gated Recurrent Unit).

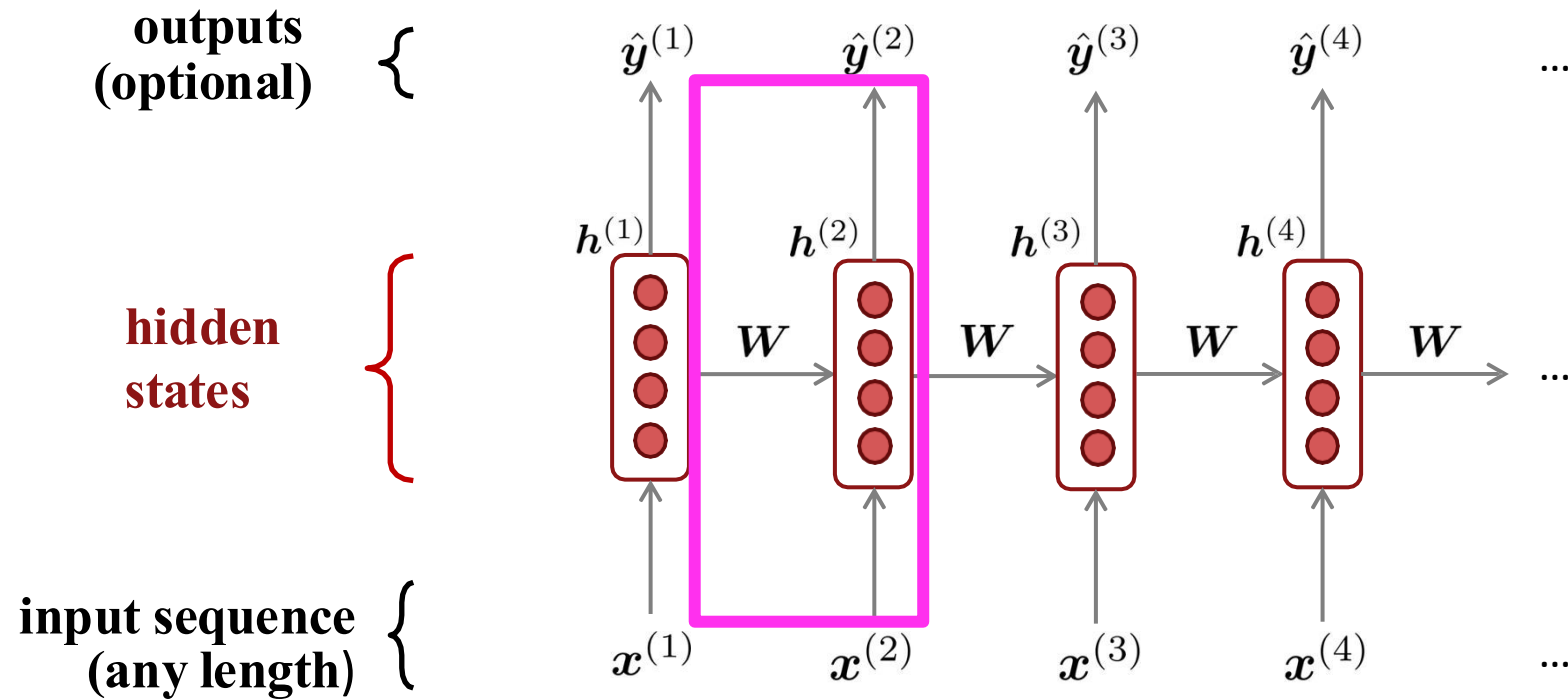


- **Applications in Biomedicine:**
 - Analysis of sequential patient data in EHRs.
 - Time-series analysis in physiological signal processing.

Recurrent Neural Networks (RNN)

A family of neural architectures

Core idea: Apply the same weights W repeatedly



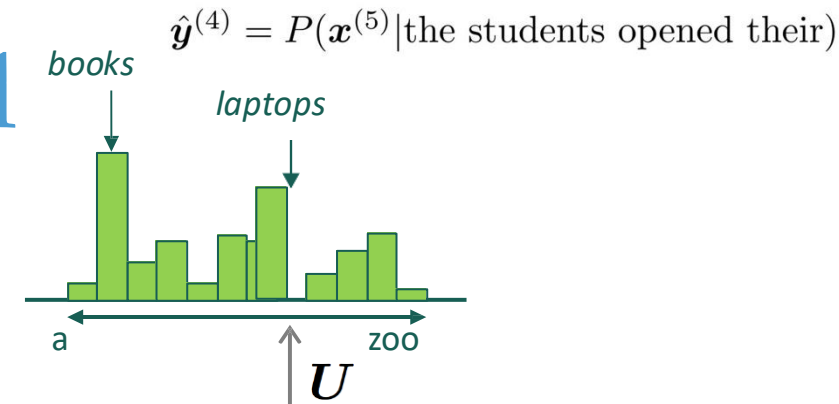
```
nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
```

StanfordCS224n

A Simple RNN Language Model

output distribution

$$\hat{y}^{(t)} = \text{softmax} \left(U h^{(t)} + b_2 \right) \in \mathbb{R}^{|V|}$$



hidden states

$$h^{(t)} = \sigma \left(W_h h^{(t-1)} + W_e e^{(t)} + b_1 \right)$$

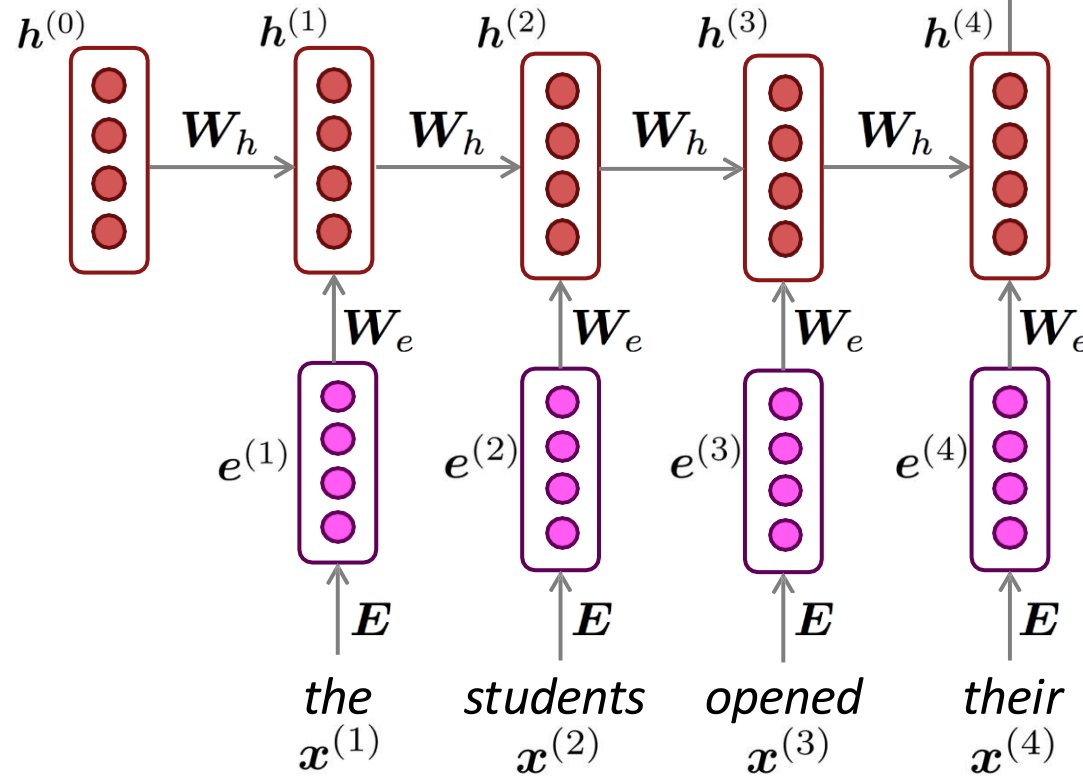
$h^{(0)}$ is the initial hidden state

word embeddings

$$e^{(t)} = E x^{(t)}$$

words / one-hot vectors

$$x^{(t)} \in \mathbb{R}^{|V|}$$



Note: this input sequence could be much longer now!

RNN Language Models

RNN **Advantages**:

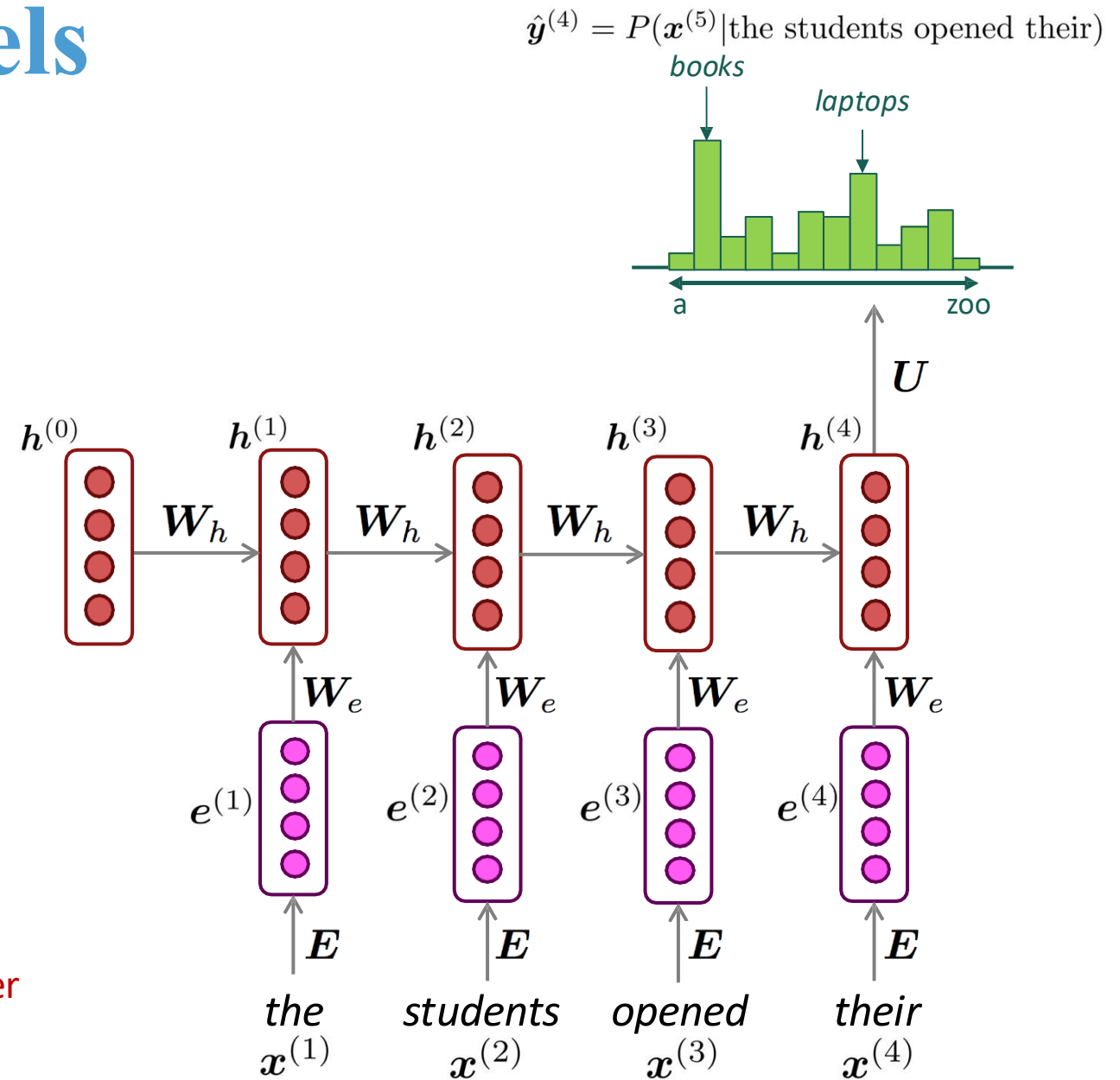
- Can process **any length** input
- Computation for step t can (in theory) use information from **many steps back**
- **Model size doesn't increase** for longer input context
- Same weights applied on every timestep, so there is **symmetry** in how inputs are processed.

RNN **Disadvantages**:

- Recurrent computation is **slow**
- In practice, difficult to access information from **many steps back**

} More on these later

StanfordCS224n



Training an RNN Language Model

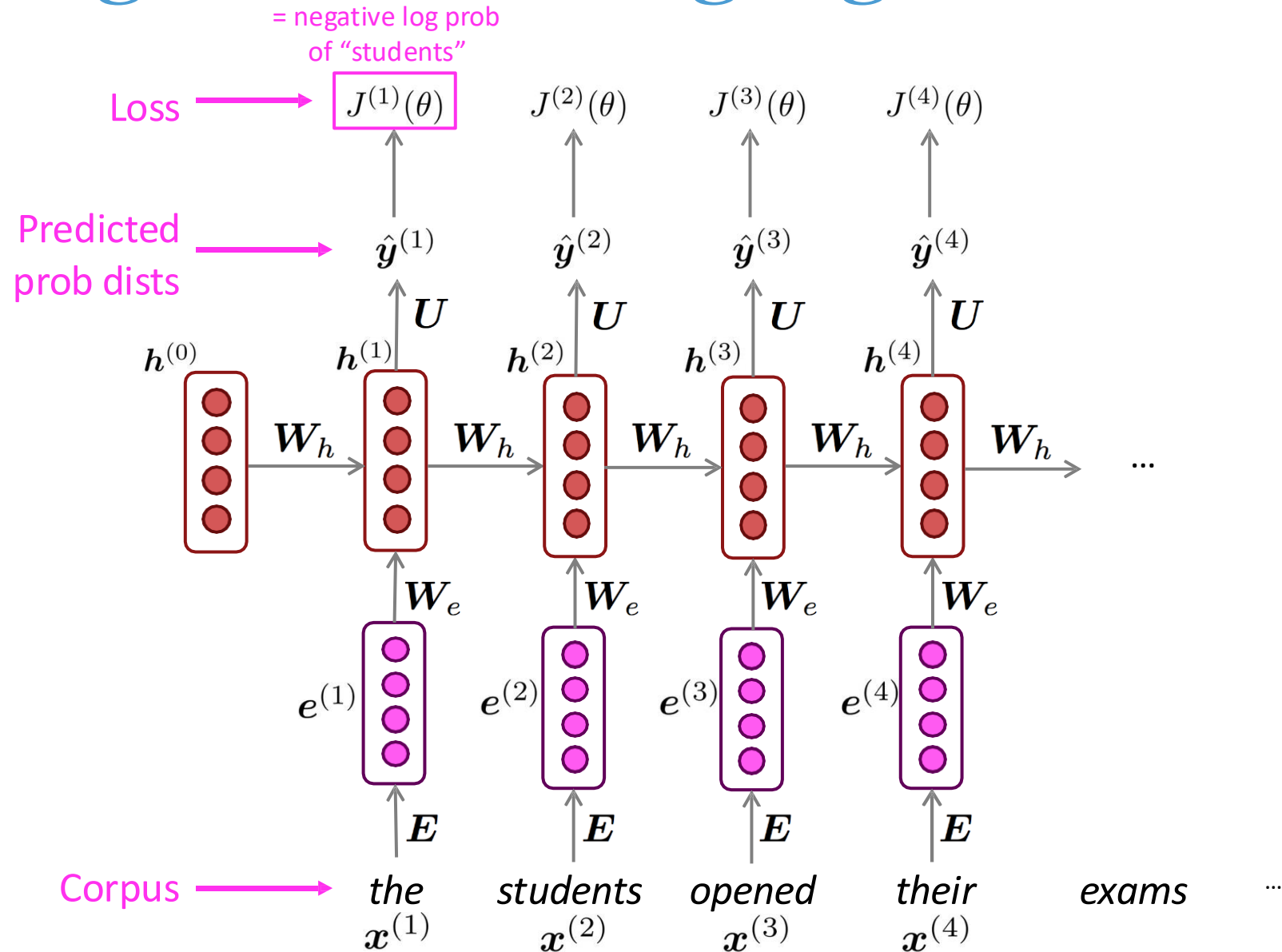
- ❖ Prepare the dataset (tokenize text into words) and convert words to numerical tensors (word embeddings).
- ❖ Build an RNN-LM and compute output distribution $\hat{\mathbf{y}}^{(t)}$ *for every step t* .
- ❖ *Loss function* on step t is *cross-entropy loss* between predicted probability distribution $\hat{\mathbf{y}}^{(t)}$ and the true next word $\mathbf{y}^{(t)}$ (one-hot for $\mathbf{x}^{(t+1)}$):

$$J^{(t)}(\theta) = CE(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) = - \sum_{w \in V} \mathbf{y}_w^{(t)} \log \hat{\mathbf{y}}_w^{(t)} = - \log \hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)}$$

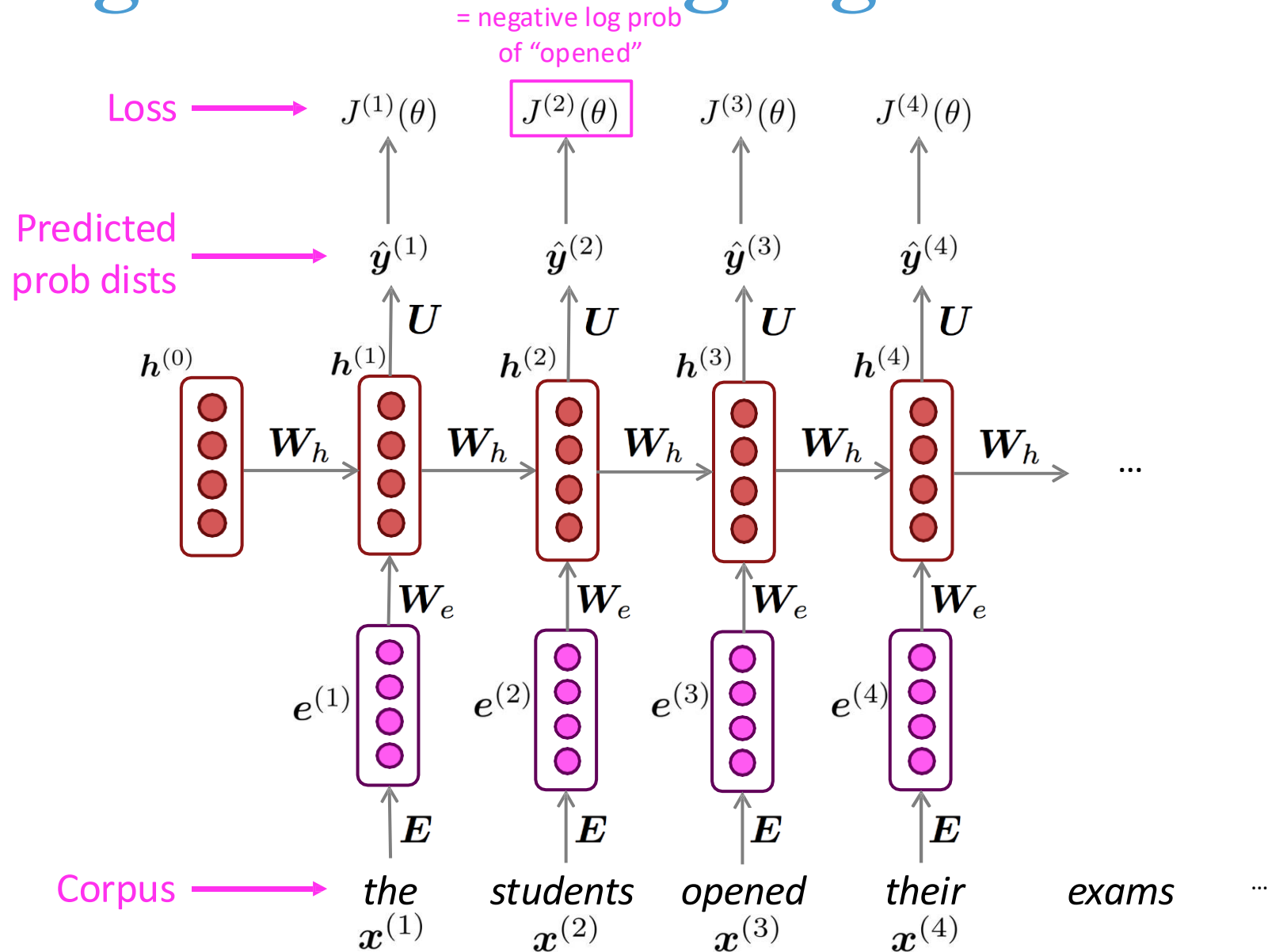
- ❖ Average this to get the *overall loss* for entire training set:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^T J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^T - \log \hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)}$$

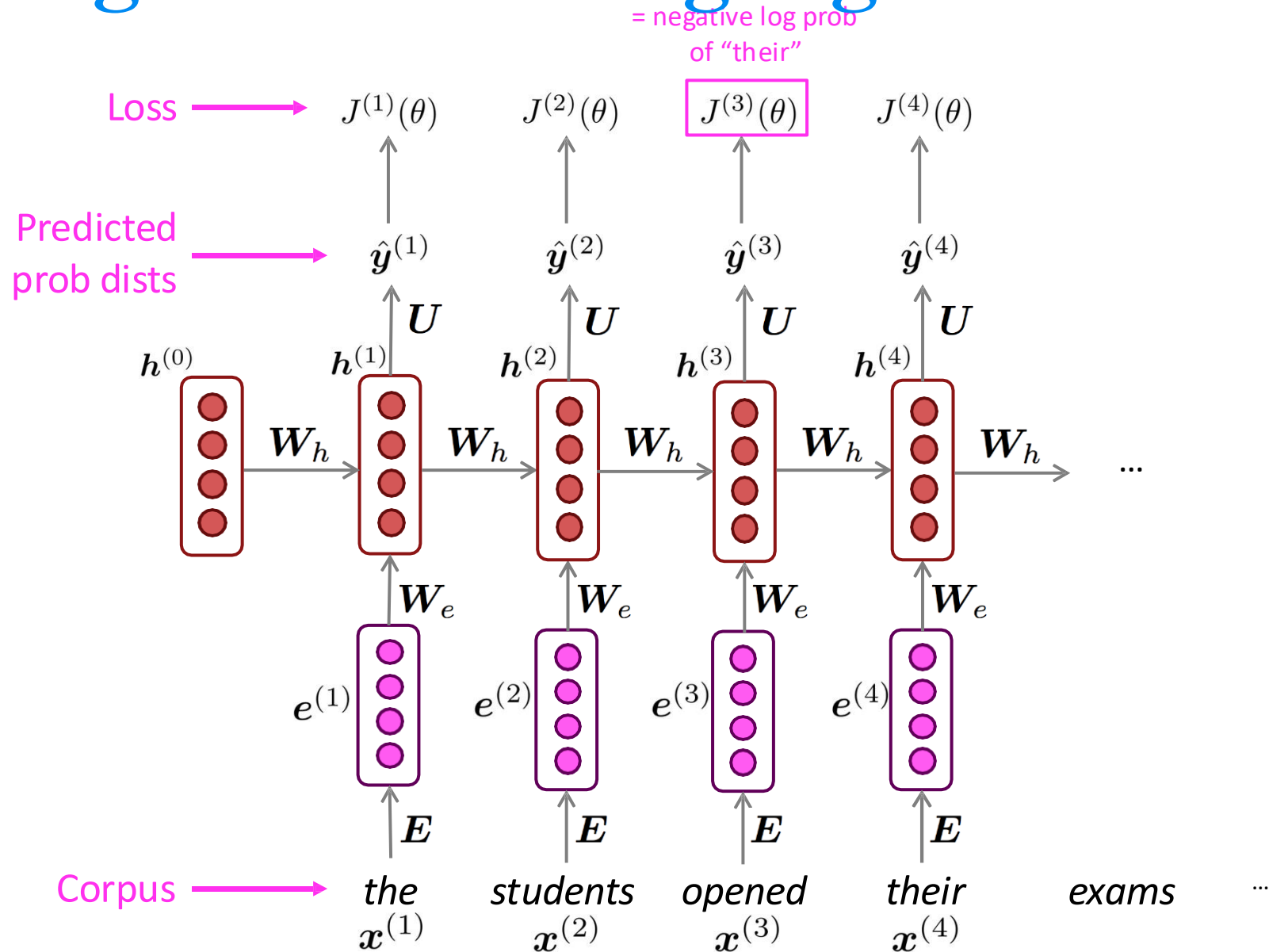
Training an RNN Language Model



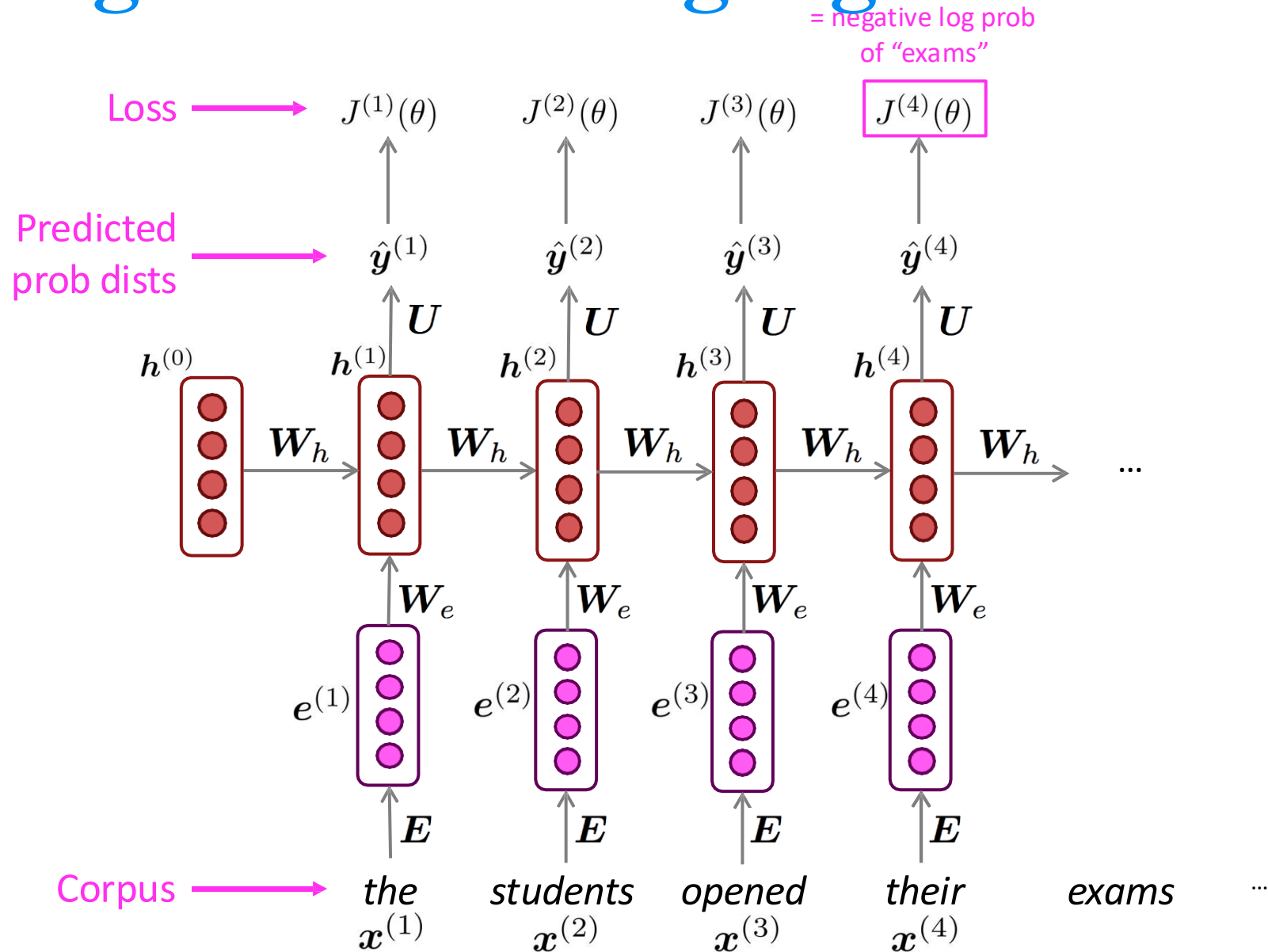
Training an RNN Language Model



Training an RNN Language Model

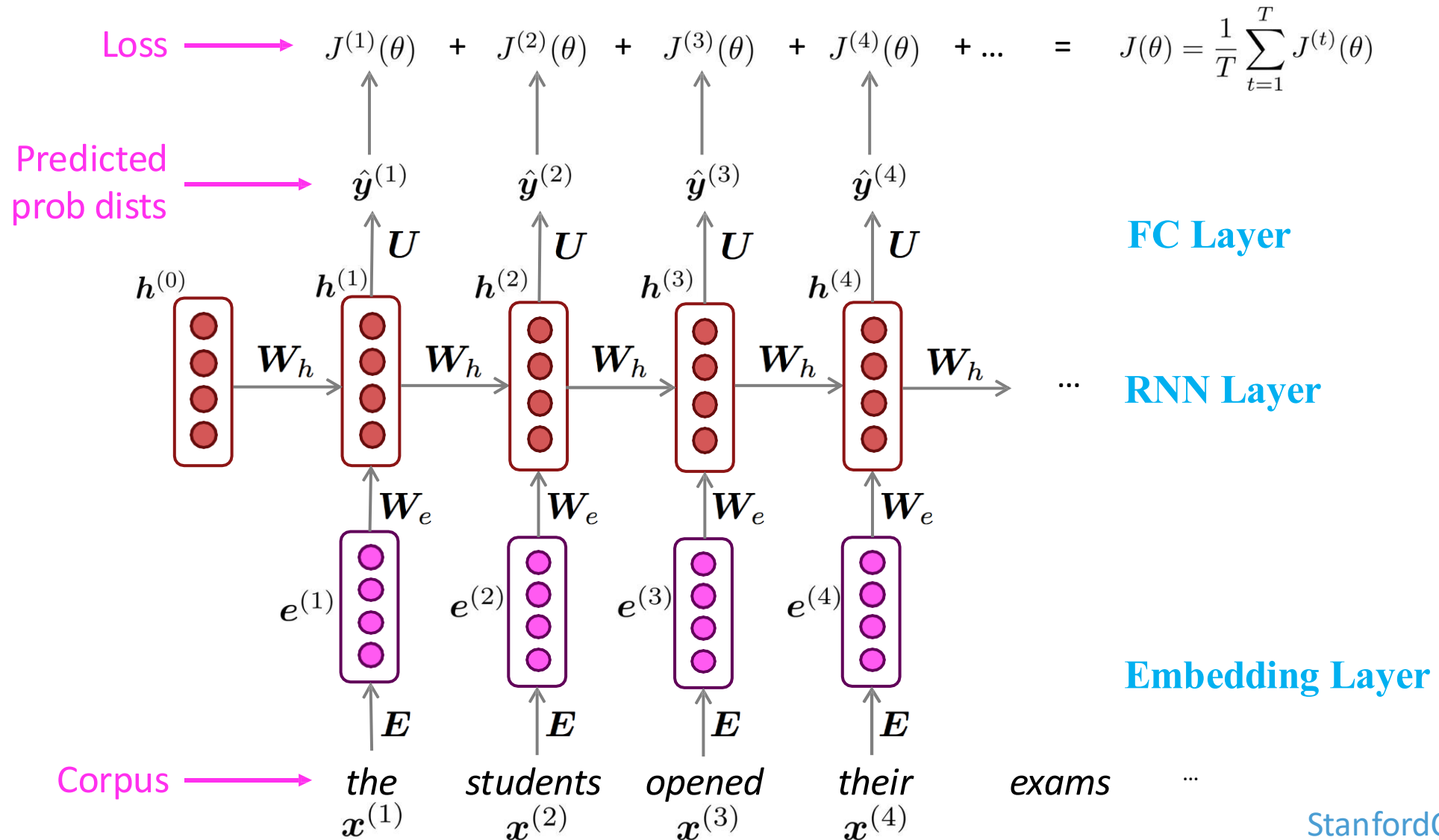


Training an RNN Language Model



Training an RNN Language Model

“Teacher forcing”



StanfordCS224n

Training Challenges

- **Memory Constraints:** Storing all word embeddings, gradients, and activations requires enormous memory.
- **Computational Cost:** Performing **backpropagation over the entire dataset** in a single step is impractical.
- **Batching is Required:** Instead of processing all data at once, models use **mini-batches** to update weights efficiently.

$$J(\theta) = \frac{1}{T} \sum_{t=1}^T J^{(t)}(\theta)$$

Challenges

Too much memory usage

Long sequences overflow memory

Exploding gradients

Slow training

Solution

✓ Mini-batch training

✓ Truncated BPTT (TBPTT)

✓ Gradient clipping

✓ Efficient batching and parallelization

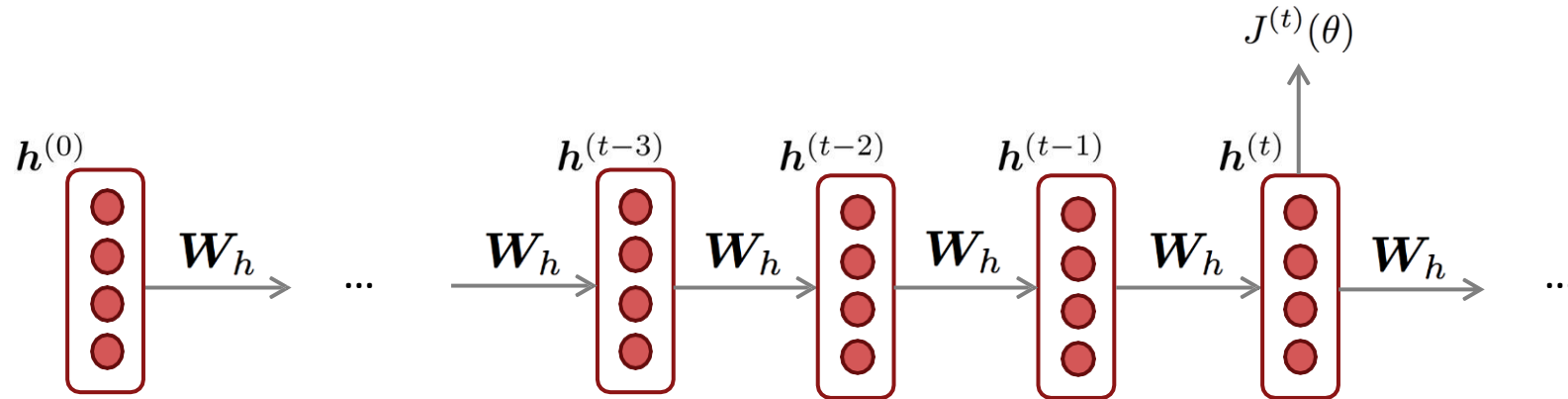
Divide the **entire dataset** into **mini-batches** (e.g., **batch size = 32**).

Truncate the sequence into smaller **sub-sequences** (e.g., **20 time steps** at a time).

Clip gradients to a maximum norm (e.g., **5**).

$$g = g \times \frac{C}{\max(\|g\|, C)}$$

Backpropagation for RNNs



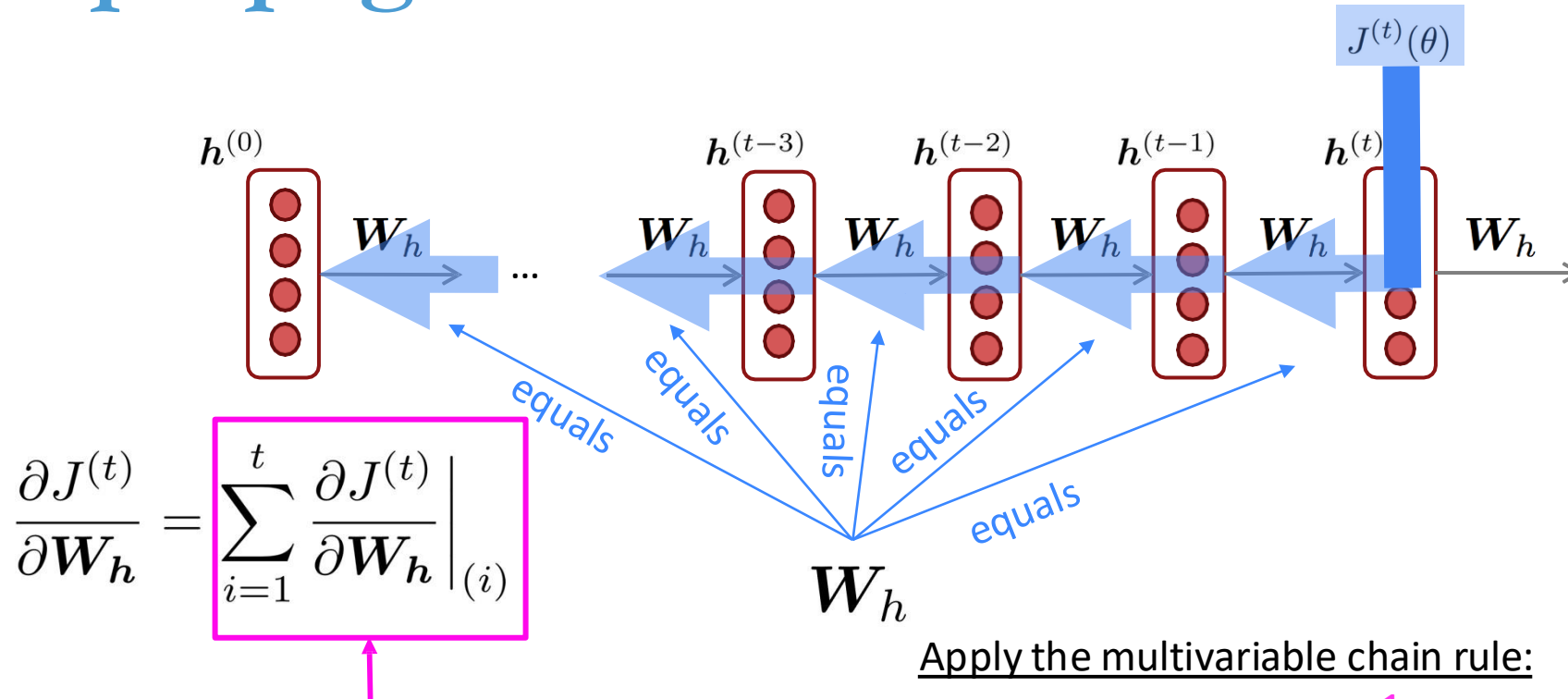
Question: What's the derivative of $J^{(t)}(\theta)$ w.r.t. the **repeated** weight matrix W_h ?

Answer:
$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^t \left. \frac{\partial J^{(t)}}{\partial W_h} \right|_{(i)}$$

“The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears”

Why?

Backpropagation for RNNs



In practice, often “truncated” after ~20 timesteps for training efficiency reasons

Question: How do we calculate this?

Answer: Backpropagate over timesteps $i = t, \dots, 0$, summing gradients as you go. This algorithm is called “**backpropagation through time**” [Werbos, P.G., 1988, *Neural Networks 1*, and others]

Apply the multivariable chain rule:

$$\begin{aligned} \frac{\partial J^{(t)}}{\partial W_h} &= \sum_{i=1}^t \left. \frac{\partial J^{(t)}}{\partial W_h} \right|_{(i)} \frac{\partial W_h}{\partial W_h} \bigg|_{(i)} \\ &= \sum_{i=1}^t \left. \frac{\partial J^{(t)}}{\partial W_h} \right|_{(i)} \end{aligned}$$

= 1

RNNs greatly improved perplexity

n-gram model →

Increasingly
complex RNNs ↓

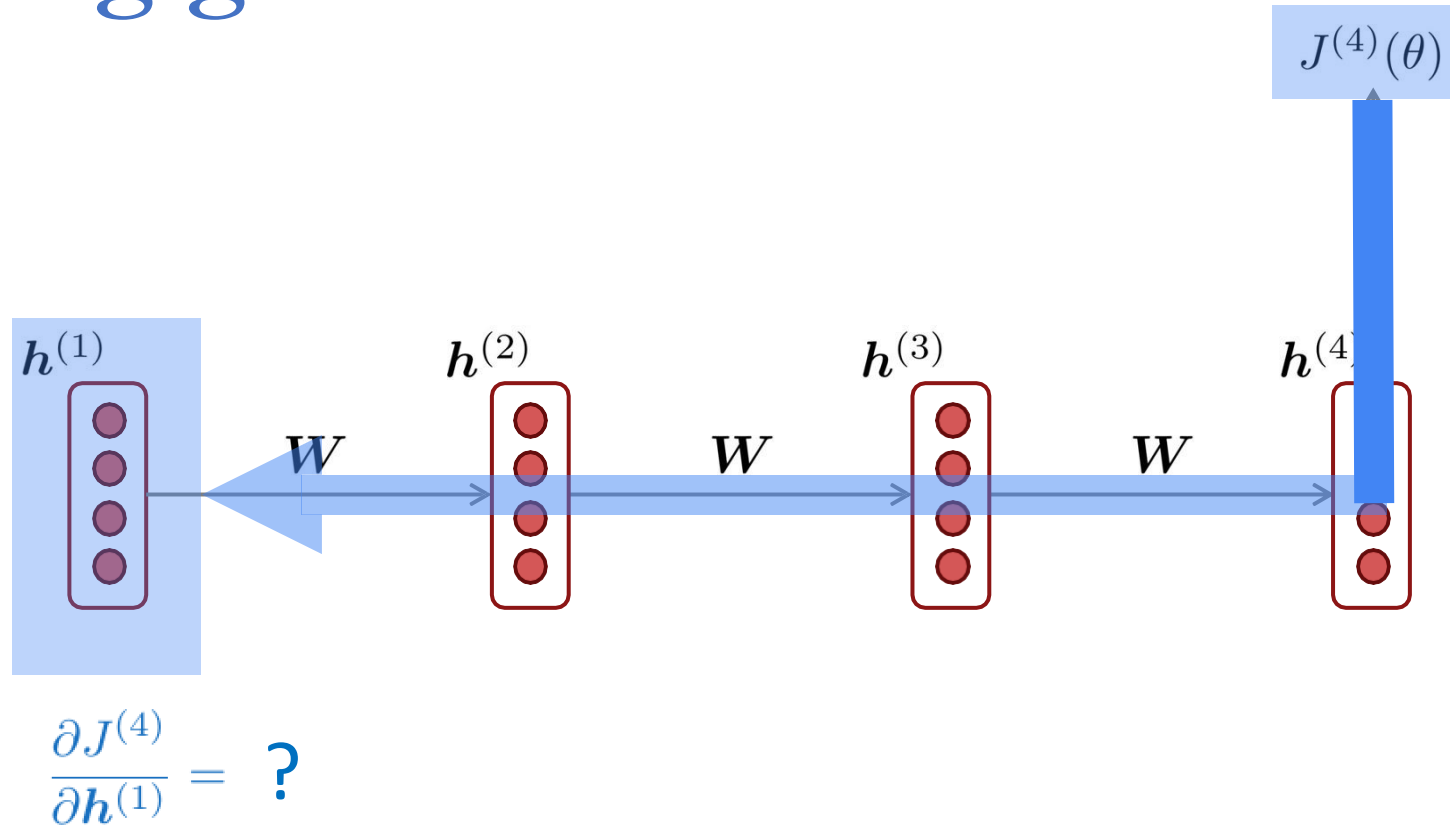
Model	Perplexity
Interpolated Kneser-Ney 5-gram (Chelba et al., 2013)	67.6
RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013)	51.3
RNN-2048 + BlackOut sampling (Ji et al., 2015)	68.3
Sparse Non-negative Matrix factorization (Shazeer et al., 2015)	52.9
LSTM-2048 (Jozefowicz et al., 2016)	43.7
2-layer LSTM-8192 (Jozefowicz et al., 2016)	30
Ours small (LSTM-2048)	43.9
Ours large (2-layer LSTM-2048)	39.8

Perplexity improves
(lower is better)

Source: <https://research.fb.com/building-an-efficient-neural-language-model-over-a-billion-words/>

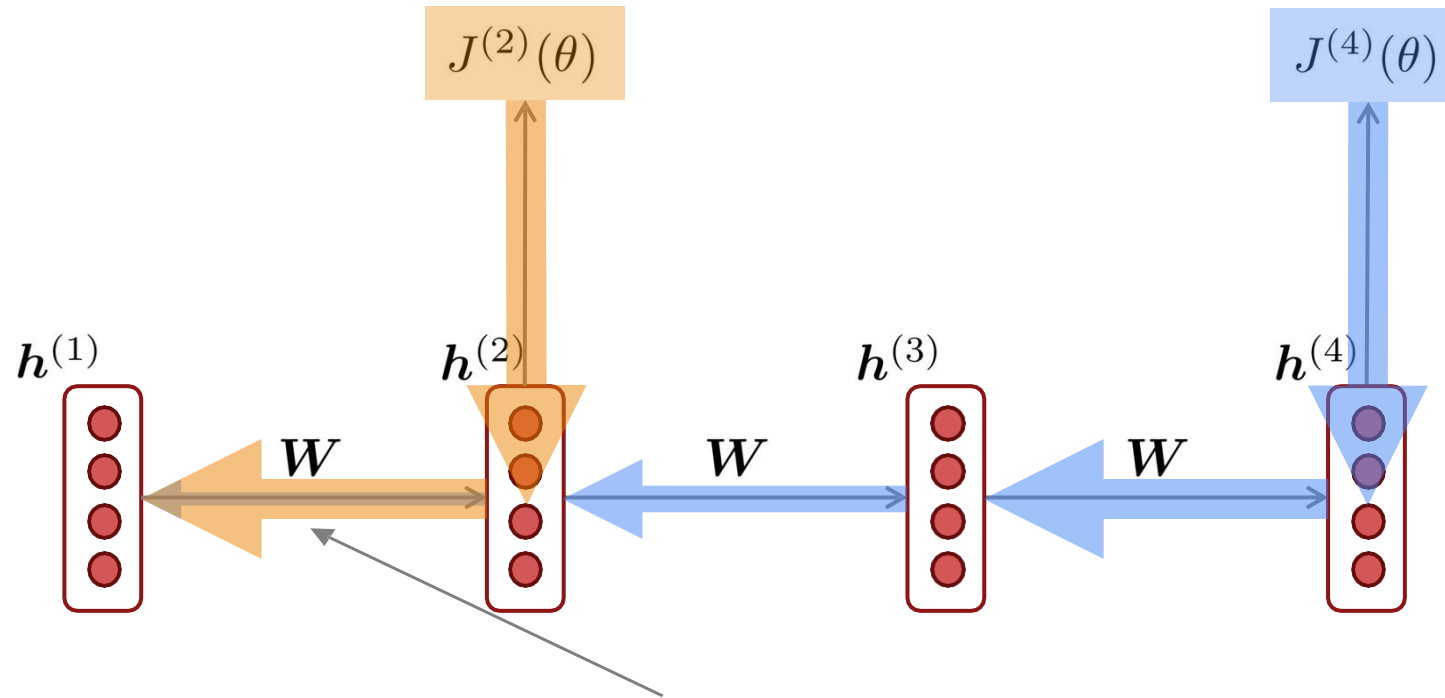
StanfordCS224n

Vanishing gradient intuition



- ▶ This decay makes it difficult for RNNs to learn long-term dependencies.
- ▶ Empirical evidence shows rapid gradient norm decay.

Why is vanishing gradient a problem?



- **Short-Range vs. Long-Range:** Imagine a long chain of dominos where each domino represents a layer or time step. If the force transferred from one domino to the next diminishes (say by a constant factor each time), then after a long chain, the force reaching the first domino becomes nearly zero. This means the first few dominos (or layers) barely "feel" the impact of the initial force (or error), and thus they do not adjust effectively based on long-term dependencies.
- **Resulting Behavior:** The model ends up "paying attention" only to the parts of the sequence that are immediately relevant (the nearby gradient signals) while ignoring distant context. This leads to challenges such as:
 - ❖ Inability to learn relationships or dependencies that span many time steps.
 - ❖ Poor performance on tasks that require integrating information over long sequences.

Long Short-Term Memory RNNs (LSTMs)

- At each time step t , the LSTM maintains a **hidden state** $h(t)$ and a **cell state** $c(t)$, both vectors of length n .
- The cell state stores **long-term information**, while the hidden state represents **the immediate output**.
- Three gates—the **forget gate**, **input gate**, and **output gate**—dynamically control the **erasing, reading, and writing** of information in the cell state.
- Each gate is computed as a vector with values between 0 and 1, where **values indicate the proportion of information to keep or update**.
- This gating mechanism allows the LSTM to **selectively maintain important information over long sequences**, effectively addressing the vanishing gradient problem encountered in traditional RNNs.

This dynamic gating system is a key innovation that makes LSTMs powerful for tasks requiring long-term memory, such as language modeling, speech recognition, and time-series prediction.

Long Short-Term Memory (LSTM)

A sequence of inputs $\mathbf{x}^{(t)}$. Compute a sequence of hidden states $\mathbf{h}^{(t)}$ and cell states $\mathbf{c}^{(t)}$. On timestep t :

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

New cell content: this is the new content to be written to the cell

Cell state: erase (“forget”) some content from last cell state, and write (“input”) some new cell content

Hidden state: read (“output”) some content from the cell

Sigmoid function: all gate values are between 0 and 1

$$\mathbf{f}^{(t)} = \sigma \left(\mathbf{W}_f \mathbf{h}^{(t-1)} + \mathbf{U}_f \mathbf{x}^{(t)} + \mathbf{b}_f \right)$$

$$\mathbf{i}^{(t)} = \sigma \left(\mathbf{W}_i \mathbf{h}^{(t-1)} + \mathbf{U}_i \mathbf{x}^{(t)} + \mathbf{b}_i \right)$$

$$\mathbf{o}^{(t)} = \sigma \left(\mathbf{W}_o \mathbf{h}^{(t-1)} + \mathbf{U}_o \mathbf{x}^{(t)} + \mathbf{b}_o \right)$$

$$\tilde{\mathbf{c}}^{(t)} = \tanh \left(\mathbf{W}_c \mathbf{h}^{(t-1)} + \mathbf{U}_c \mathbf{x}^{(t)} + \mathbf{b}_c \right)$$

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \tilde{\mathbf{c}}^{(t)}$$

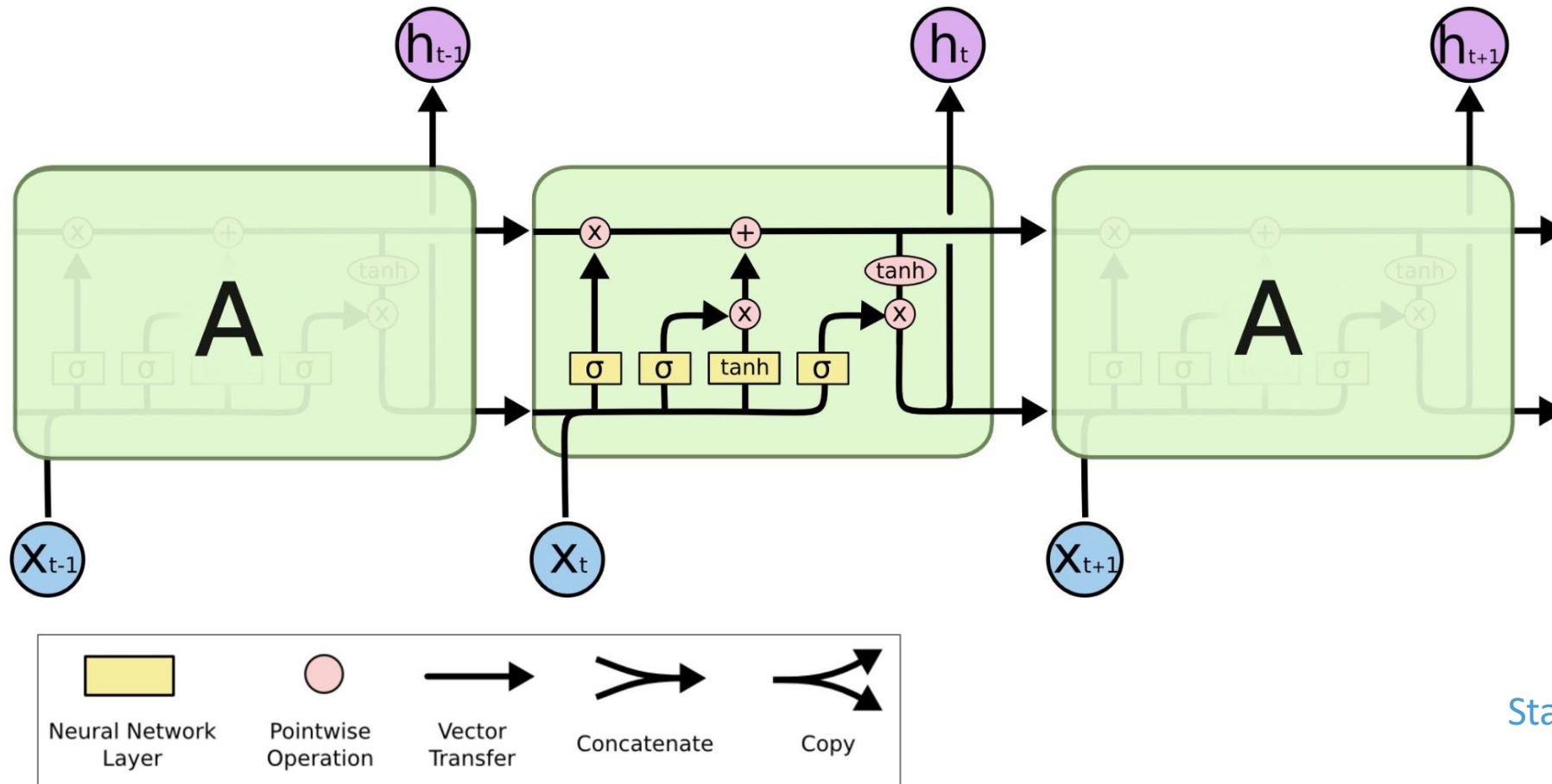
$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh \mathbf{c}^{(t)}$$

All these are vectors of same length n

Gates are applied using element-wise (or Hadamard) product: $\odot$

Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:

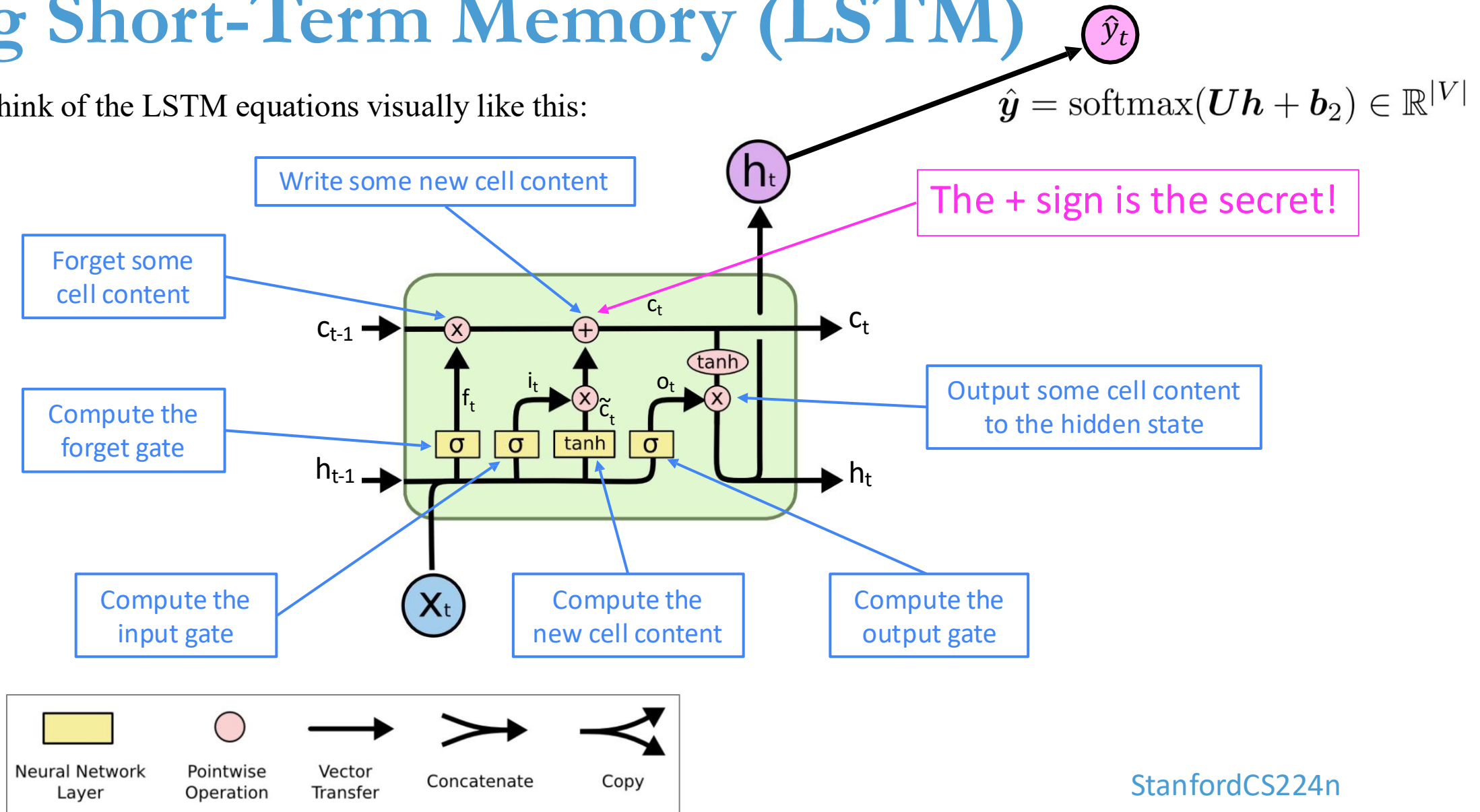


StanfordCS224n

Source: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:



StanfordCS224n

Source: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

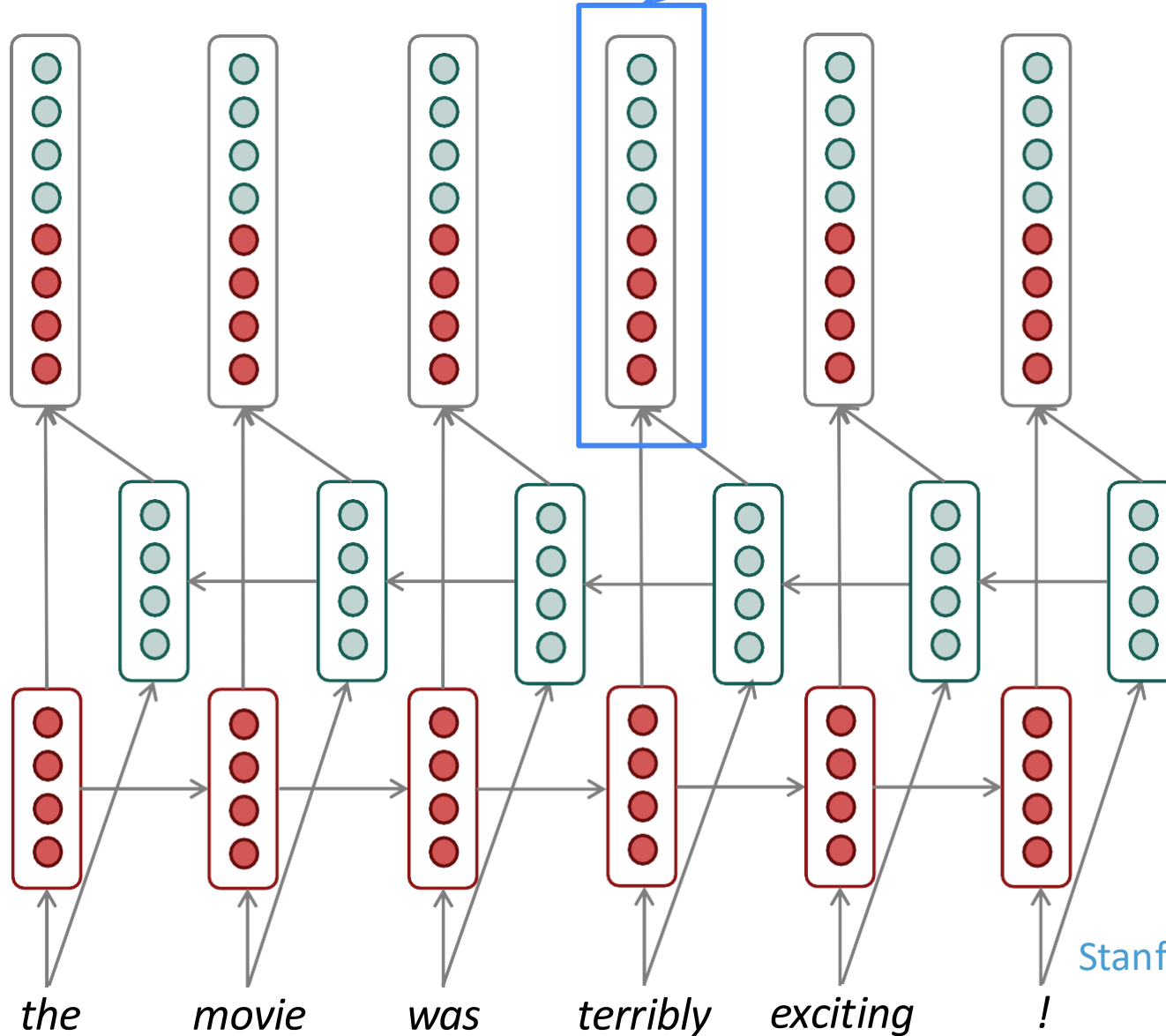
Bidirectional RNNs

This contextual representation of “terribly” has both left and right context!

Concatenated
hidden states

Backward RNN

Forward RNN



StanfordCS224n

Bidirectional RNNs

On timestep t :

This is a general notation to mean “compute one forward step of the RNN” – it could be a simple RNN or LSTM computation.

Forward RNN $\vec{h}^{(t)} = \text{RNN}_{\text{FW}}(\vec{h}^{(t-1)}, \mathbf{x}^{(t)})$

Backward RNN $\overleftarrow{h}^{(t)} = \text{RNN}_{\text{BW}}(\overleftarrow{h}^{(t+1)}, \mathbf{x}^{(t)})$

Generally, these two RNNs have separate weights

Concatenated hidden states $\mathbf{h}^{(t)} = [\vec{h}^{(t)}; \overleftarrow{h}^{(t)}]$

We regard this as “the hidden state” of a bidirectional RNN. This is what we pass on to the next parts of the network.

StanfordCS224n

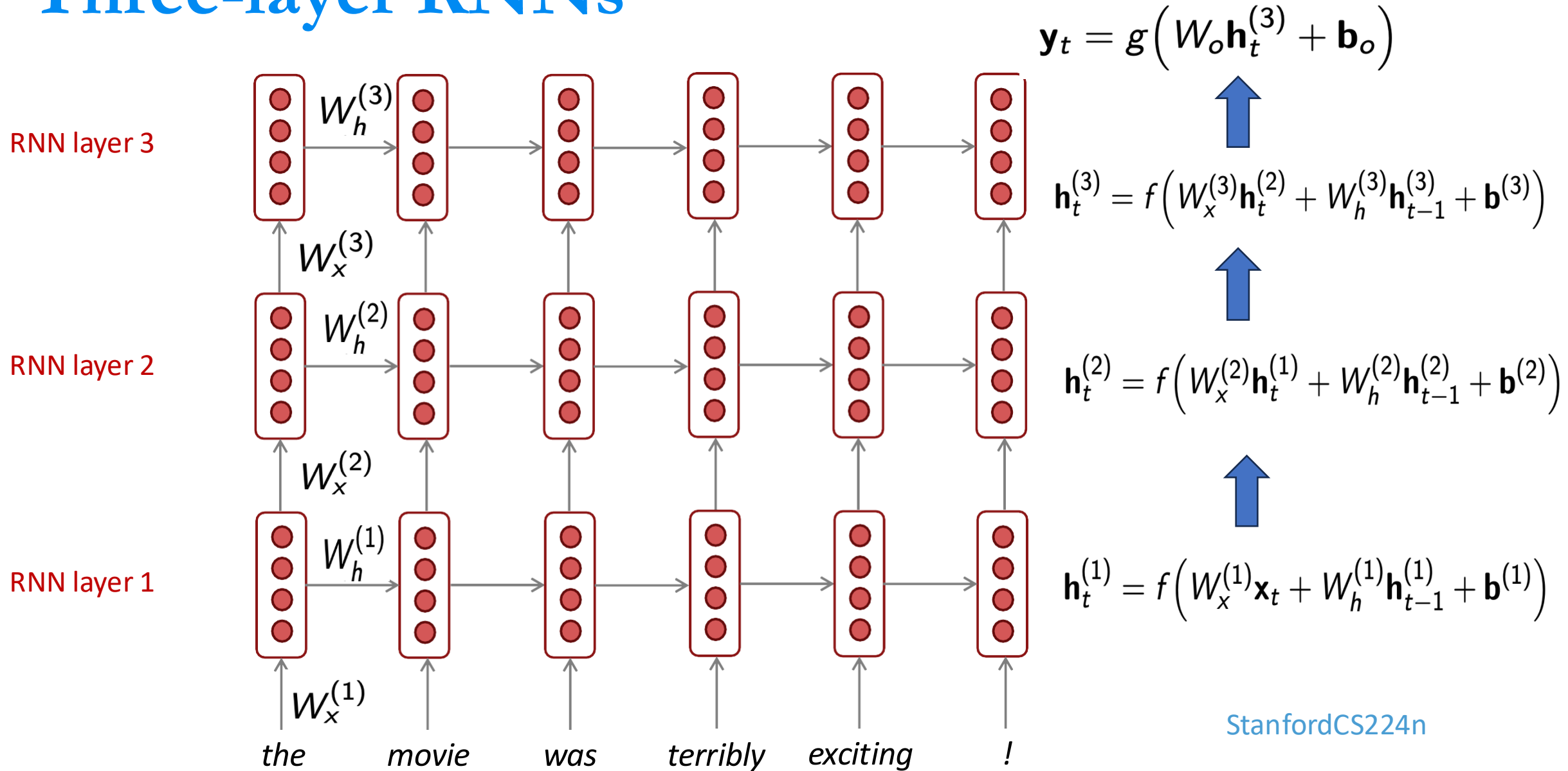
Stacked RNNs

Stacked RNNs (also known as deep RNNs): multiple recurrent layers are placed on top of each other.

- ▶ The output of each RNN layer serves as the input to the next, creating a hierarchical representation of the sequential data. The **lower RNNs** should **compute lower-level features** and the **higher RNNs** should compute **higher-level features**.
- ▶ This architecture is used to capture complex, abstract temporal patterns. Hierarchical layers can learn more abstract features. Better capture complex temporal dependencies.
- ▶ Improved Performance: Often achieve higher accuracy on tasks such as language modeling, speech recognition, and time series prediction.
- ▶ However, they also introduce challenges such as increased training complexity and computational cost.
- ▶ With careful design and tuning, stacked RNNs are a powerful tool for sequence modeling.



Three-layer RNNs



StanfordCS224n

Why Transformers?

Parallel Computation + Short Path Length:

- ▶ Self-attention can handle all tokens in parallel.
- ▶ Minimal path length for global dependencies, vital for deep architectures.

Transformer Dominance in NLP:

- ▶ Nearly all state-of-the-art language tasks use Transformer based models.
- ▶ Default approach: "Grab a large pretrained Transformer" (BERT, GPT, T5, etc.).

Vision Transformer (ViT):

- ▶ Patch-based input turned into token embeddings.
- ▶ Now a go-to model for image recognition, detection, and segmentation.

Islam, S., Elmekki, H., Elsebai, A., Bentahar, J., Drawel, N., Rjoub, G., & Pedrycz, W. (2024). A comprehensive survey on applications of transformers for deep learning tasks. *Expert Systems with Applications*, 241, 122666.

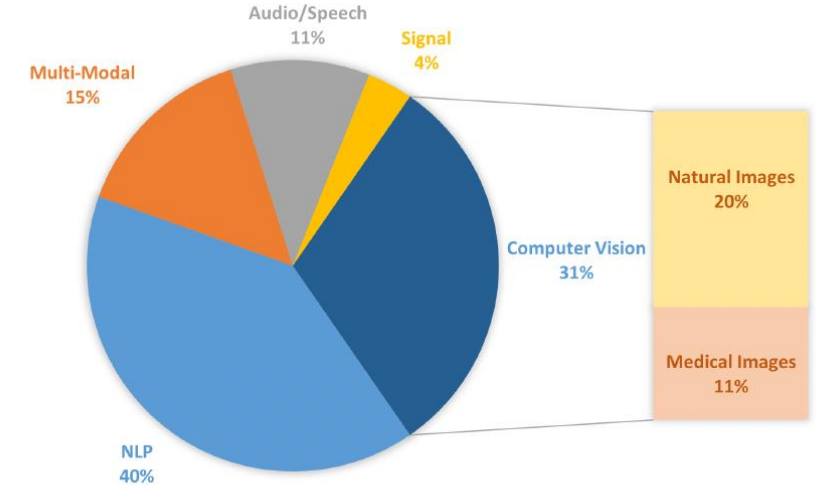
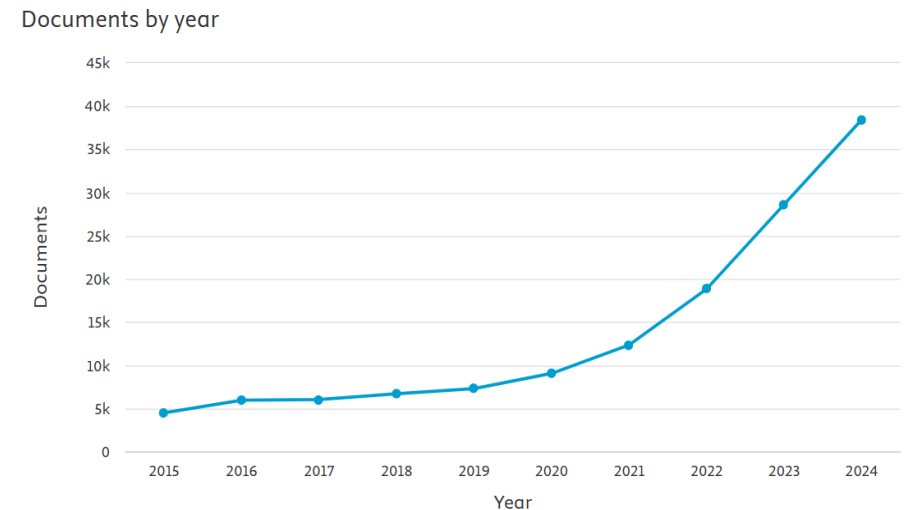


Figure 4: Proportion of transformer application in Top-5 fields



Transformers are Everywhere Now!



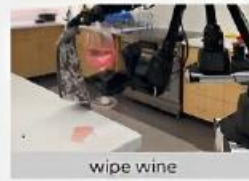
cook shrimp



push chairs



use cabinet



wipe wine



call elevator

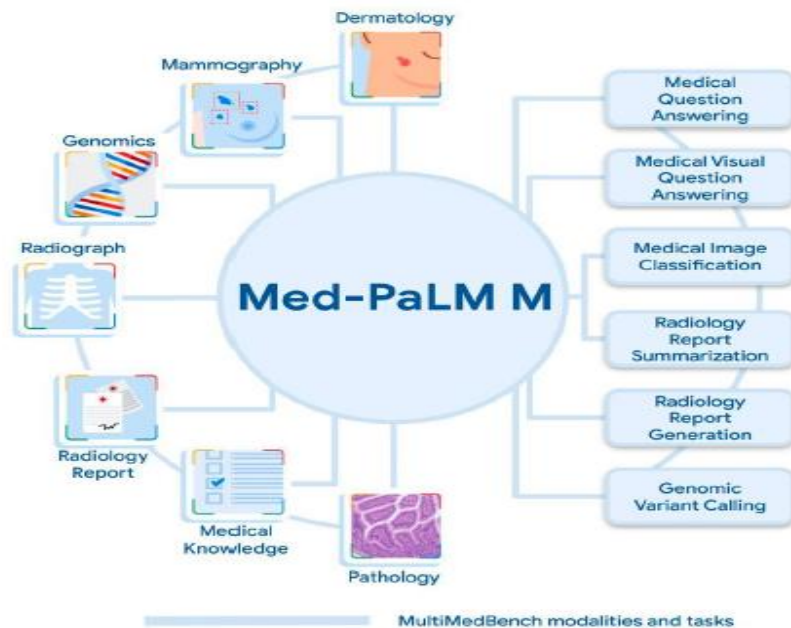


high five

Robotics, Simulations, Physical Tasks



Playing Games



Biology + Healthcare



From Attention Mechanism to Transformer

- If we have attention, do we even need recurrent connections?
- Can we transform our RNN into a purely attention-based model?

Attention Is All You Need

Neurips 2017

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

Citation: 173883+

Sequence-to-sequence Model

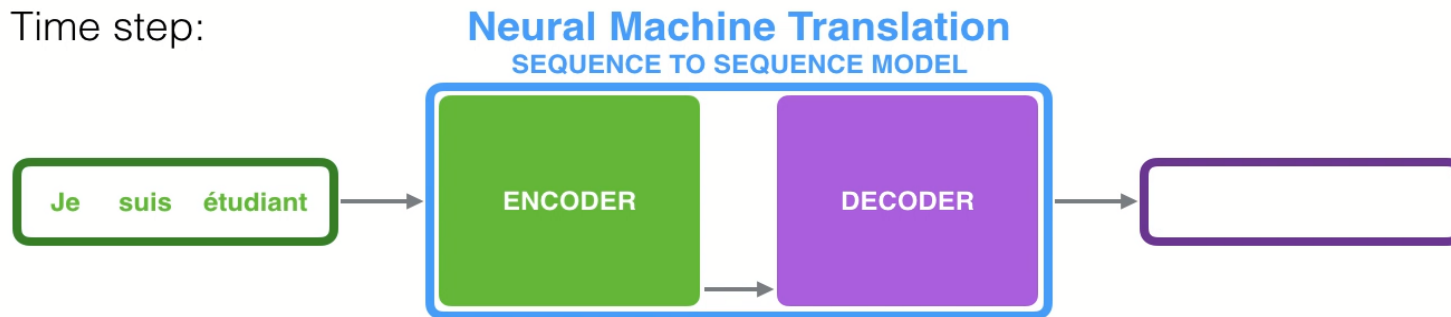
Sequence-to-Sequence Overview

- A general encoder-decoder architecture.
- **Encoder:** maps input to a neural representation.
- **Decoder:** generates output from the neural representation.
- When both input and output are sequences: seq2seq model.

Seq2Seq Applications: **Sequence-to-sequence is versatile!**

- ❖ **Machine Translation:** French ---→ English
- ❖ **Summarization:** long text ----→ concise summary
- ❖ **Dialogue:** previous utterances ---→ next response
- ❖ **Parsing:** input text---→ parse tree (as a sequence)
- ❖ **Code Generation:** natural language --→ Python code

Time step:



In the visualization, each pulse from the encoder or decoder represents an RNN processing its input and updating its hidden state based on current and past inputs.

<https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/>

Neural Machine Translation

Sequence-to-Sequence as a Conditional Language Model

Like any language model, the decoder predicts the next word in the target sequence Y.

It is conditional because predictions are made based on both:

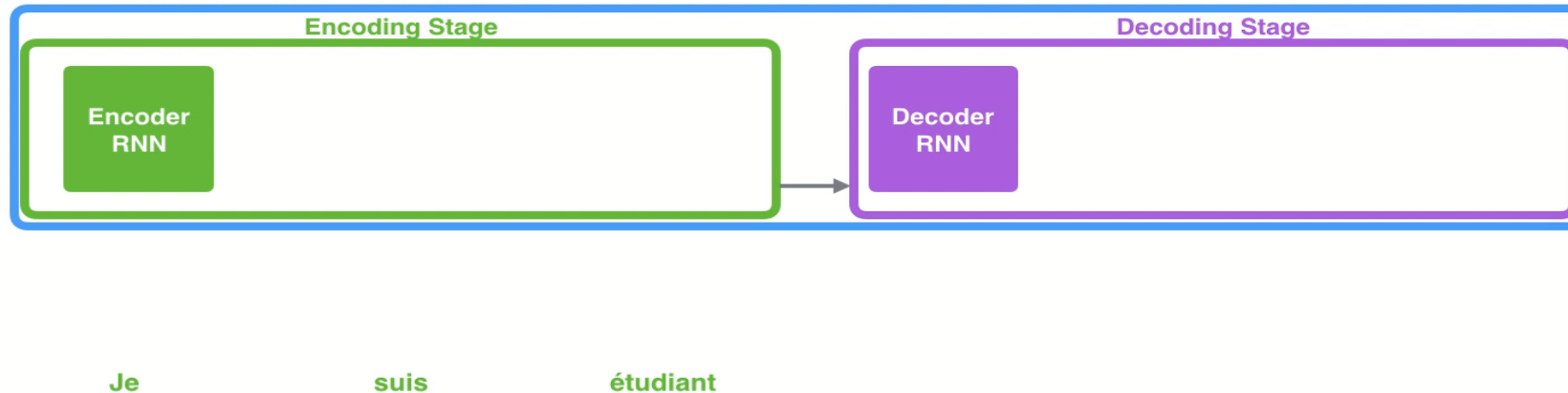
- ❖ Previously generated target tokens.
- ❖ The full source sequence X (via the encoder).

$$P(y|x) = P(y_1|x) P(y_2|y_1, x) P(y_3|y_1, y_2, x) \dots P(y_T|y_1, \dots, y_{T-1}, x)$$

Probability of next target word, given target words so far and source sentence x

This allows the model to generate context-aware translations or responses.

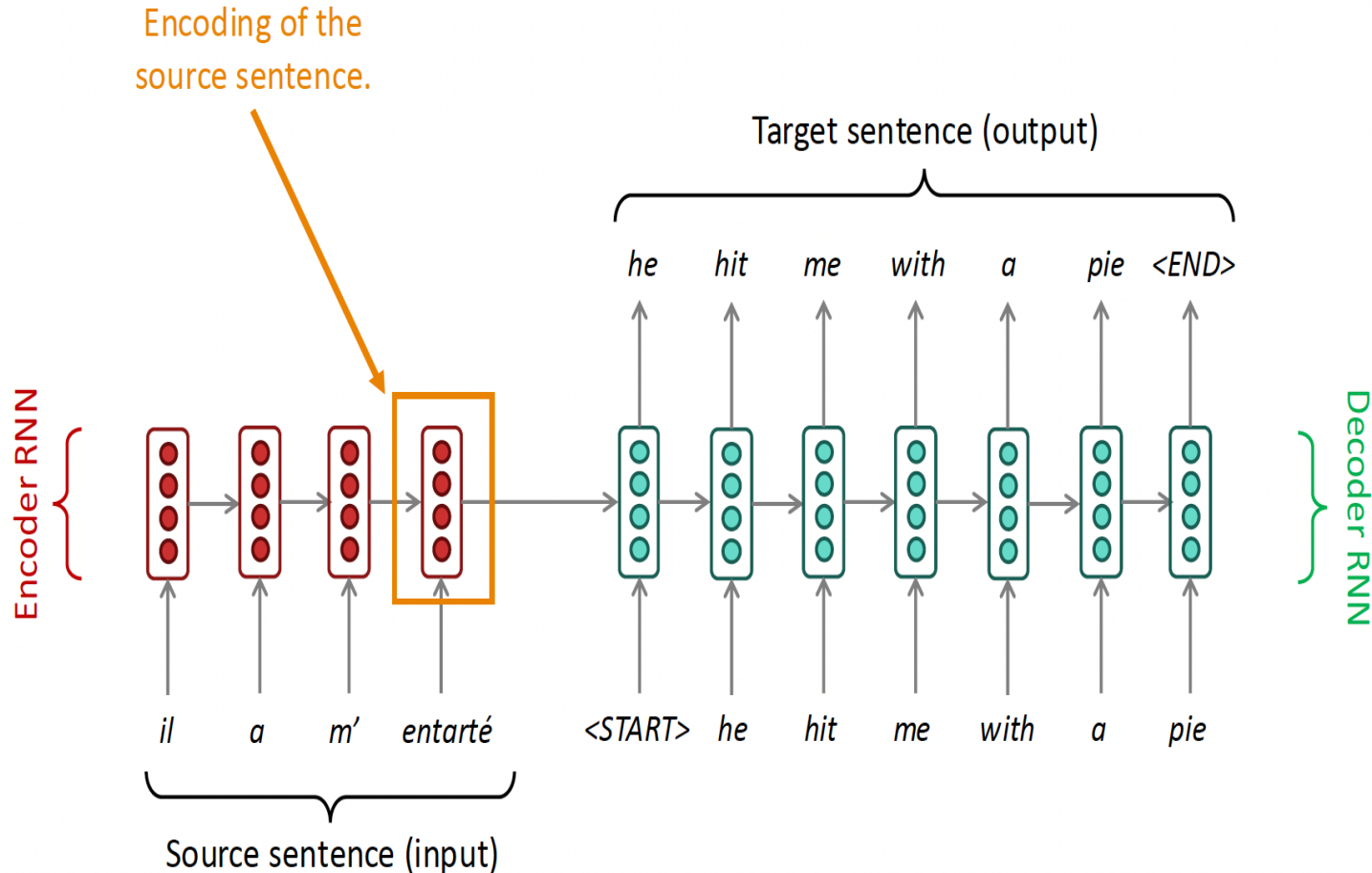
Neural Machine Translation SEQUENCE TO SEQUENCE MODEL



The **context** vector turned out to be a bottleneck for these types of models. It made it challenging for the models to deal with long sentences.

<https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/>

Limitations of RNNs for Long Sequences



Long-distance dependencies are hard to learn:

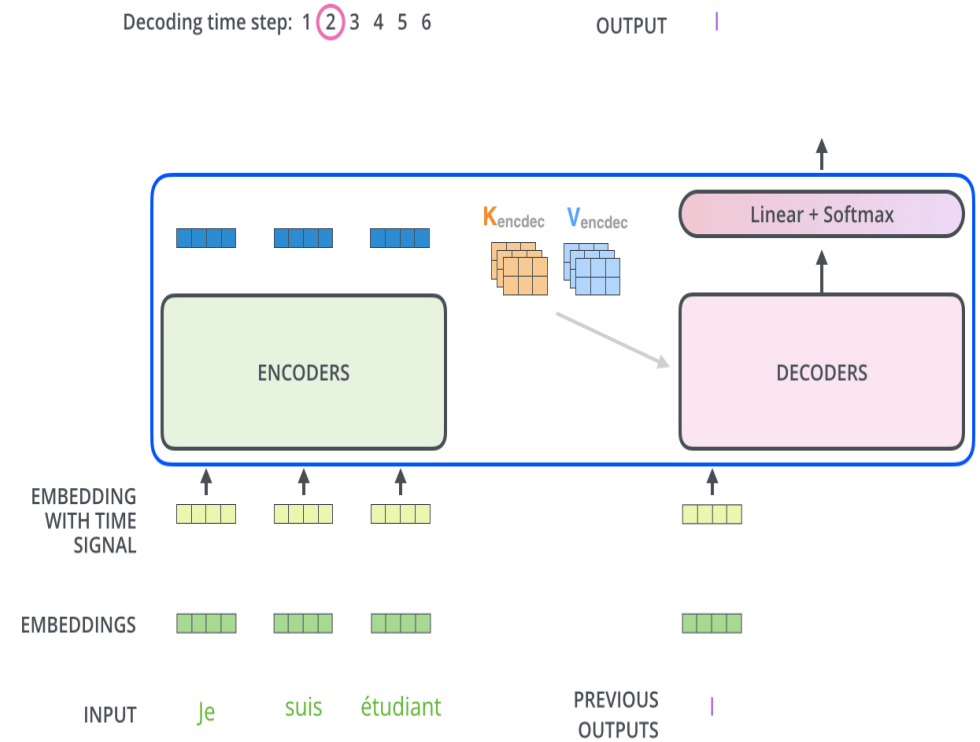
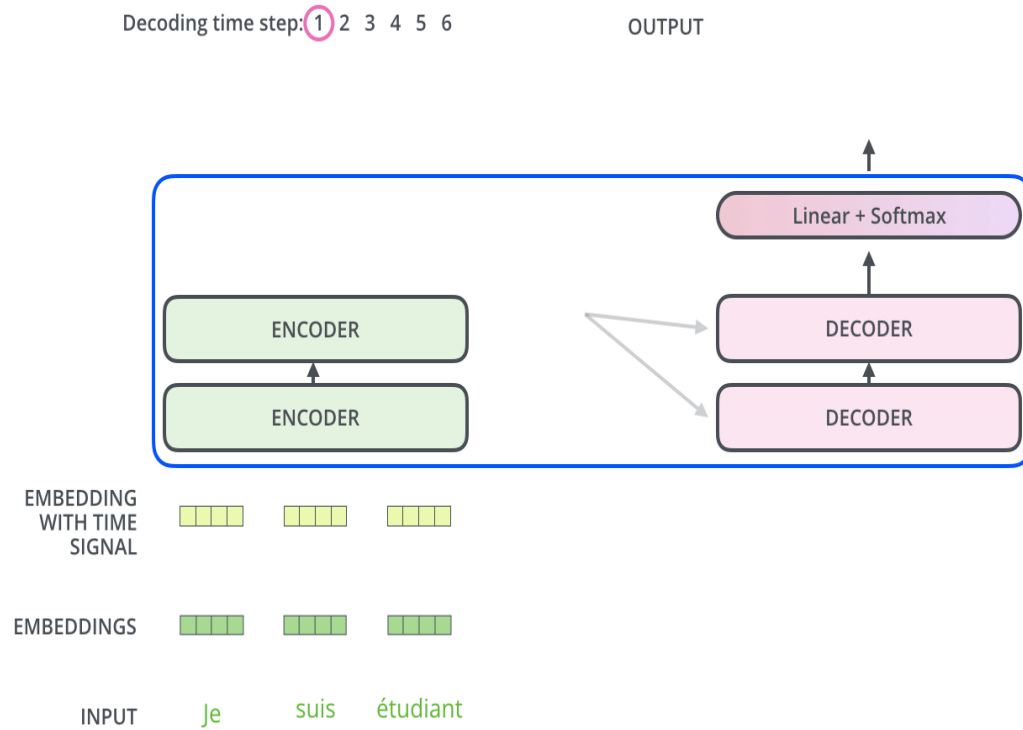
- ❖ Distant word pairs require O sequence length steps to interact.
- ❖ Leads to vanishing or exploding gradients.

RNNs are inherently sequential:

- Forward/backward passes require $O(\text{sequence length})$ dependent operations.
- Each hidden state of RNN must be computed in order.

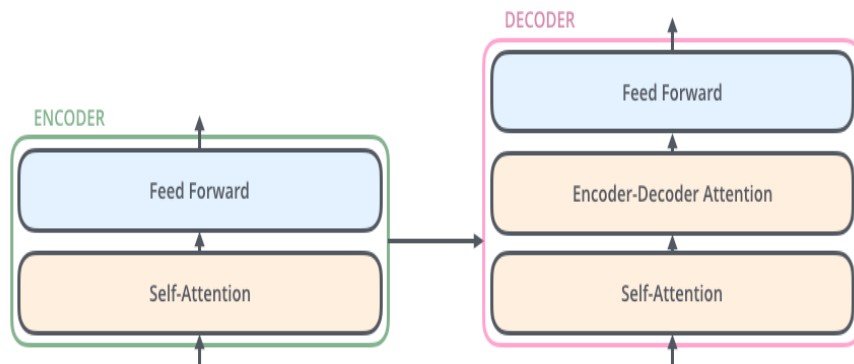
These challenges motivated the development of **attention-based and transformer** models.

Combine Encoder and Decoder Values: Animation

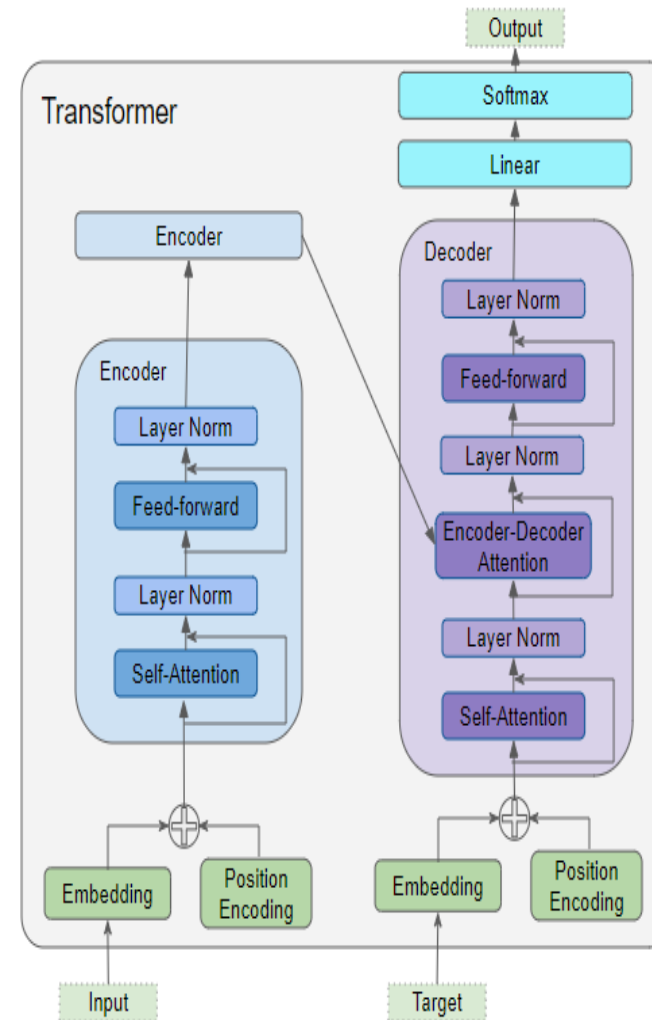


Transformer: A High-Level Look

- **Transformer** is composed of an **encoder** and a **decoder**.
- The **input and output sequence embeddings** are added with **positional encoding** before being fed into the encoder and the decoder.
- The **encoder** is a stack of multiple Transformer layers, used to **transform the text embeddings into a representation** that can support a variety of tasks.
- The **decoder** is also a stack of multiple Transformer layers, used to **predict the next token to continue the input text**. It also inserts a sub-layer, known as the **encoder-decoder attention**.



<https://jalammar.github.io/illustrated-transformer/>



Because the decoder needs two kinds of information:

- **masked self-attention**: what have I generated so far?

- **encoder-decoder attention**: what part of the input should I focus on now?

That is why the decoder is more complicated than the encoder.

<https://towardsdatascience.com/transformers-explained-visually-part-2-how-it-works-step-by-step-b49fa4a64f34/>

Transformer Layer (Block)

► **Input:** $D \times N$ matrix of word embeddings, where D is the embedding dimension and N the sequence length.

► **Multi-Head Attention:**

- ❖ Each token can attend to every other token.
- ❖ Output dimension is $D \times N$.
- ❖ Residual connection: add the original inputs back.

► **LayerNorm:**

- Applied to each column (token) independently.
- Normalizes across the embedding dimension.

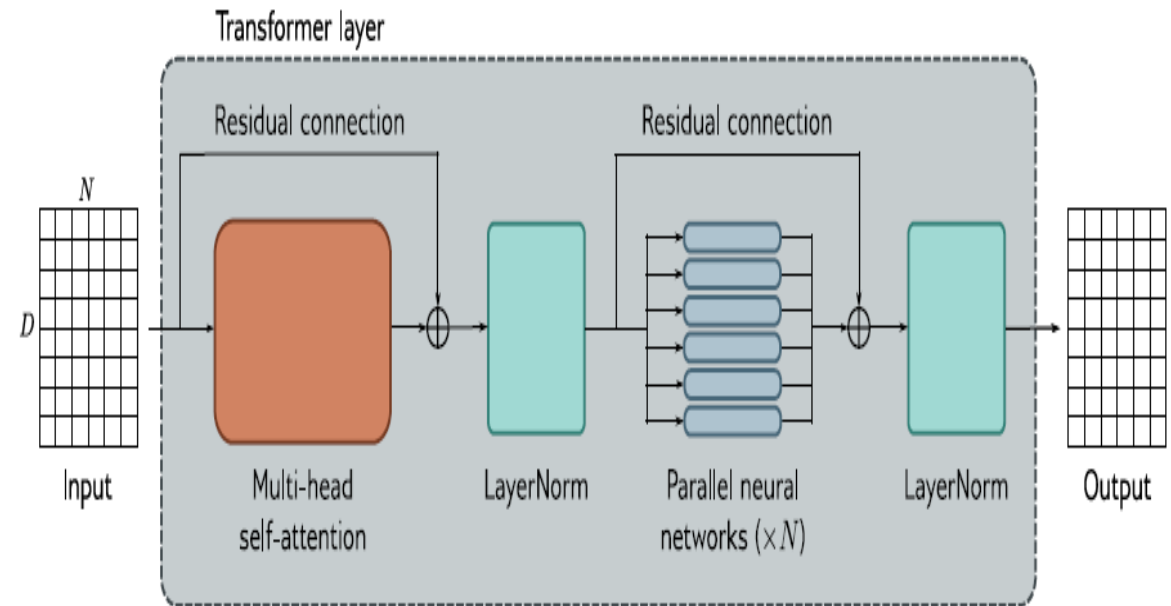
► **Fully Connected Feed-Forward:**

- ✓ Same MLP applied to each column.
- ✓ Residual connection again.

► **Final LayerNorm:**

- Normalizes outputs across D for each token.

► **Result:** Output is a $D \times N$ matrix with updated token representations.



$$\mathbf{X} \leftarrow \mathbf{X} + \text{MhSa}[\mathbf{X}]$$

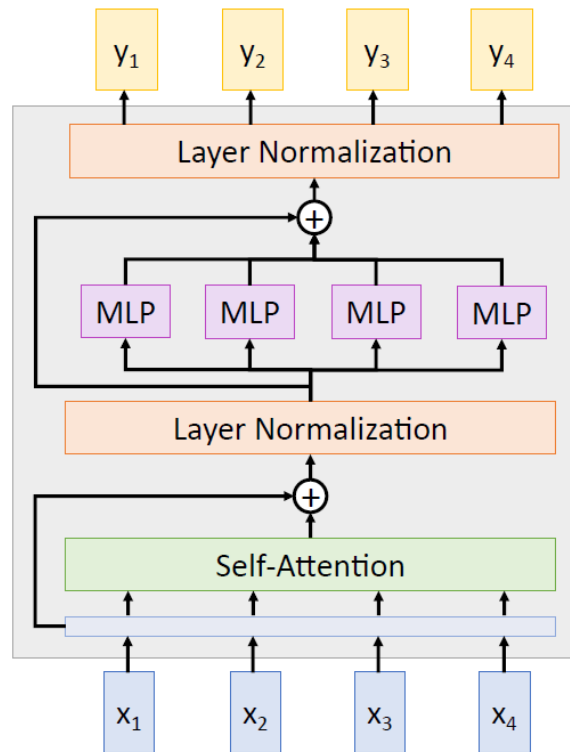
$$\mathbf{X} \leftarrow \text{LayerNorm}[\mathbf{X}]$$

$$\mathbf{x}_n \leftarrow \mathbf{x}_n + \text{mlp}[\mathbf{x}_n] \quad \forall n \in \{1, \dots, N\}$$

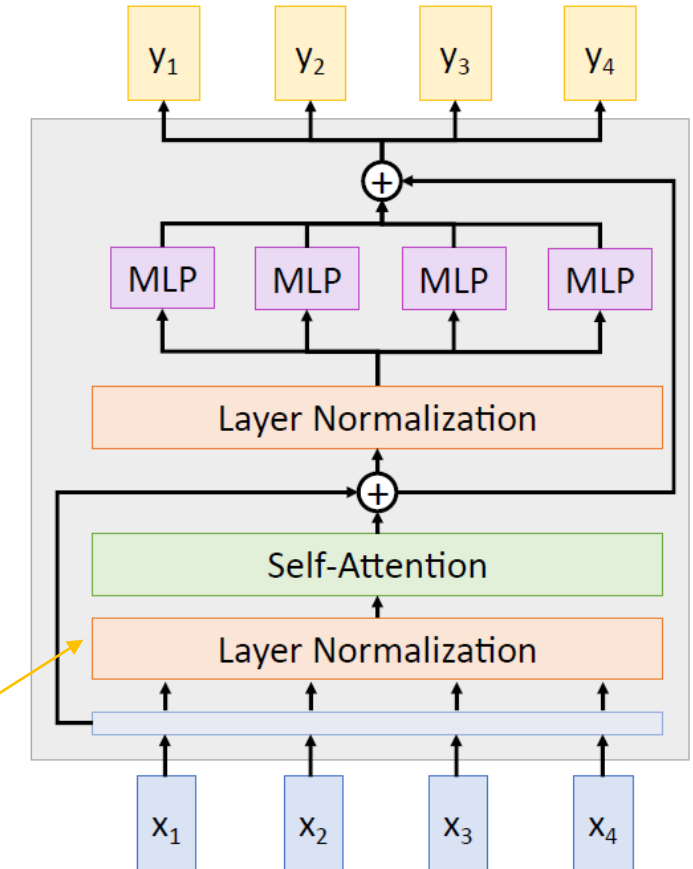
$$\mathbf{X} \leftarrow \text{LayerNorm}[\mathbf{X}],$$

Putting it All Together

- Self-attention is the only interaction between vectors.
- Layer normalization and MLP work independently per vector.
- The structure is highly scalable and highly parallelizable.



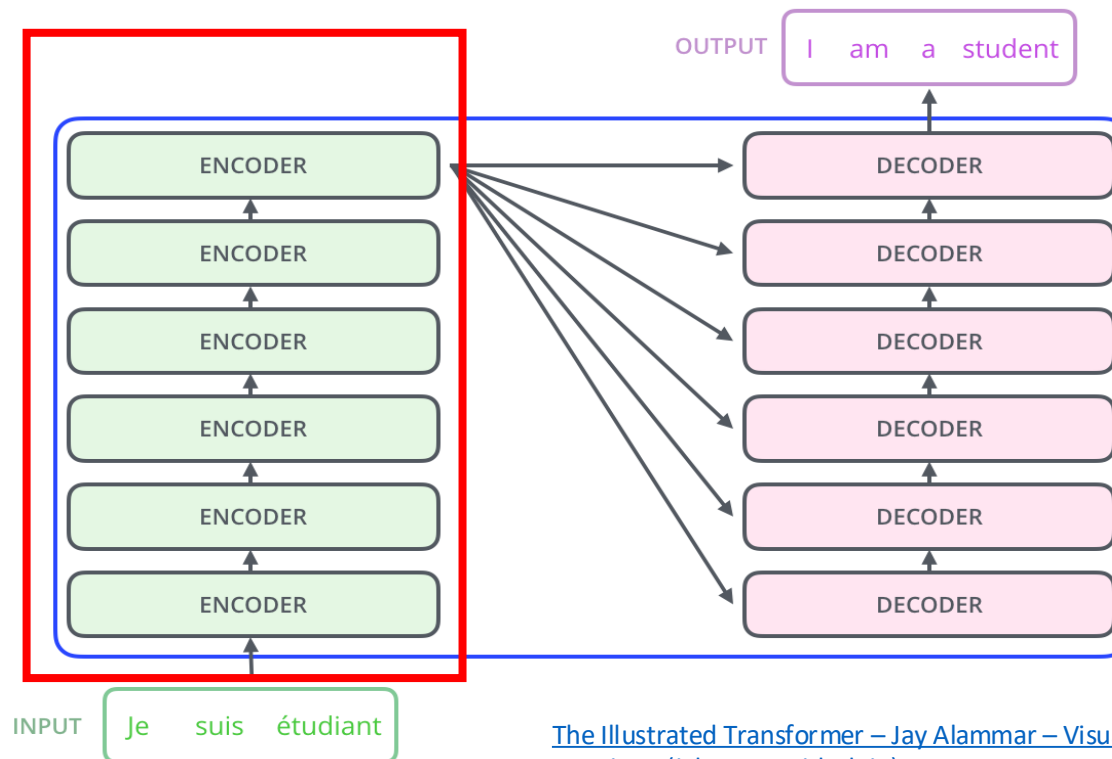
(Johnson, 2022)



In practice, we often put the layer normalization inside the residual attention, which tend to give more stable training and is commonly used in practice.

Encoder

- The Transformer encoder consists of multiple identical Transformer layers that process the input sequence in parallel. Each layer refines the input representation by capturing dependencies across all positions. ($N = 6$ in the paper *Attention is all you need*).
- The encoder outputs a contextualized representation for each token, which serves as input to the decoder.



- ▶ **Input:** $D \times N_{\text{enc}}$ matrix of embeddings (or projected tokens).
- ▶ **Self-Attention Layer:**
 - ▶ Each token attends to every other token in the source.
 - ▶ Multi-head mechanism for capturing diverse relationships.
 - ▶ Residual connection + LayerNorm keep gradients stable.
- ▶ **Feed-Forward Layer:**
 - ▶ Position-wise MLP applied to each token's embedding.
 - ▶ Another residual connection + LayerNorm.
- ▶ **Stacking Layers:**
 - ▶ Typically L identical encoder layers.
 - ▶ Output is $D \times N_{\text{enc}}$, providing contextualized embeddings for each source token.

[The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. \(jalammar.github.io\)](https://alammar.github.io/illustrated-transformer/)

Transformer Decoder

► Decoder Sublayers:

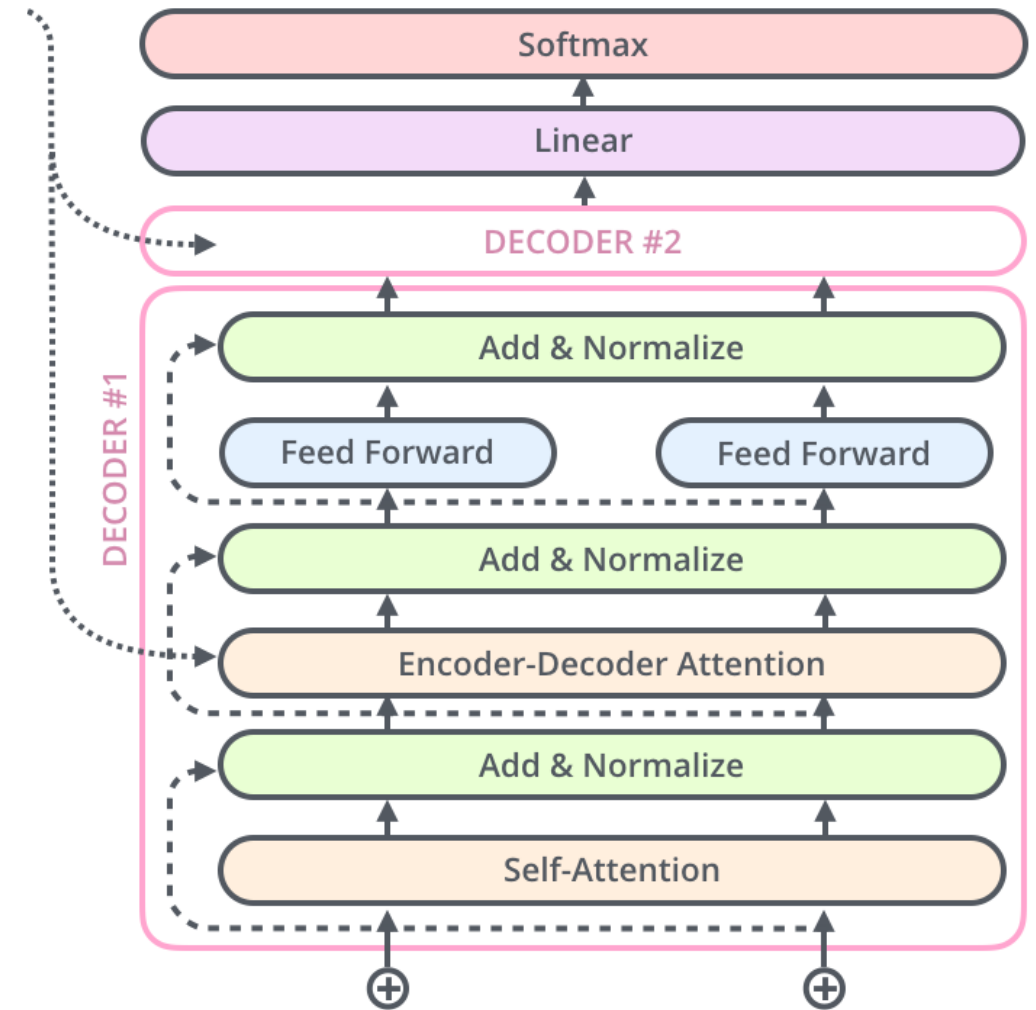
1. **Masked Self-Attention:** Targets attending to themselves (past tokens).
2. **Cross-Attention:** Queries from the decoder, Keys and Values from encoder.
3. **Feed-Forward Network:** Applies position-wise transformations to each token.

► Residuals + LayerNorm:

- Each sublayer uses skip connections and normalization.
- Ensures stable training and consistent dimensionality.

► Outcome:

- Decoder hidden states are enriched with relevant source info.
- Final step: linear layer + softmax for next-token prediction.

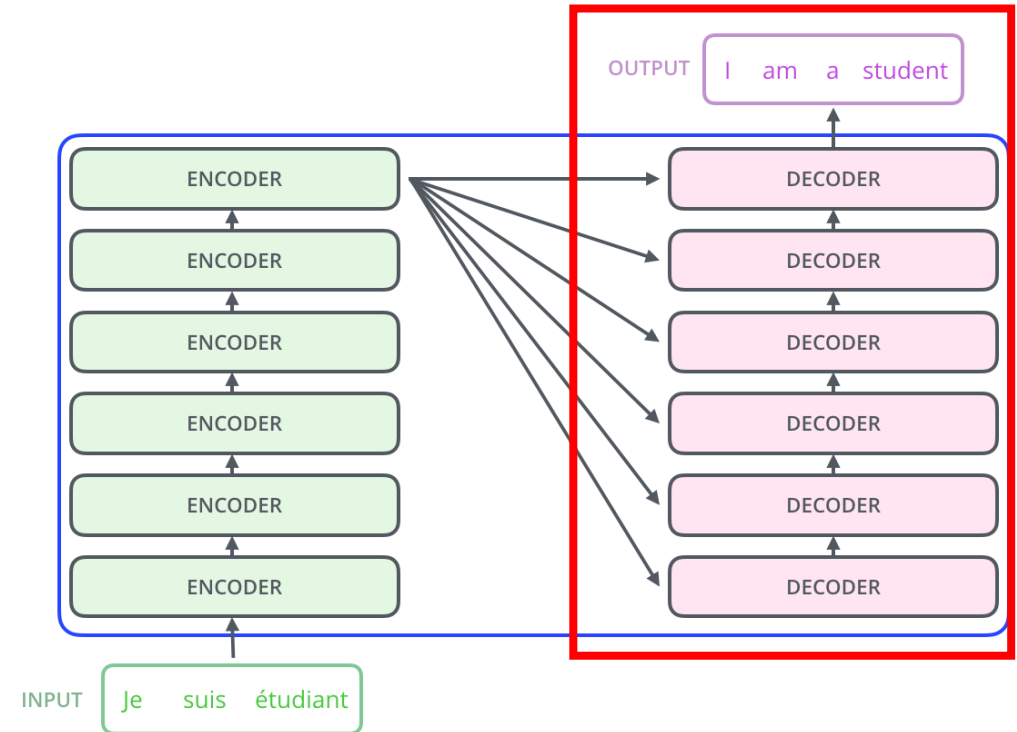


Thinking

Machines

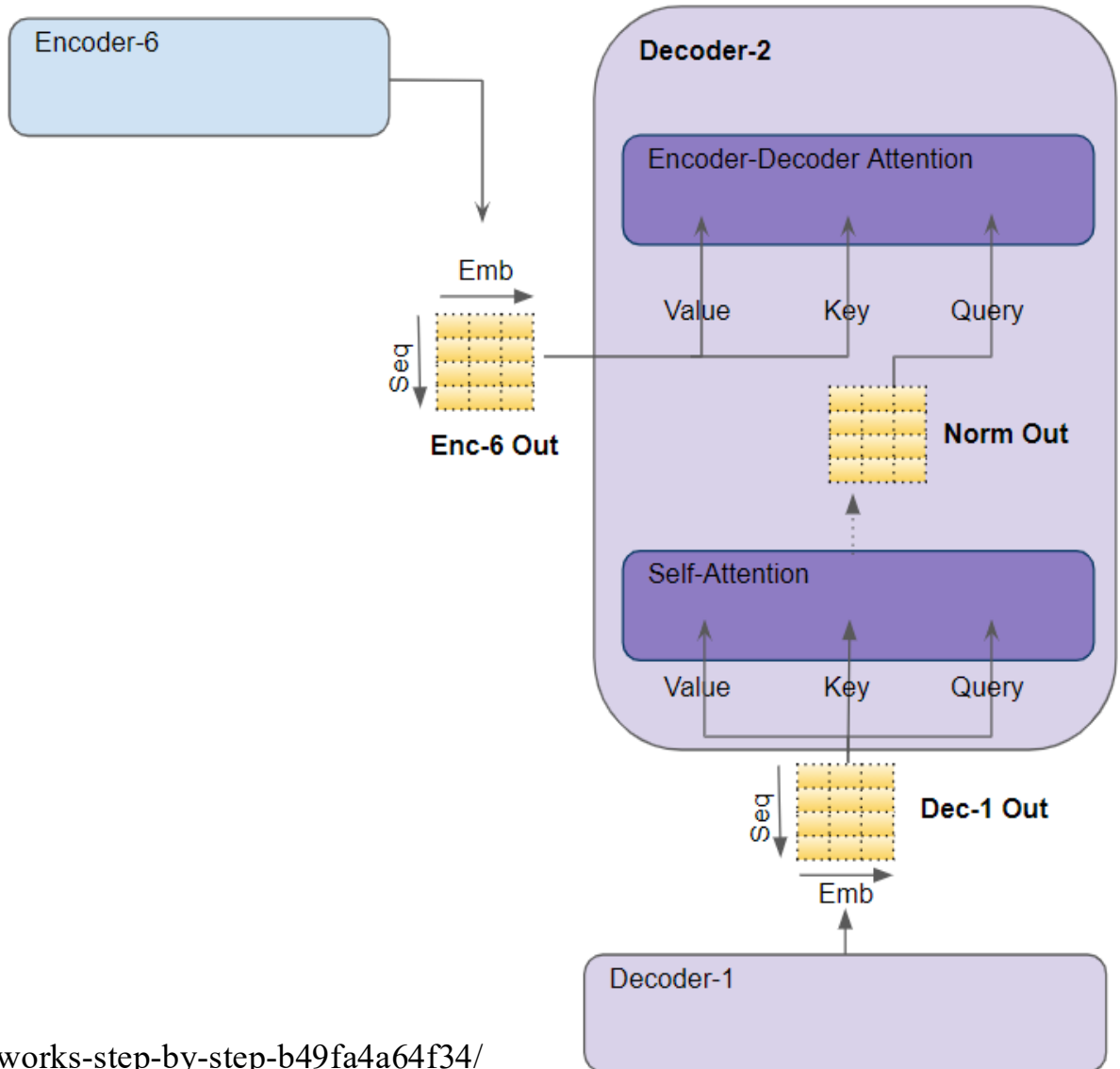
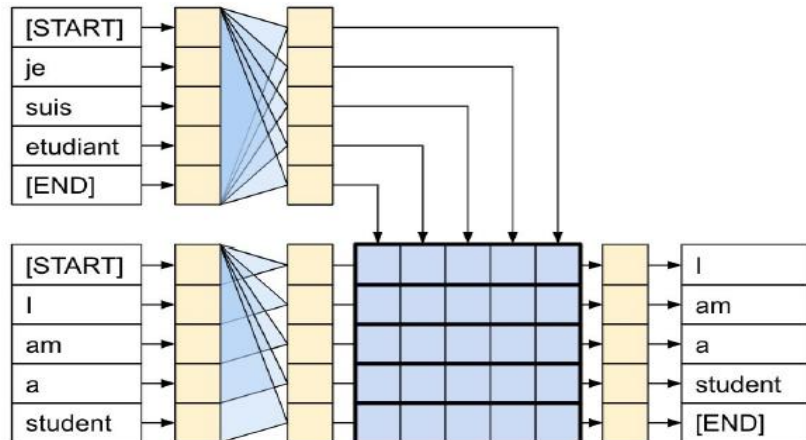
Decoder: Masked Self-Attention

- The Transformer decoder is also composed of multiple layers and generates the output sequence step by step. In the **decoder self-attention**, queries, keys and values are all from the outputs of the previous decoder layer.
- However, each position in the decoder is allowed only to attend to all positions in the decoder up to that position. This **masked attention** preserves the autoregressive property, ensuring that the prediction only depends on those output tokens that have been generated.



Decoder: Cross-Attention

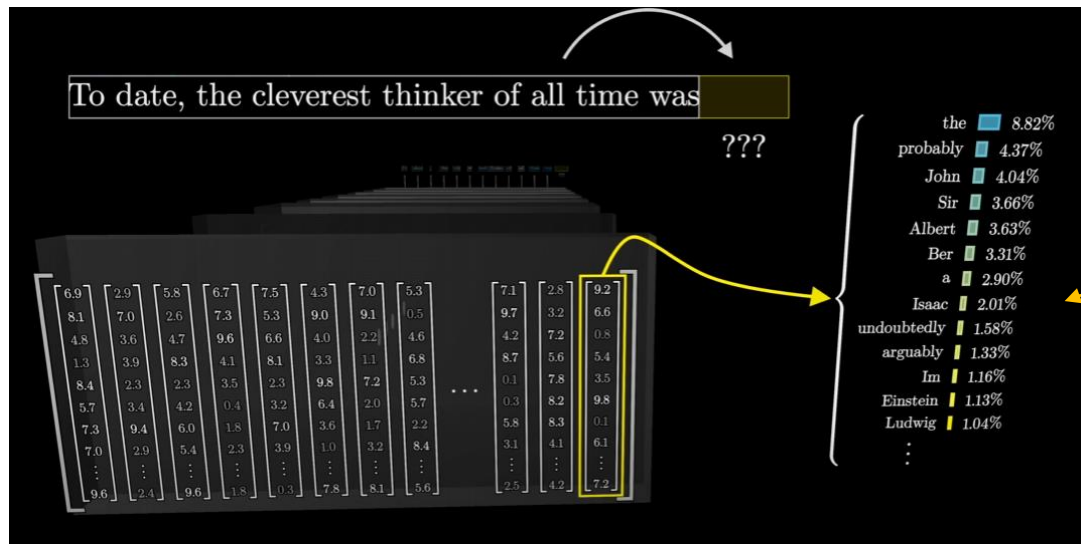
- Cross-attention in the decoder allows it to incorporate information from the encoder's output.
- **Queries are from the outputs of the decoder's self-attention sublayer (decoder's hidden states).**
- **The keys and values are from the Transformer encoder outputs.**
- In reality, cross-attention is also multi-headed.
- Such design enables the model to align generated tokens with relevant input features dynamically.



<https://towardsdatascience.com/transformers-explained-visually-part-2-how-it-works-step-by-step-b49fa4a64f34/>

Decoder: Final Layer

- The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.
- The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0).
- The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.



Which word in our vocabulary
is associated with this index?

Get the index of the cell
with the highest value
(argmax)

log_probs



am

5

Softmax

logits



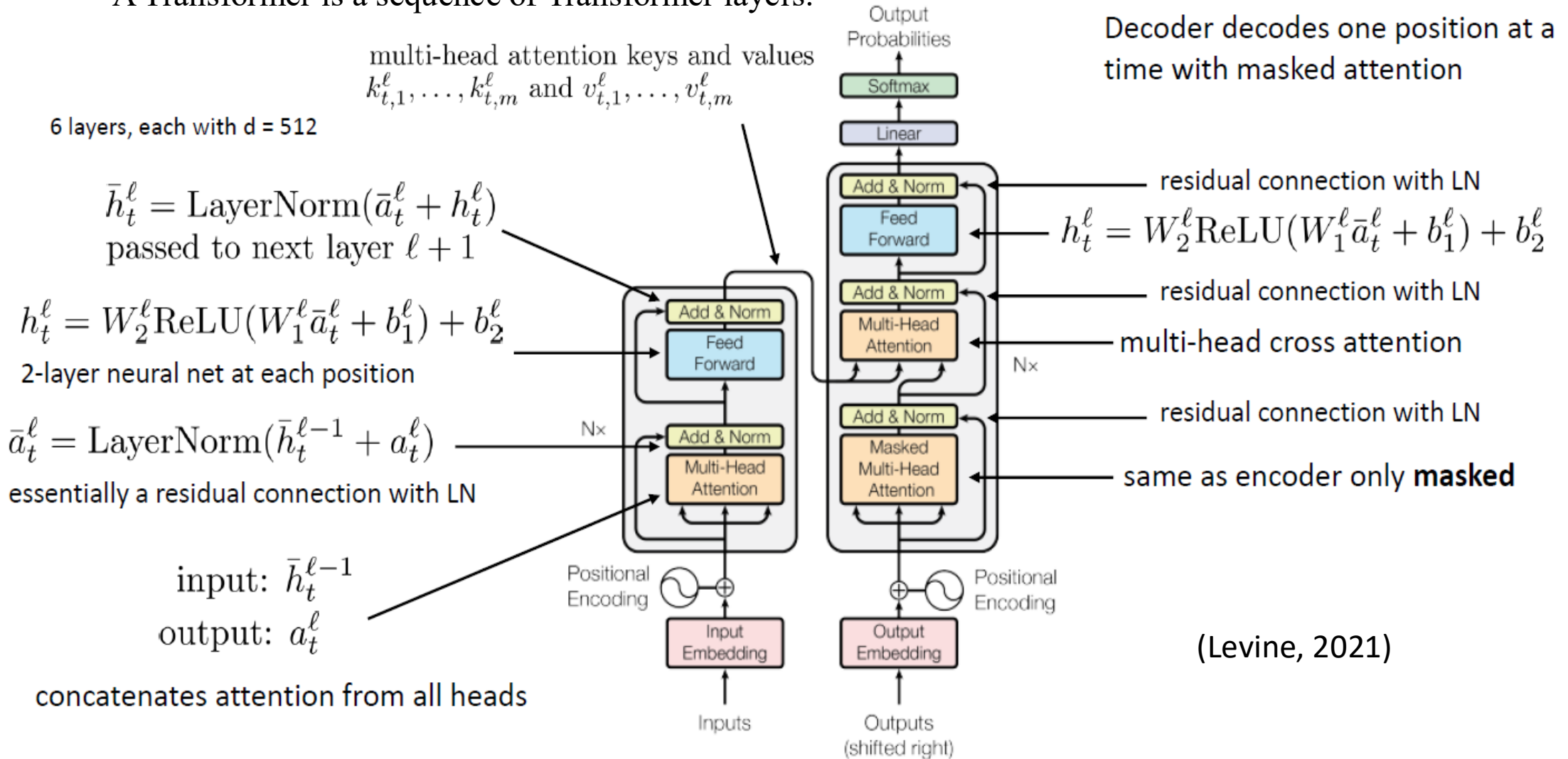
Linear

Decoder stack output



Transformer: Putting it All Together

A Transformer is a sequence of Transformer layers.



(Levine, 2021)

Attention Mechanisms: Overview

Given:

- ▶ n query vectors $q_1, \dots, q_n \in \mathbb{R}^d$
- ▶ m key vectors $k_1, \dots, k_m \in \mathbb{R}^d$
- ▶ m value vectors $v_1, \dots, v_m \in \mathbb{R}^p$

The attention mechanism outputs n vectors $o_1, \dots, o_n \in \mathbb{R}^q$ to

$$o_j = \frac{1}{C} \sum_{i=1}^m f(q_j, k_i) g(v_i)$$

where $f(q_j, k_i)$ measures similarity and $g(v_i)$ is a linear transformation.

The **query** asks what is relevant
the **keys** decide what matches
the **values** provide the information
that gets passed along.

- ▶ $f(q_j, k_i)$ is often the **embedded Gaussian similarity**:

$$f(q_j, k_i) = \exp \left(\theta(q_j)^\top \phi(k_i) \right)$$

- ▶ $\theta(q_j) = W_q q_j, \phi(k_i) = W_k k_i$
- ▶ $g(v_i) = W_v v_i$
- ▶ $C = \sum_{i=1}^m f(q_j, k_i)$ normalizes the scores

This setup lets each output vector depend on all input vectors.

- **query**: “emails about travel reimbursement”
- **keys**: subject lines of emails
- **values**: the email bodies

Self-Attention and Learnable Query

Self-Attention:

- ▶ $Q = K = V = X \in \mathbb{R}^{d \times n}$
- ▶ Computes internal correlations:

$$O = W_v X \cdot \text{softmax}((W_k X)^T (W_q X))$$

Learnable Query:

- ▶ $K = V = X$, Q is trainable and independent of X
- ▶ Output size is fixed, independent of input size:

$$O = W_v X \cdot \text{softmax}((W_k X)^T Q)$$

Aspect

Self-Attention

Learnable Query

Self-attention is good for

Query Definition

$Q=K=V=X$ (input data itself)

Q is independent and learnable

- ❖ contextualizing all tokens
- ❖ modeling relationships within a sequence
- ❖ building rich token-level representations

Input Dependency

Input size determines query size

Query size is fixed and independent

Flexibility

Suitable for variable-length inputs

Suitable when fixed-length output is needed

Learnable queries are good for

- extracting summaries
- object detection
- retrieving task-specific information
- producing a fixed number of outputs or slots

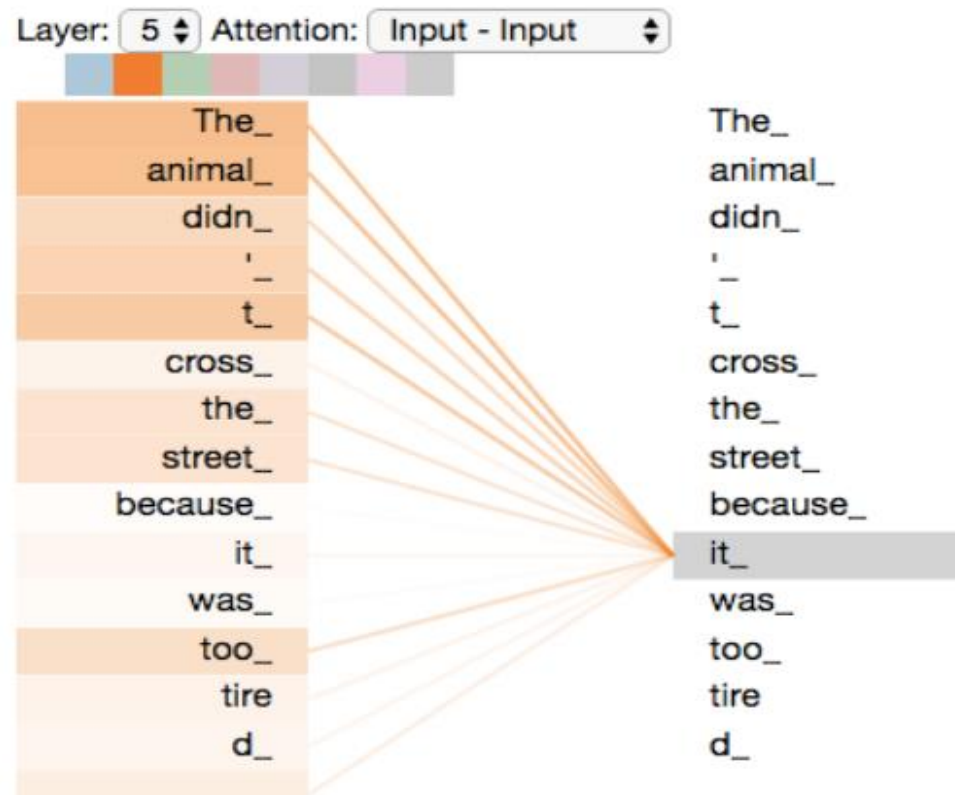
Use Case

Typically used for internal relationships in input data (like word dependencies)

Often used when fixed-size representation is needed, regardless of input size

Self-Attention: Motivation

- Suppose the following sentence is an input sentence we want to translate: *The animal didn't cross the street because it was too tired*. What does “it” in the sentence refer to? Is it referring to the street or the animal? It's a simple question to a human, but not as simple to an algorithm.
- The self-attention mechanism enables the model to associate “it” with “animal”.



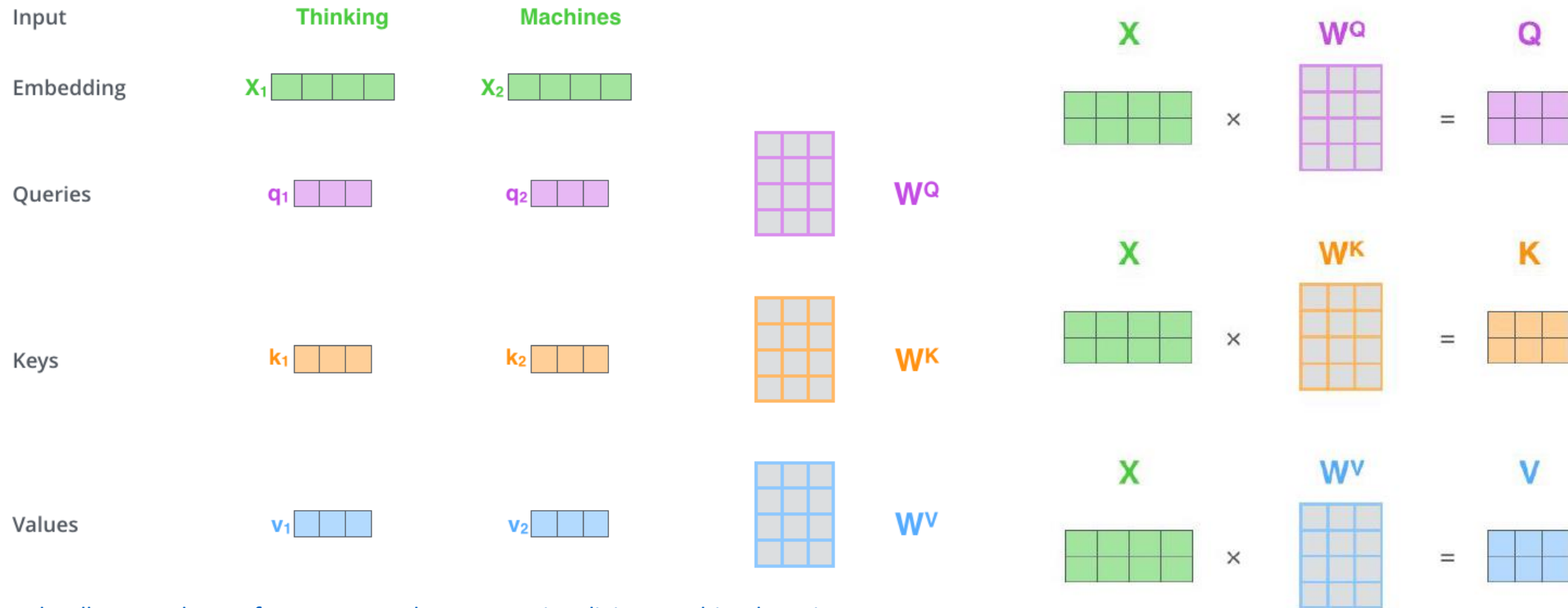
As we are encoding the word “it” in encoder #5, which is the top encoder in the stack, part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of “it”.

- ❖ The **right column** shows the token whose attention you are examining: here it is “it”.
- ❖ The **left column** shows the tokens it can attend to.
- ❖ The **orange lines** indicate attention weights from “it” to the other tokens.
- ❖ **Darker/thicker lines** mean stronger attention.

[The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. \(jalamar.github.io\)](https://jalamar.github.io)

Self-Attention: Details

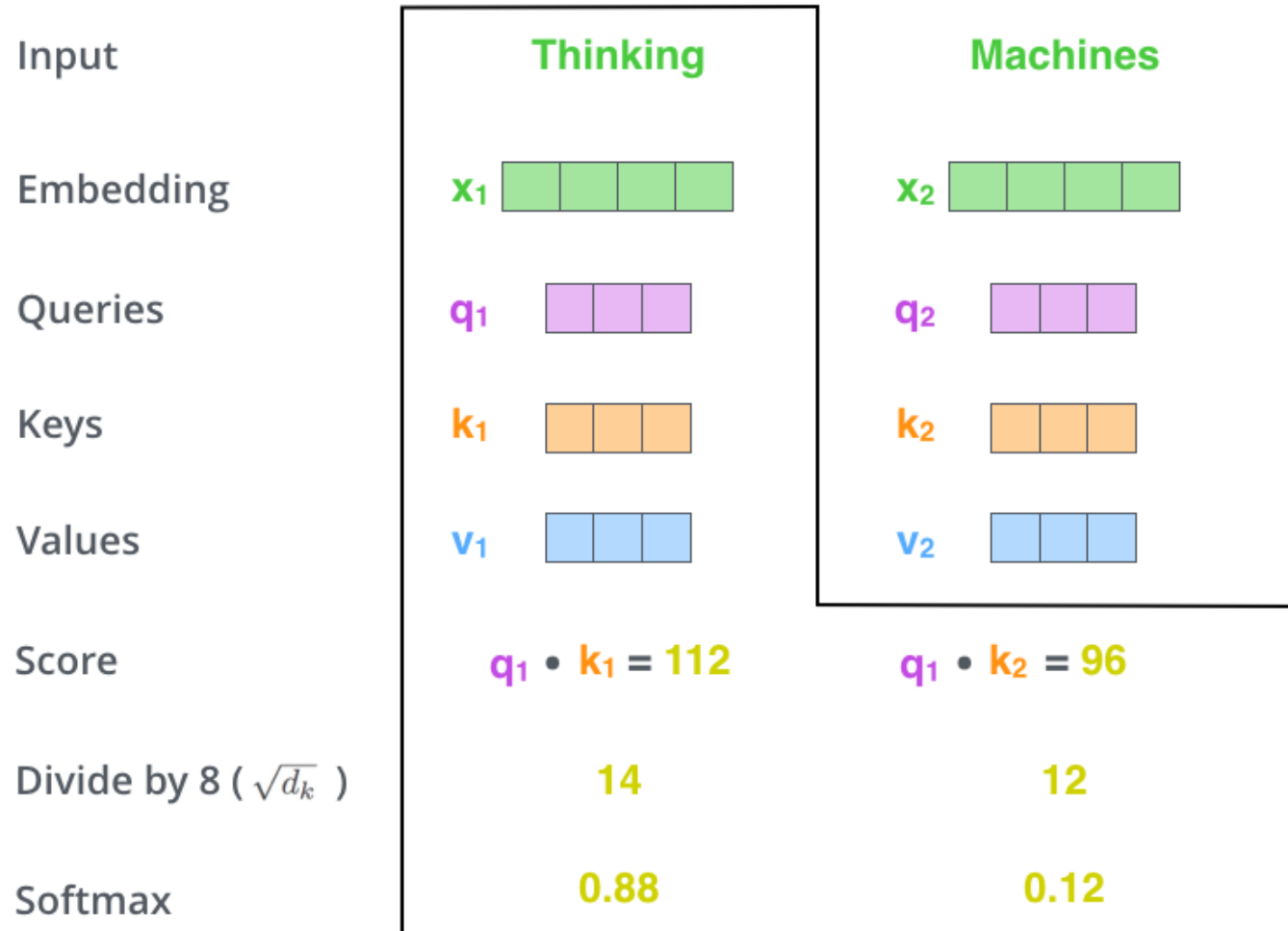
- The first step is to create three vectors from each of encoder's input vectors.** These vectors are created by multiplying the embedding by three matrices that we trained during the training process (usually smaller in dimension than the embedding vector). Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512.



[The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. \(jalammar.github.io\)](https://jalammar.github.io)

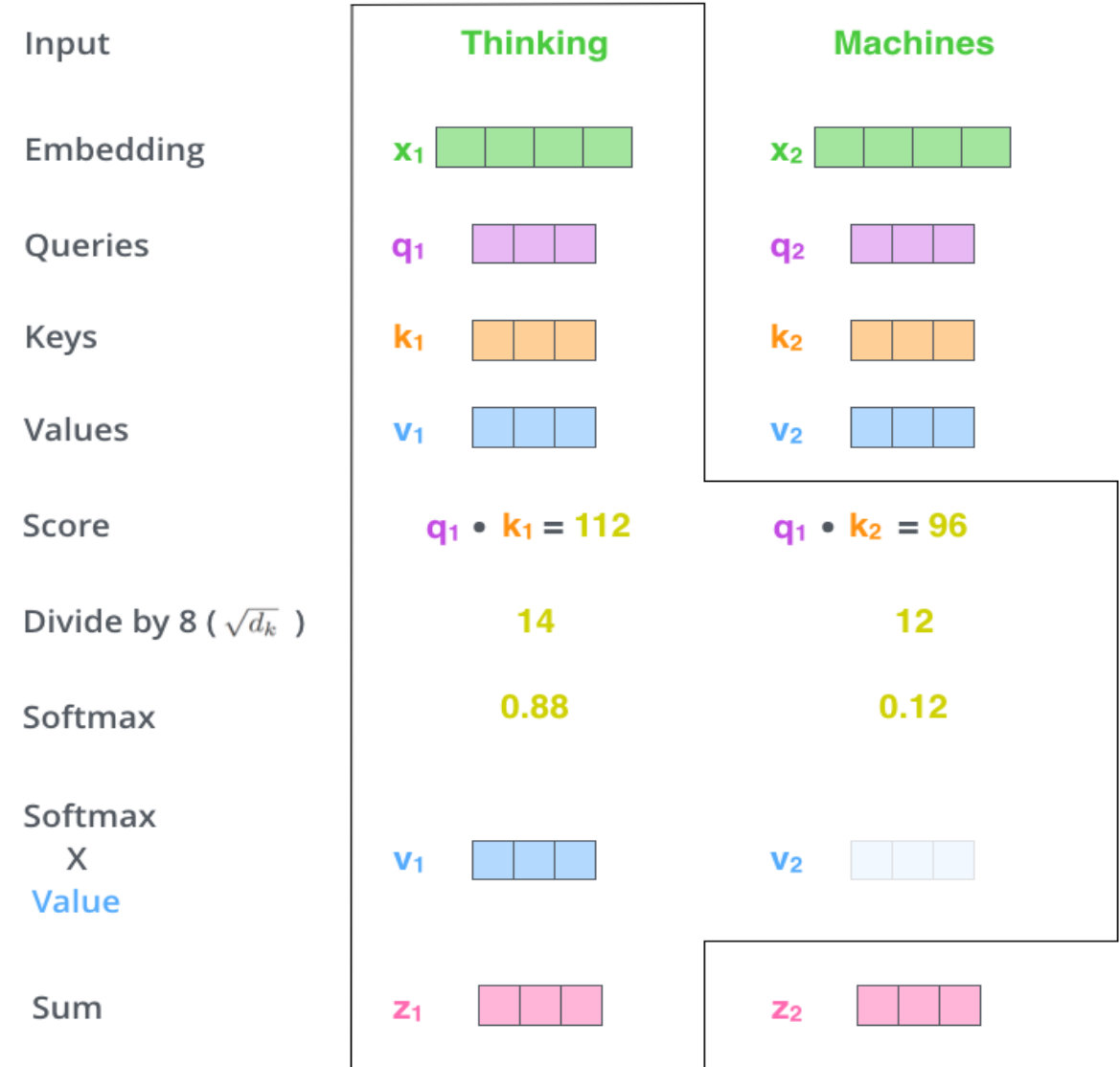
Self-Attention: Details

- The second step is to calculate the scoring function and then divide it by the square root of the dimension of the key vectors.



Self-Attention: Details

- The third step is to multiply each value vector by the softmax score and sum up the weighted value vectors.
- The resulting vector is the one we can send along to the feed-forward neural network.
- In the actual implementation, such calculation is done in matrix form for faster processing.



Self-Attention: Matrix Calculation

- Query, key and value matrices are calculated through matrix multiplication.

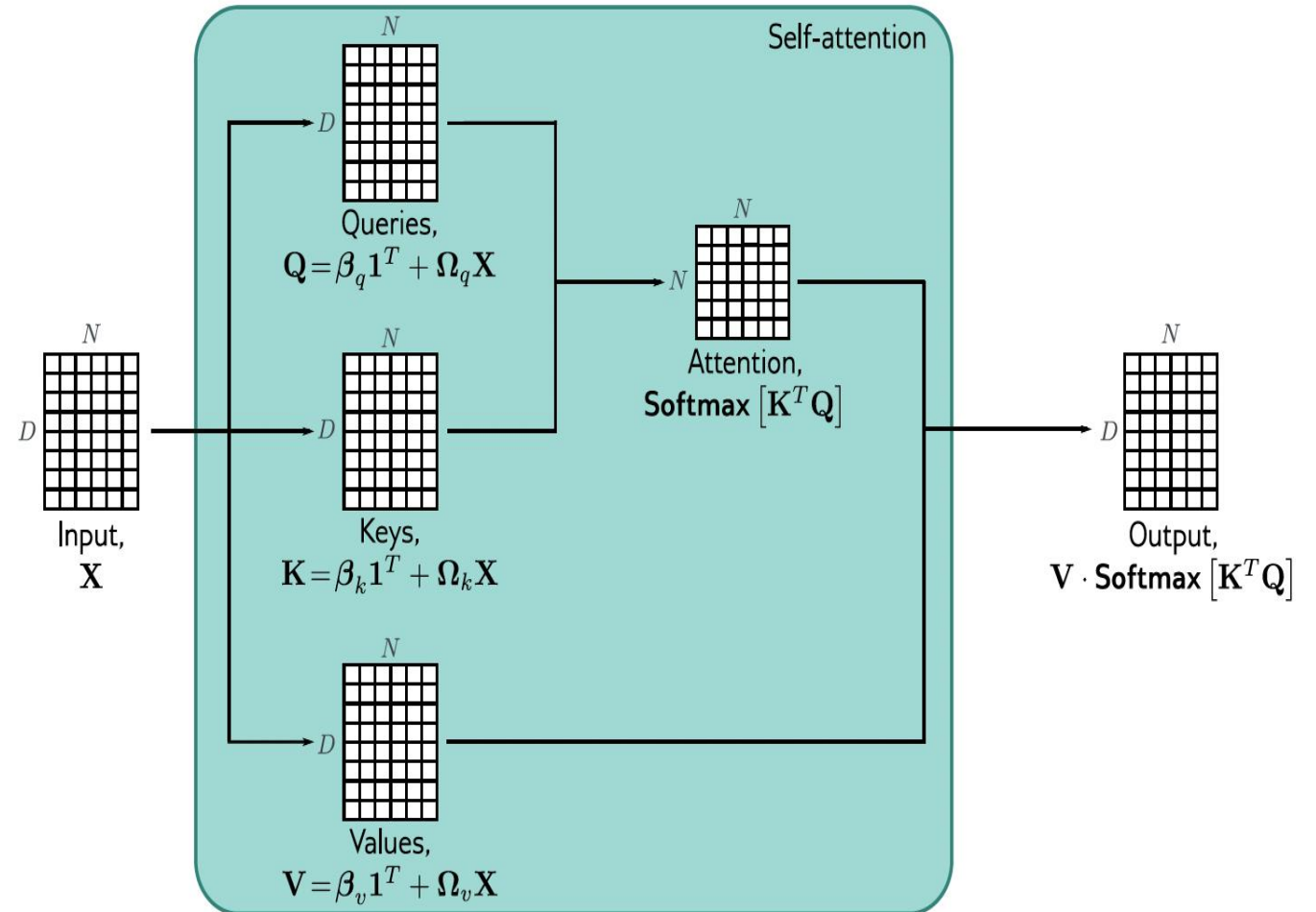
$$Q = W_Q X = \beta_q 1^T + \Omega_q X$$

$$K = W_K X = \beta_k 1^T + \Omega_k X$$

$$V = W_V X = \beta_v 1^T + \Omega_v X$$

$$Y = \text{Softmax}(QK^T)V$$

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) = Z$$



Scaled Dot-Product Attention

- Scaled Dot-Product Attention is a key component of the Transformer architecture, widely used in modern deep learning models such as BERT and GPT. The formula is as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- Scaled Dot-Product Attention possess the following key advantages:
 - **Efficiency:** It relies on simple matrix multiplications and scaling, making it highly efficient and suitable for parallel computation on GPUs/TPUs.
 - **Scalability:** It performs well on large-scale datasets and is the foundation of many state-of-the-art models.
 - **Flexibility:** It can handle sequences of different lengths, as the attention mechanism dynamically computes the relationships between queries and keys.

Why scale by $\sqrt{d_k}$?

- ▶ Elements of $q_i^\top k_j$ have variance d_k if q_i, k_j are independent with unit variance.
- ▶ Without scaling, large dot products push softmax into regions with small gradients.
- ▶ Scaling by $\sqrt{d_k}$ normalizes the variance:

$$\text{Var}(q_i^\top k_j) = d_k \Rightarrow \frac{q_i^\top k_j}{\sqrt{d_k}} \text{ has unit variance}$$

This stabilizes gradients during training and improves convergence.

From Self-Attention to Transformers

- The basic concept of self-attention can be used to develop a very powerful type of sequence model, called **a Transformer**.
- But to make this actually work, we need to develop a few additional components to address some functional limitations.
 - **Masked encoding:** How to prevent attention lookups into the future? Ensures causality in autoregressive models.
 - **Multi-head Attention:** Allows querying multiple positions at each layer. Increases model capacity to capture complex patterns.
 - **Positional encoding:** Address lack of sequence information, especially for images and videos. Provides sequence order awareness, crucial for text, images, and videos.

The combination of these techniques makes Transformers robust and efficient for sequence modeling.

Masked Attention

- **Problem:** During training, self-attention might access future tokens, leading to data leakage.
- **Solution:** Use a mask to block attention from future positions.
- **Implementation:** Add a mask matrix M:

$$Scores = \frac{QK^T}{\sqrt{d_k}}, \quad Mask = \begin{cases} 0, & \text{past} \\ -\infty, & \text{future} \end{cases}$$
$$Masked\ Scores = Scores + Mask$$

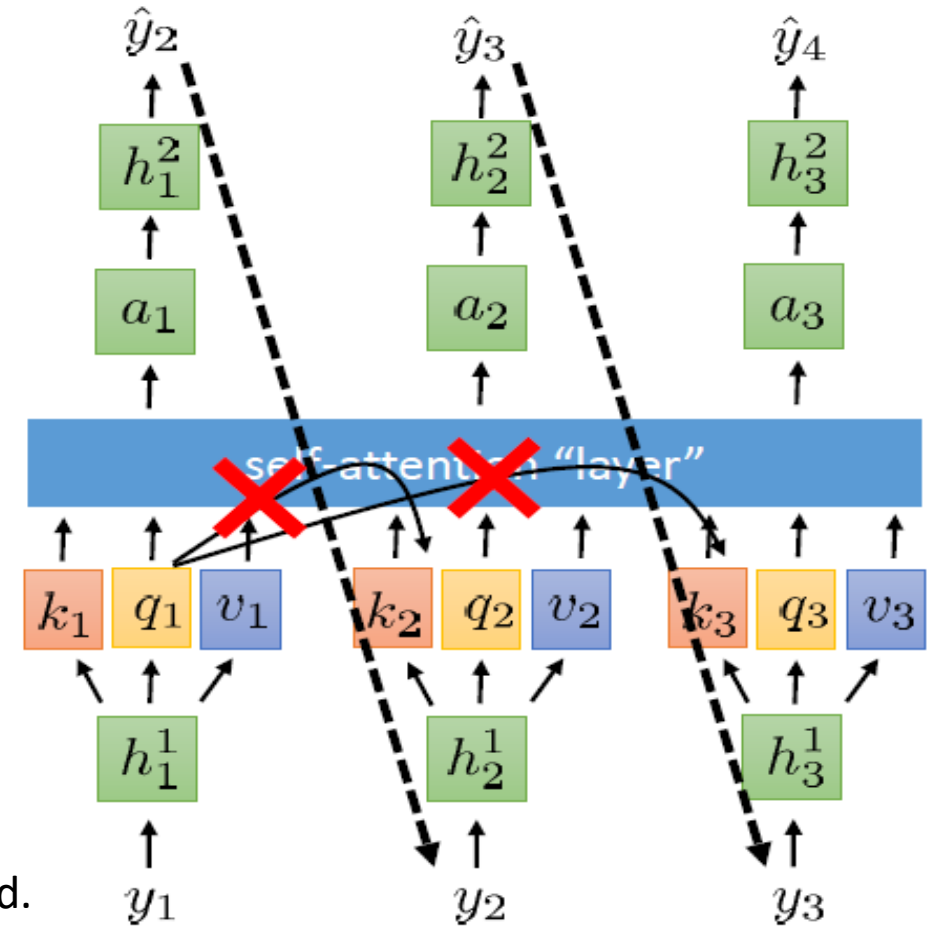
We can look at these (not greyed out) words

	[START]	The	chef	who
[START]	$-\infty$	$-\infty$	$-\infty$	$-\infty$
The		$-\infty$	$-\infty$	$-\infty$
chef			$-\infty$	$-\infty$
who				$-\infty$

For encoding these words

- The **rows** represent the token currently being encoded or predicted.
- The **columns** represent the tokens that this row is allowed to attend to.

A **crude** self-attention “language model”:



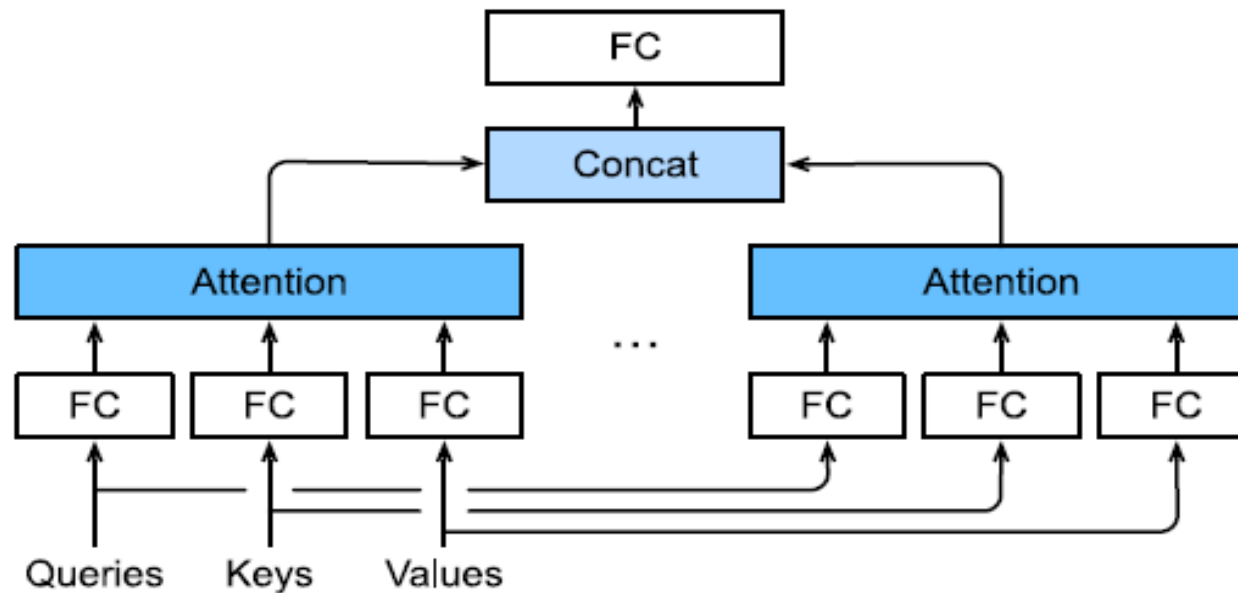
(Levine, 2021)

Multi-Head Attention

- In practice, given the same set of queries, keys and values we may want our model to combine knowledge from different behaviors of the same attention mechanism, such as capturing dependencies of various ranges (e.g., shorter-range vs. longer-range) within a sequence. Thus, it may be beneficial to allow our attention mechanism to jointly use different representation subspaces of queries, keys and values.
- Instead of performing a single attention pooling, queries, keys, and values can instead be transformed with h independently learned linear projections. These h projected queries, keys and values are fed into attention pooling in parallel.
- This design is called **multi-head attention**, where each of the h attention pooling outputs is **a head**.

Multi-Head Attention

- Give a query $q \in \mathbb{R}^{d_q}$, a key $k \in \mathbb{R}^{d_k}$ and a value $v \in \mathbb{R}^{d_v}$, each attention head h_i is computed as $h_i = f\left(W_i^{(q)} q, W_i^{(k)} k, W_i^{(v)} v\right) \in \mathbb{R}^{p_v}$ where $W_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$, $W_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$ and $W_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$ are learnable parameters and f is attention pooling such as additive attention or scaled dot product attention.
- The feed-forward layer is not expecting multiple matrices. Therefore, the output is another linear transformation via learnable parameters $W_o \in \mathbb{R}^{p_o \times M p_v}$ of concatenation of M heads: $W_o \begin{pmatrix} h_1 \\ \dots \\ h_M \end{pmatrix} \in \mathbb{R}^{p_o}$.



Multi-Head Attention

Multi-head attention expands the model's ability to focus on different positions, as well as gives the attention layer multiple “representation subspaces”, thus improving the expressivity of the model.

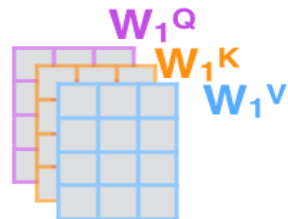
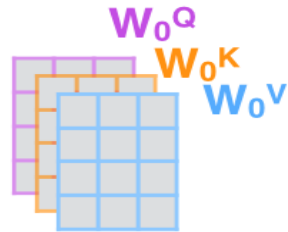
1) This is our input sentence*

Thinking
Machines

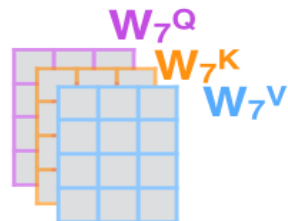
2) We embed each word*



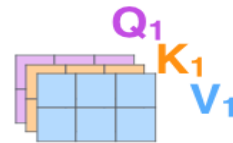
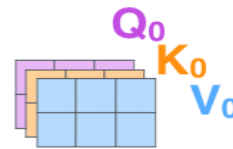
3) Split into 8 heads.
We multiply X or R with weight matrices



...



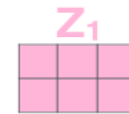
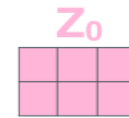
4) Calculate attention using the resulting $Q/K/V$ matrices



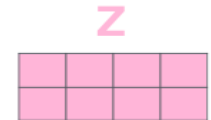
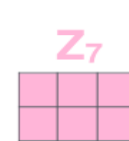
...



5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer



...



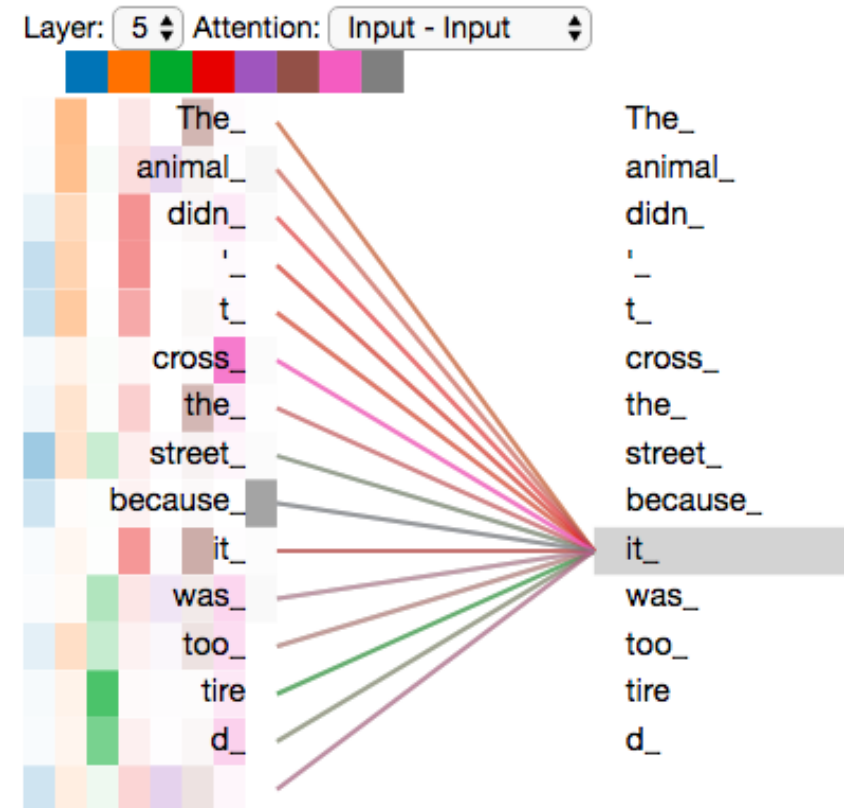
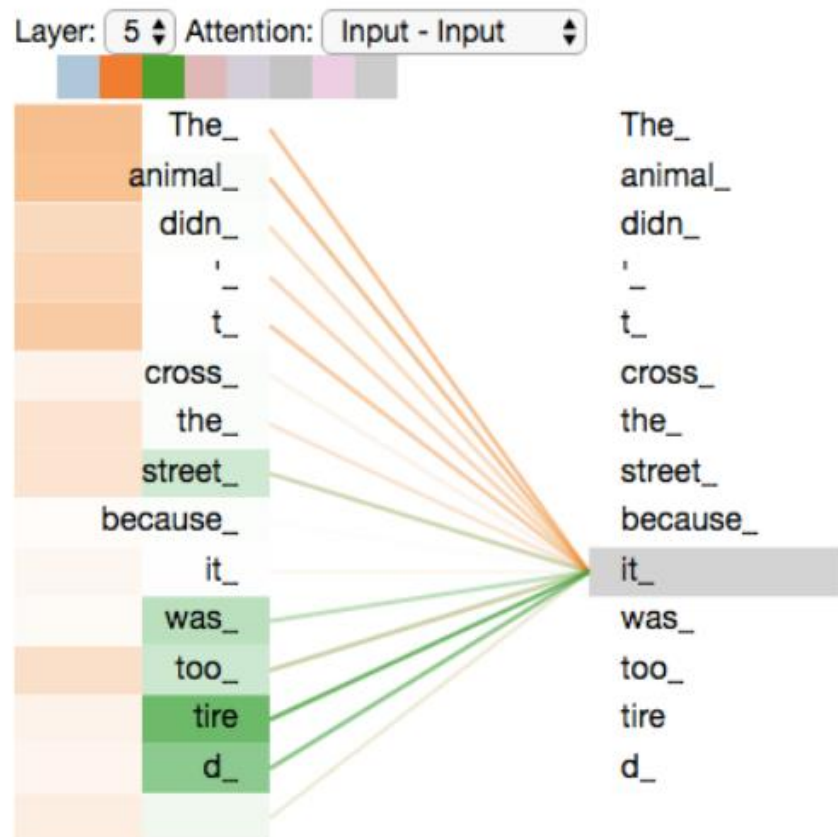
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



Multi-Head Attention: Example

As we encode the word “it”, one attention head is focusing most on “the animal”, while another is focusing on “tired” -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

However, once all attention heads are to be considered, things can get started to be harder for interpretation.



Positional Embedding

- Unlike RNNs, which recurrently process tokens of a sequence one-by-one, self-attention ditches sequential operations in favor of parallel computation.
- However, self-attention by itself does not preserve the order of sequence, that is, it is equivariant to permutations of the inputs. What should we do to account for the order of words in the input sequence when it really matters?

what we see:

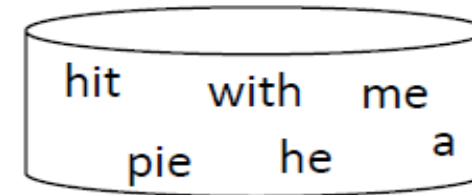
he hit me with a pie

what naïve self-attention sees:

a pie hit me with he

a hit with me he pie

he pie me with a hit



(Levine, 2021)

- The dominant approach for preserving information about the order of tokens is to represent this to the model as an additional input associated with each token. These inputs are called *positional embedding*, and can either be learned or a fixed *a priori*.

Positional Embedding

- The naïve positional encoding would just append t to the input: $\overline{x}_t = \begin{pmatrix} x_t \\ t \end{pmatrix}$. However, it would not be a great idea, because the **absolute** position is less important than the **relative** position.

I walk my dog every day



every single day I walk my dog



The fact that “my dog” is right after “I walk” is the important part, not its absolute position

- ❖ Therefore, we want to represent position in a way that tokens with similar **relative position** will have similar position encoding.
- ❖ What about using frequency-based representations?

Sequence	Index of token	Positional Encoding Matrix			
I	0	P_{00}	P_{01}	...	P_{0d}
am	1	P_{10}	P_{11}	...	P_{1d}
a	2	P_{20}	P_{21}	...	P_{2d}
Robot	3	P_{30}	P_{31}	...	P_{3d}

Positional Encoding Matrix for the sequence 'I am a robot'

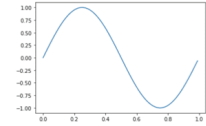
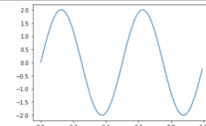
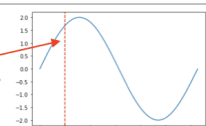
Positional Embedding: Example

- One typical scheme for fixed positional encodings are based on sine and cosine functions.
- Suppose that $X \in \mathbb{R}^{n \times d}$ contains the d -dimensional embeddings for n tokens of a sequence. The positional encoding outputs $X + P$ using a positional embedding matrix $P \in \mathbb{R}^{n \times d}$ of the same shape, whose element on the i th row and the $(2j)$ th or $(2j + 1)$ th column is given by

$$p_{i,2j} = \sin\left(\frac{i}{1000 \frac{2j}{d}}\right), p_{i,2j+1} = \cos\left(\frac{i}{1000 \frac{2j}{d}}\right)$$

► Angular Frequency (ω_j):

- $\omega_j = \frac{1}{1000 \frac{2j}{d}}$
- Each increment $i \rightarrow i + 1$ increases the argument by ω_j .

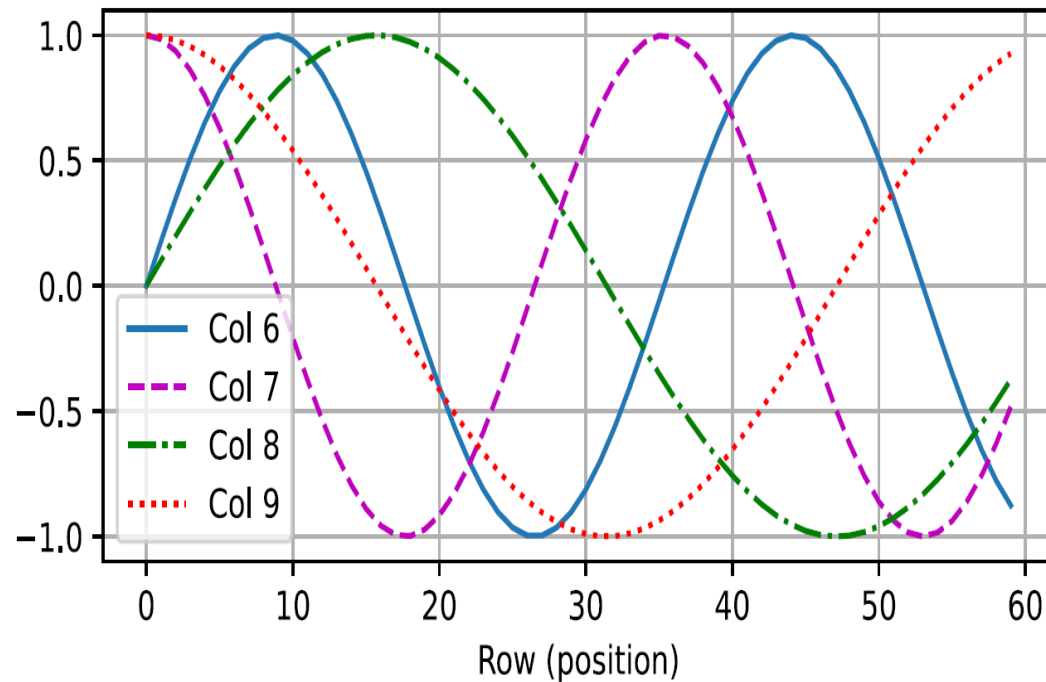
Equation	Graph	Frequency	Wavelength
$\sin(2\pi t)$		1	1
$\sin(2 * 2\pi t)$		2	1/2
$\sin(t)$		$1/2\pi$	2π
$\sin(ct)$	Depends on c	$c/2\pi$	$2\pi/c$

► Wavelength (λ_j):

- $\lambda_j = \frac{2\pi}{\omega_j} = 2\pi \times 1000 \frac{2j}{d}$
- A larger exponent $\frac{2j}{d}$ yields a larger wavelength.

Positional Embedding: Example

- In the positional embedding matrix P , rows correspond to positions within a sequence and columns represent different positional encoding dimensions.
- In the example below, we can see that the 6th and the 7th columns of positional embedding matrix have a higher frequency than the 8th and the 9th columns.



► Lower j :

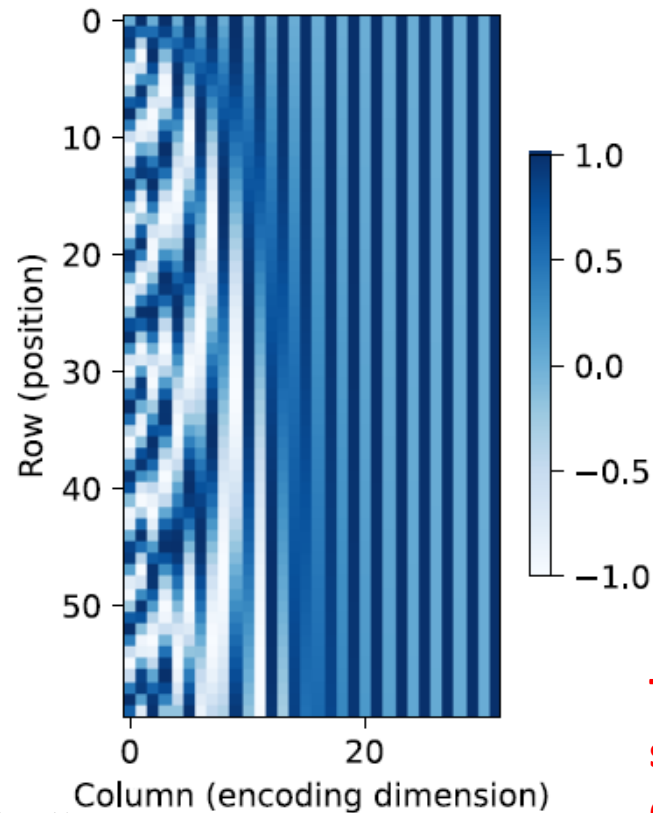
- $\omega_j \approx 1$ for small $\frac{2j}{d}$, $\lambda_j \approx 6.28$ steps in discrete math.
- Rapid oscillations over fewer positions.

► Higher j :

- ω_j becomes very small, yielding large λ_j .
- The sinusoid changes more slowly, capturing coarse-grained position.
- Suppose $d = 4$, base = 1000, and $j = 0, 1$.
 - For $j = 0$: $\omega_0 = 1/1000^0 = 1$, so $\lambda_0 = 2\pi$.
 - For $j = 1$: $\omega_1 = 1/1000^{\frac{2}{4}} = 1/1000^{0.5} = 1/\sqrt{1000}$, $\lambda_1 = 2\pi\sqrt{1000}$.
- Position i increments the phase by ω_j .
- This ensures some encodings oscillate quickly, others slowly.

Positional Embedding: Example

0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111



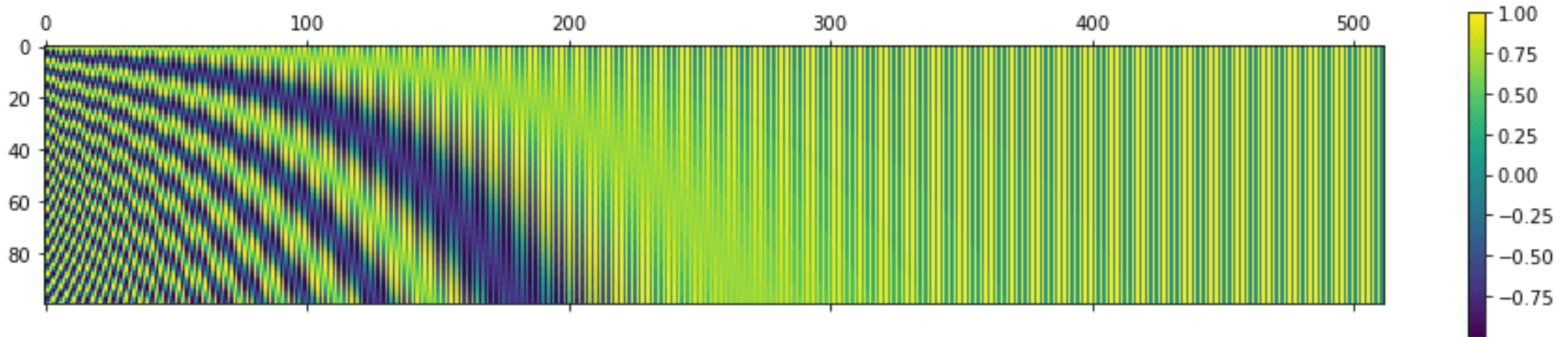
“even-odd” indicator

“first half-second half indicator” indicator

- ▶ **Monotonically decreasing frequency:** In sinusoidal positional encoding, each higher dimension has a lower frequency.
 - ▶ $\omega_j = \frac{1}{10000 \frac{2^j}{d}}$ decreases with j .
 - ▶ Encodes absolute position using increasingly coarse granularity.
- ▶ **Binary representation analogy:**
 - ▶ High-order bits change less frequently than low-order bits.
 - ▶ Similarly, high- j dimensions oscillate more slowly in positional encoding.
- ▶ **Continuous vs. Binary:** Float-based sinusoidal encodings are *more space-efficient* and allow fractional offsets.

The goal is not to represent position perfectly in a symbolic sense, but to preserve enough information so attention layers can recover useful positional relationships.

Positional Embedding



► Absolute Position:

- Different frequencies let the model pinpoint an absolute location in the sequence.
- Example: $i = 10$ vs. $i = 100$ produce noticeably different multi-sinusoidal phases.

► Relative Position:

- Key property: For an offset δ , $PE(i + \delta)$ can be derived from $PE(i)$ via a linear transformation (thanks to trig identities).
- Allows model to easily learn relative shifts (e.g., paying attention to positions $i + 1, i + 2$) without separate parameters.

- **Result:** The encoding provides both absolute and relative cues in a single representation.

► Positional Encoding Analogy:

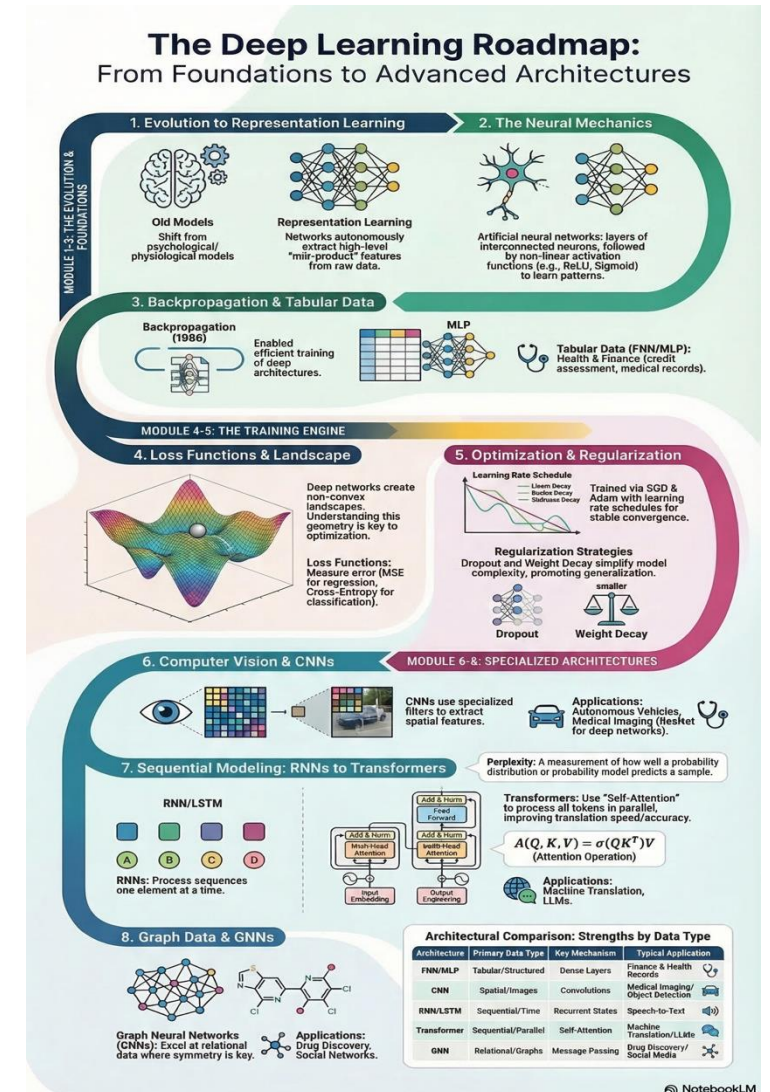
- In dimension 0, sine wave might oscillate within a handful of steps.
- In dimension $d - 1$, the wave might only complete a fraction of a single period over the entire sequence.
- This layered range of frequencies parallels the layered range of bit flips in binary.

► Space Efficiency:

- Instead of storing a discrete bit for each position, the continuous sinusoidal approach uses fewer dimensions to encode both small and large positional changes.
- Facilitates interpolation and generalization across different sequence lengths.

Content

7 Graph Data and Graph Neural Networks (GNNs/GCNs)



Graph-Structured Data

Graph-structured data is a type of data representation where **entities (nodes)** and their **relationships (edges)** are explicitly modeled as a graph. This structure captures the connections between data points, allowing for more effective analysis of relational patterns.

Examples:

- Social networks, citation networks, multi-agent systems
- Knowledge graphs
- Recommendation System
- Protein interaction networks
- Molecules
- Road maps
- Brain networks

Graphs are a general language for describing and analyzing entities with relations/interactions

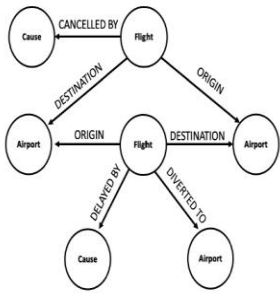
Why Are Graph-structured Data Important?

Graphs capture **complex relationships and dependencies** between entities:

- **Interconnected entities influence each other** (e.g., in social networks, a person's behavior depends on their connections).
- **Knowledge is structured in relational forms** (e.g., in knowledge graphs, concepts are linked based on meaning and context).
- **Biological and medical data exhibit intricate interactions** (e.g., protein-protein interaction networks, brain connectivity graphs).

By modeling data as graphs, we can better understand structures, uncover hidden patterns, and improve AI-driven decision-making.

Graph-Structured Data is Everywhere

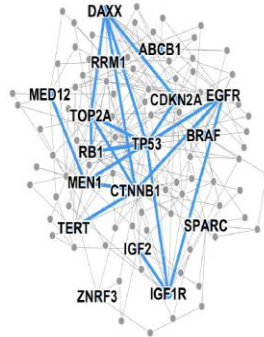


Event Graphs



Image credit: [SalientNetworks](#)

Computer Networks



Disease Pathways



Image credit: [Medium](#)

Social Networks

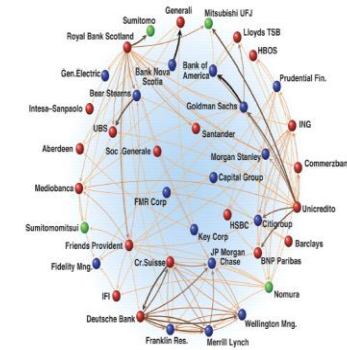


Image credit: [Science](#)

Economic Networks



Image credit: [Lumen Learning](#)

Communication Networks

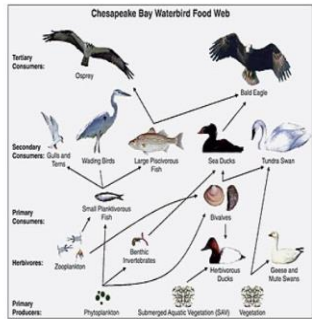


Image credit: [Wikipedia](#)

Food Webs

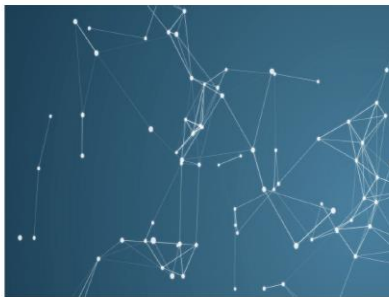


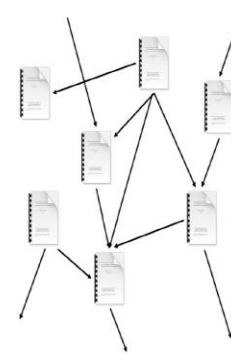
Image credit: [Pinterest](#)

Particle Networks



Image credit: [visitlondon.com](#)

Underground Networks



Citation Networks



Image credit: [Missoula Current News](#)

Internet

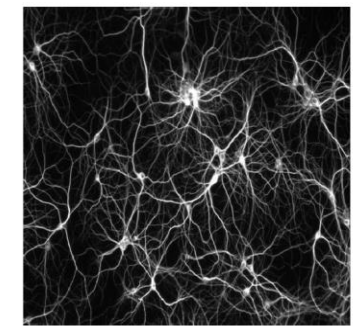


Image credit: [The Conversation](#)

Networks of Neurons

StanfordCS224w

Graph-Structured Data is Everywhere

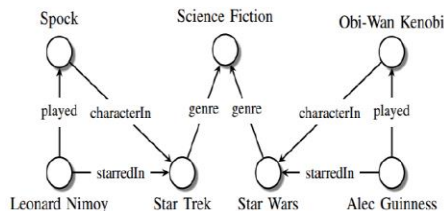


Image credit: [Maximilian Nickel et al](#)

Knowledge Graphs

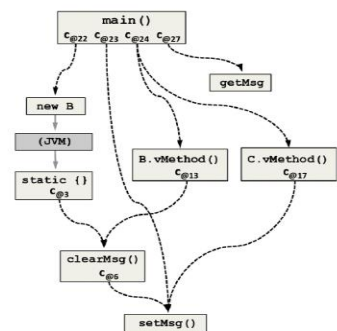


Image credit: [ResearchGate](#)

Code Graphs

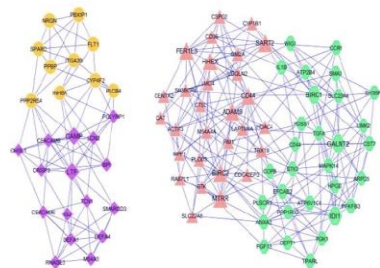
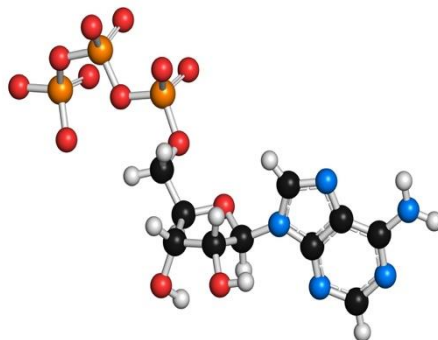


Image credit: [ese.wustl.edu](#)

Regulatory Networks



Molecules

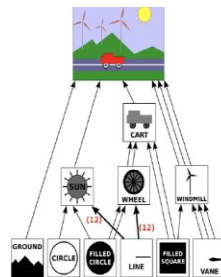


Image credit: [math.hws.edu](#)

Scene Graphs

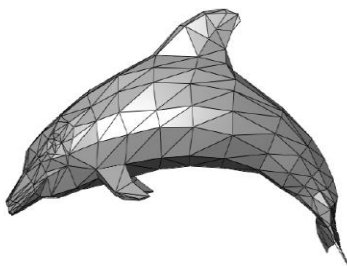
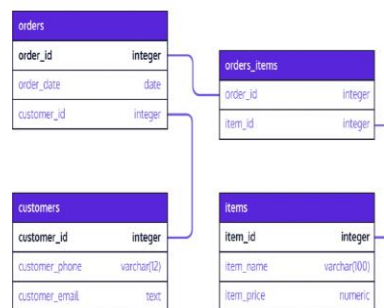
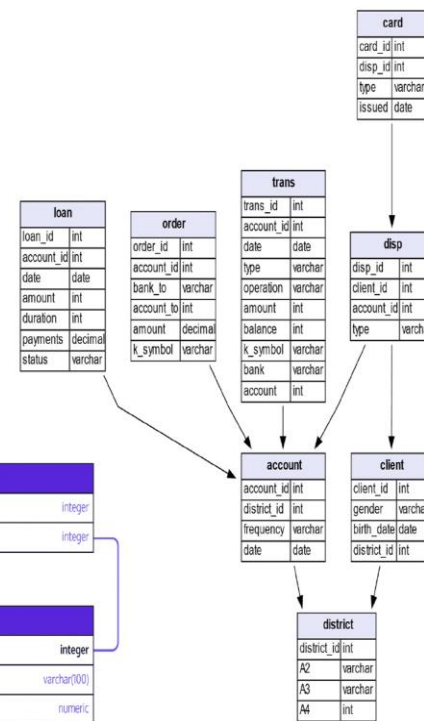


Image credit: [Wikipedia](#)

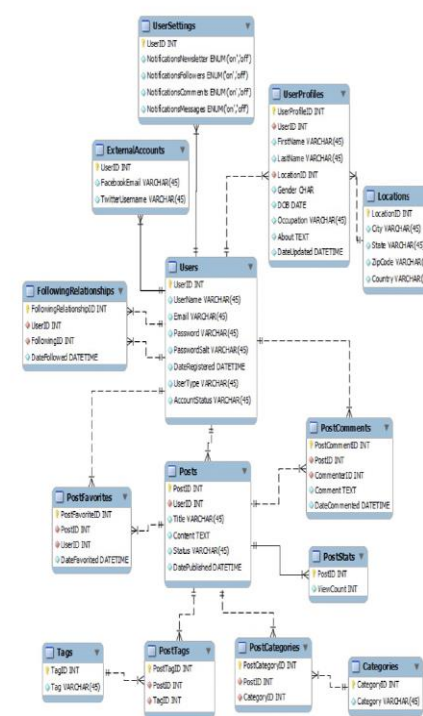
3D Shapes



Commerce



Finance



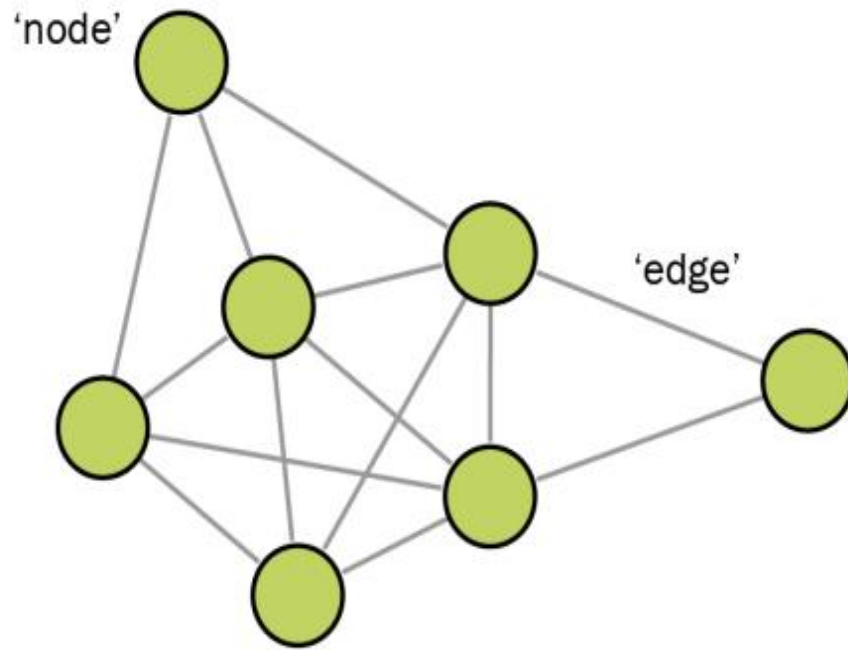
Social Media

StanfordCS224w

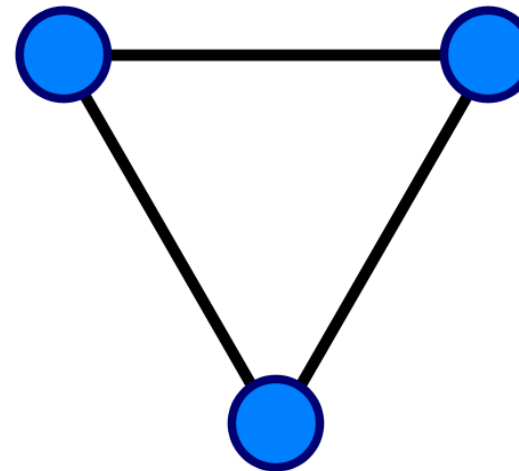
MORE THAN A JOURNEY

Graph Set-up

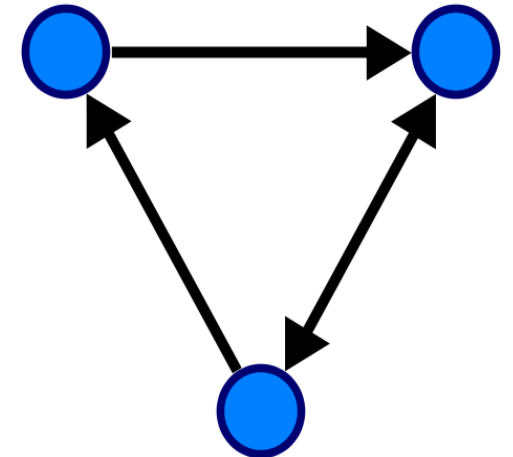
- Graph $G = (V, E)$ is defined by a set of nodes V and a set of edges E between these nodes. An edge going from node $u \in V$ to node $v \in V$ as $(u, v) \in E$.



Undirected

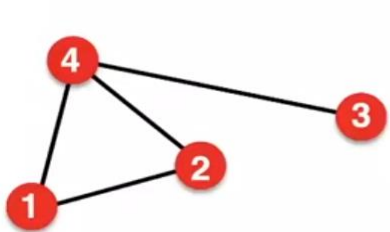


Directed



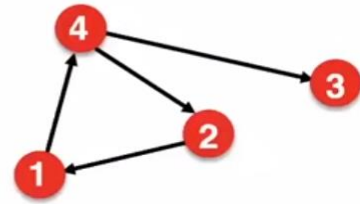
Adjacency Matrix

- A convenient way to represent graphs is through an adjacency matrix $A \in \mathbb{R}^{|V| \times |V|}$. We order the nodes in the graph so that every node indexes a particular row and column in the adjacency matrix.



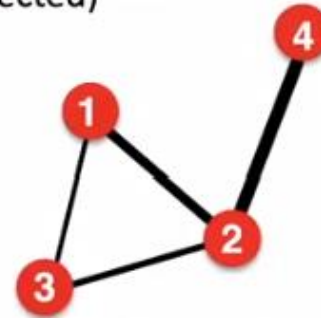
$A_{ij} = 1$ if there is a link from node i to node j
 $A_{ij} = 0$ otherwise

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

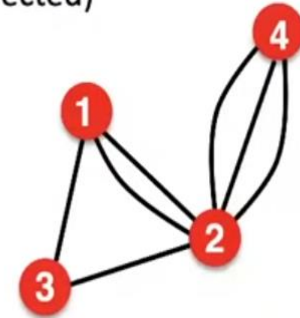


$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

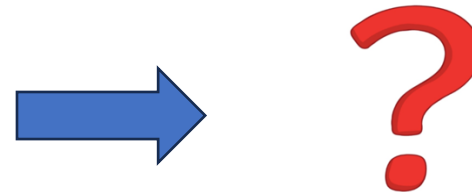
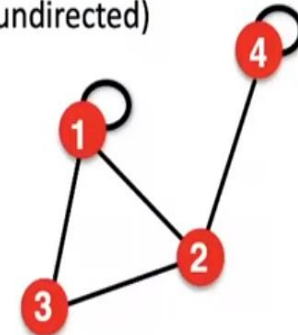
Weighted
 (undirected)



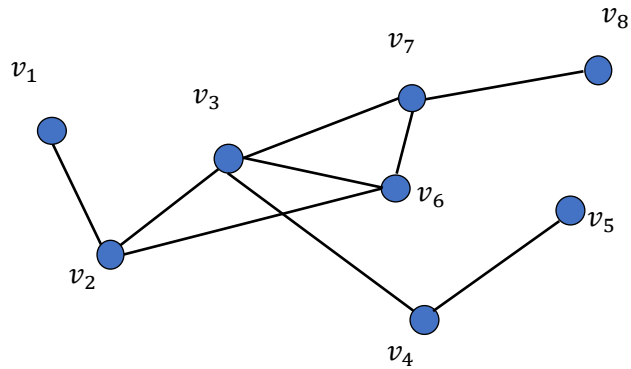
Multigraph
 (undirected)



Self-edges (self-loops)
 (undirected)



Graphs and Graph Signals



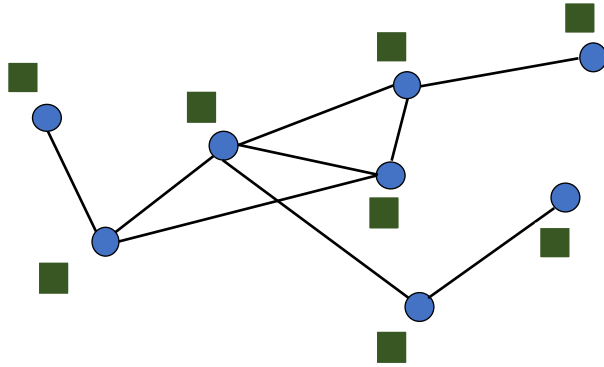
$$\mathcal{V} = \{v_1, \dots, v_N\}$$

$$\mathcal{E} = \{e_1, \dots, e_M\}$$

$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

195

Graphs and Graph Signals



Graph Signal: $f : \mathcal{V} \rightarrow \mathbb{R}^N$

$$\mathcal{V} = \{v_1, \dots, v_N\}$$

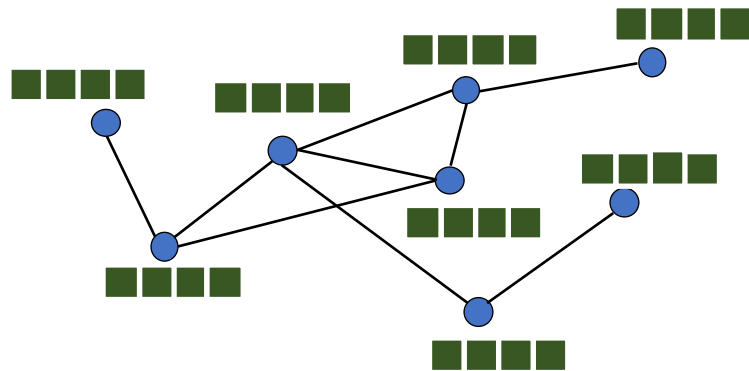
$$\mathcal{E} = \{e_1, \dots, e_M\}$$

$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

196

$$v \rightarrow \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \end{bmatrix}$$

Graphs and Graph Signals



Graph Signal: $f : \mathcal{V} \rightarrow \mathbb{R}^{N \times d}$

$$\mathcal{V} = \{v_1, \dots, v_N\}$$

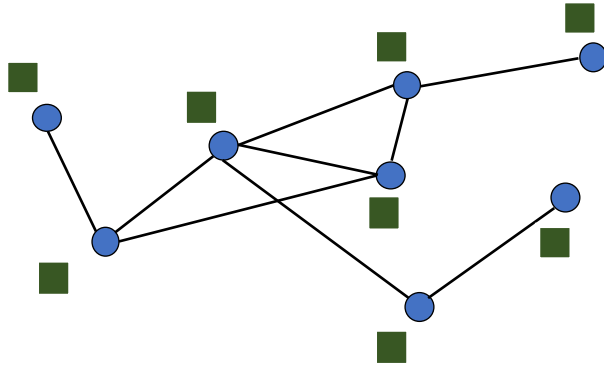
$$\mathcal{E} = \{e_1, \dots, e_M\}$$

$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

197

$$v \rightarrow \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \end{bmatrix}$$

Graphs and Graph Signals



Graph Signal: $f : \mathcal{V} \rightarrow \mathbb{R}^N$

$$\mathcal{V} = \{v_1, \dots, v_N\}$$

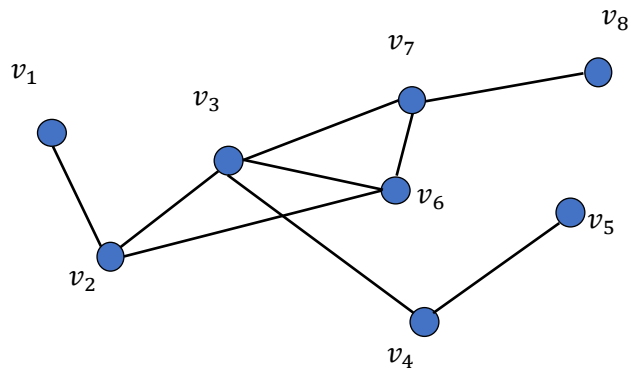
$$\mathcal{E} = \{e_1, \dots, e_M\}$$

$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

198

$$v \rightarrow \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \end{bmatrix}$$

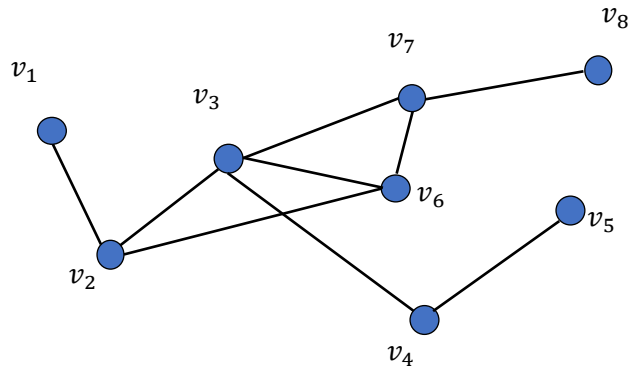
Matrix Representations of Graphs



199

Spectral graph theory. American Mathematical Soc.; 1997.

Matrix Representations of Graphs



Adjacency Matrix: $A[i, j] = 1$ if v_i is adjacent to v_j
 $A[i, j] = 0$, otherwise

Adjacency Matrix

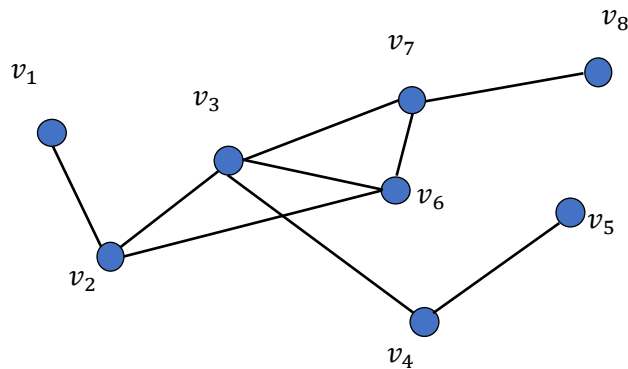
200

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

A

Spectral graph theory. American Mathematical Soc.; 1997.

Matrix Representations of Graphs



Adjacency Matrix: $A[i, j] = 1$ if v_i is adjacent to v_j
 $A[i, j] = 0$, otherwise

Degree Matrix: $\mathbf{D} = \text{diag}(\text{degree}(v_1), \dots, \text{degree}(v_N))$

Degree Matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$\mathbf{D}$

Adjacency Matrix

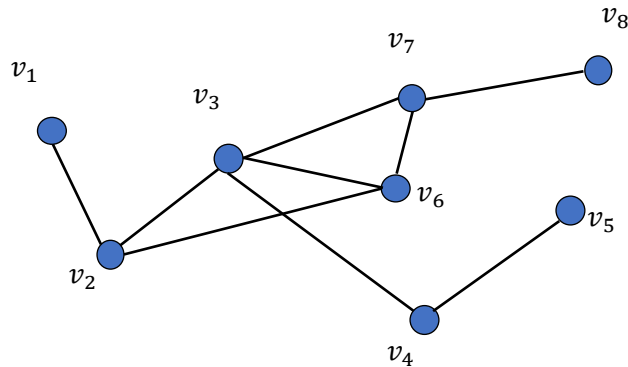
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$\mathbf{A}$

201

Spectral graph theory. American Mathematical Soc.; 1997.

Matrix Representations of Graphs



Adjacency Matrix: $A[i, j] = 1$ if v_i is adjacent to v_j
 $A[i, j] = 0$, otherwise

Degree Matrix: $\mathbf{D} = \text{diag}(\text{degree}(v_1), \dots, \text{degree}(v_N))$

Degree Matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$\mathbf{D}$

Adjacency Matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$\mathbf{A}$

202

Laplacian Matrix

$$= \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & -1 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & -1 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$

$\mathbf{L}$

Spectral graph theory. American Mathematical Soc.; 1997.

Node, Edge, and Global Features

Node features represent **characteristics or attributes** of individual nodes for downstream tasks like **node classification, clustering, and link prediction**.

Common Types of Node Features

- ❖ **Categorical Features:** Node types (e.g., "user" or "product" in a recommendation system).
- ❖ **Numerical Features:** Values like age, price, or degree centrality.
- ❖ **Textual Features:** Descriptions, reviews, or labels in textual form.
- ❖ **Vectorized Embeddings:** Learned representations from NLP models or pre-trained embeddings.

Edge features define **relationships or interactions** between nodes for **link prediction and edge classification**.

Common Types of Edge Features

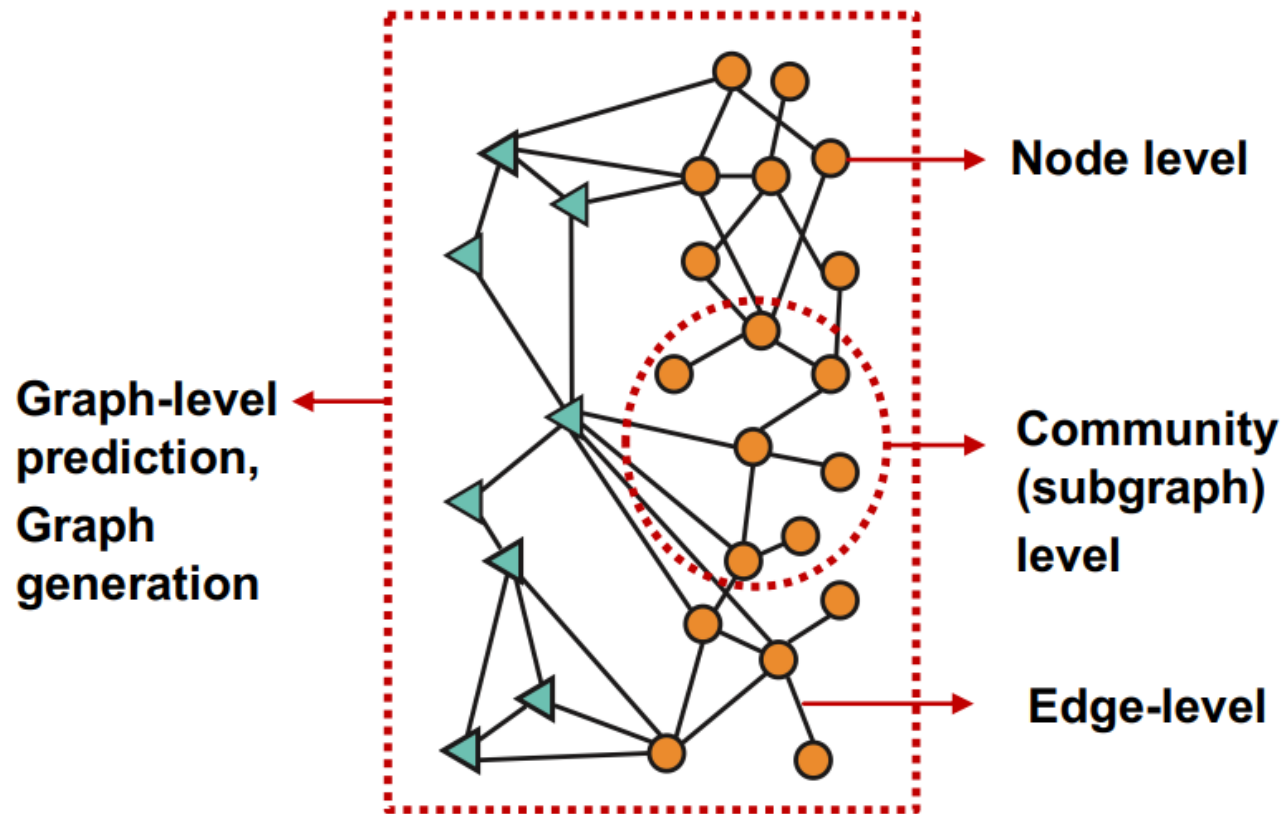
- **Weight:** The strength or importance of a connection (e.g., frequency of interactions).
- **Type:** The kind of relationship (e.g., friendship, purchase, citation).
- **Timestamp:** When the connection was established (useful for dynamic graphs).
- **Directionality:** Whether the edge is directed or undirected.

Graph-Level Features: graphs have global properties or features that apply to the entire network.

Examples include:

- **Graph Density** (How connected is the graph?).
- **Average Clustering Coefficient** (Tendency of nodes to form clusters).
- **Graph Size** (Number of nodes and edges).

Different Types of Task



Graph-based machine learning involves multiple tasks categorized by the focus of analysis. The main categories of tasks include:

- ▶ **Node-Level Tasks:** Predicting properties of individual nodes.
- ▶ **Edge-Level Tasks:** Inferring relationships between node pairs.
- ▶ **Community-Level Tasks:** Detecting and analyzing groups of closely connected nodes.
- ▶ **Graph-Level Tasks:** Understanding global graph properties.

GNN Designs

📌 Summary

Step

1. Define Graph
2. Feature Engineering
3. Message Passing
4. Choose Architecture
5. Loss Function
6. Training
7. Evaluation
8. Deployment

Task

Nodes, edges, features

Define node and edge features

Select aggregation method

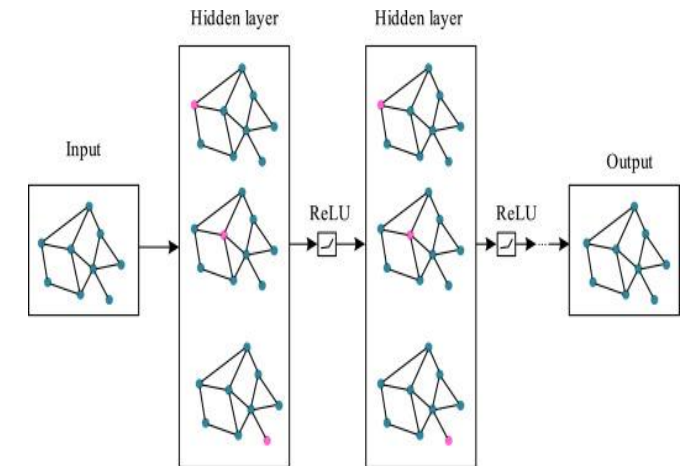
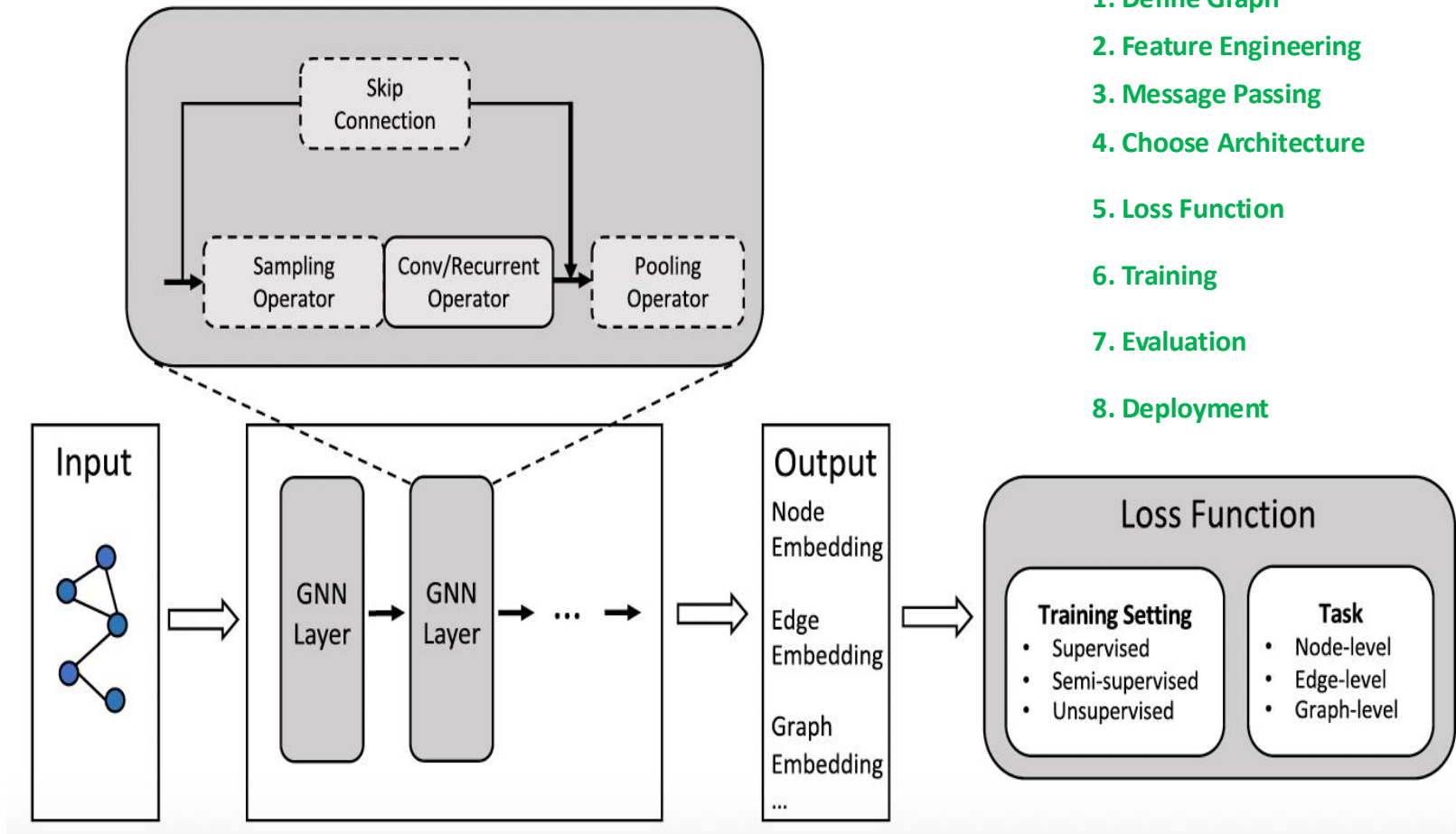
GCN, GAT, GraphSAGE, etc.

Supervised (cross-entropy), unsupervised (contrastive)

Use mini-batching and optimizers

Classification, link prediction, graph-level tasks

Optimize for inference speed



Jie Zhou, et al. (2020). AI Open

Key Modules in Graph Neural Networks

GNNs process graph-structured data by propagating and aggregating information across nodes and edges.

Three key modules in GNNs:

❖ Sampling Module

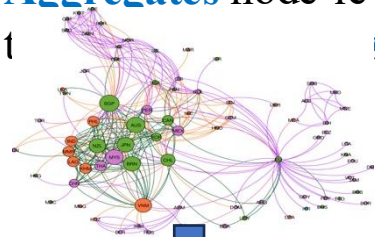
Aims to **reduce the size** of each node's neighborhood, especially for **large graphs**, preventing the neighbor explosion problem.

❖ Propagation Module

- Performs **message passing** via **convolutions** (e.g., GCNs) or **recurrent operators** (e.g., GRUs) on node features.
- Uses **skip connections** to mitigate over-smoothing and incorporate historical representations.

❖ Pooling Module

Aggregates node-level embeddings into **subgraph or graph-level** representations, extracting higher-level features needed for **classification**.



Sampling Module

- Reduces neighborhood size
- Efficient processing on large graphs

Propagation Module

- Convolution / Recurrent ops
- Skip connections mitigate over-smoothing
- Aggregates feature + topological info

Pooling Module

- Aggregates node reps into subgraph/graph-level reps
- Extracts high-level features

Tasks

- Node classification
- Link prediction
- Graph classification
- Anomalous node detection
- Clustering
-

The Sampling Module

- Efficient Graph Processing via Sampling
 - ► Direct propagation on large graphs is computationally infeasible.
 - ► The Sampling Module reduces cost by selecting subsets of nodes or edges.
- Key Challenge:
 - ❖ **Neighbor Explosion:** The number of neighbors grows exponentially with depth. GNNs aggregate messages from each node's neighbors in the previous layer. Tracking back multiple layers can exponentially increase the neighbor set. Storing and processing all neighborhood information becomes intractable for large graphs.
 - ❖ **Computational Efficiency:** Full neighbor aggregation is impractical for large graphs.
 - ❖ **Memory Constraints:** Storing all neighborhood information for each node is infeasible.
 - ❖ **Scalability:** Enables GNNs to handle large graphs effectively.
- Common sampling techniques: Node Sampling; Layer Sampling; Subgraph Sampling.

Impact on Permutation Properties:

- Node-level predictions remain unchanged under node reordering.
- Node representations transform consistently when input ordering changes.

Impact on Task Performance:

- Preserve downstream performance in classification, link prediction, etc.
- Sampling strategies must capture essential structural information despite reduced neighborhood size.
- Aim for low variance while avoiding high computational costs.

Common Sampling Methods

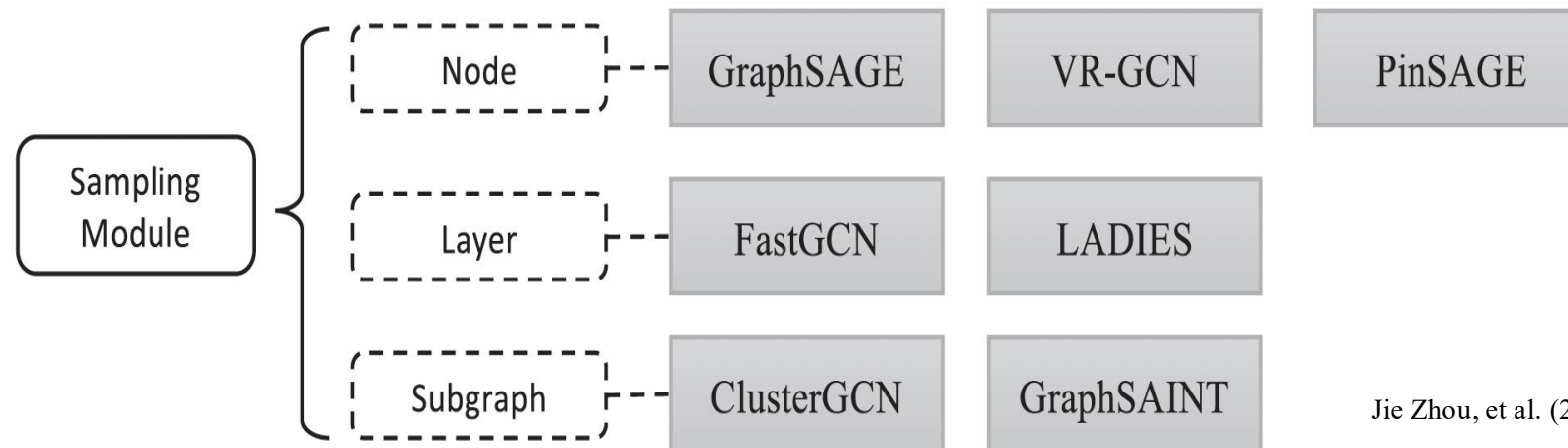
- **Node Sampling:** Selects a subset of nodes and their immediate neighbors.
 - ▶ Reduces computational complexity by limiting the number of participating nodes.
 - ▶ Often used in algorithms like GraphSAGE.

Layer Sampling: It selects a fixed number of neighbors per layer.

- ▶ Controls exponential growth by restricting the number of aggregated neighbors.
- ▶ Balances efficiency and performance in large-scale graphs.

Subgraph Sampling: Extracts a subgraph based on connectivity patterns.

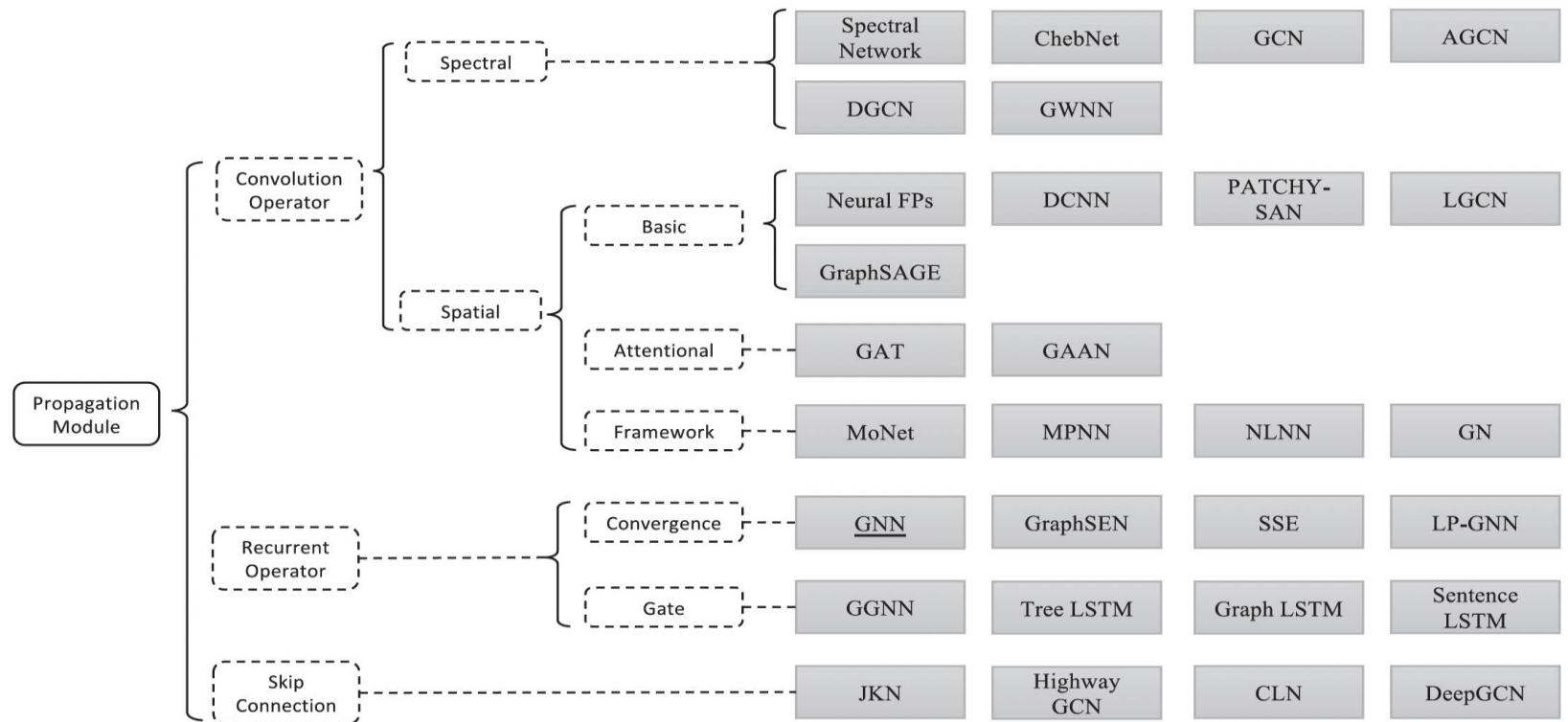
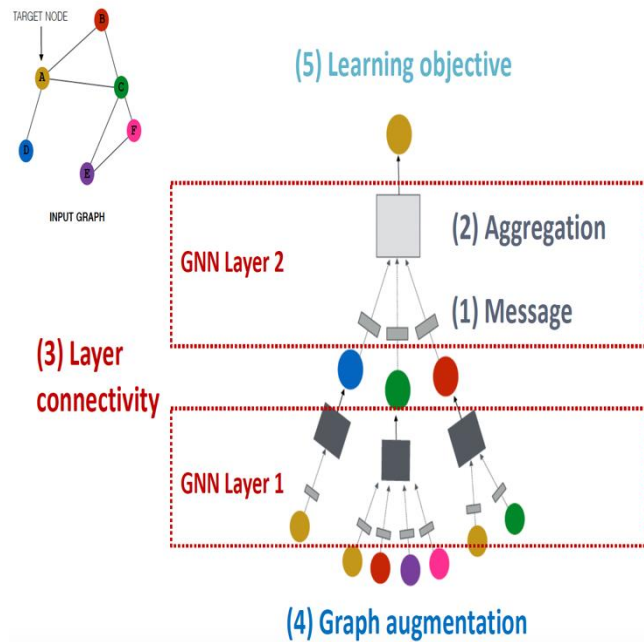
- ▶ Useful for mini-batch training by working on graph partitions.
- ▶ Preserves graph topology while reducing computation.



Jie Zhou, et al. (2020). AI Open

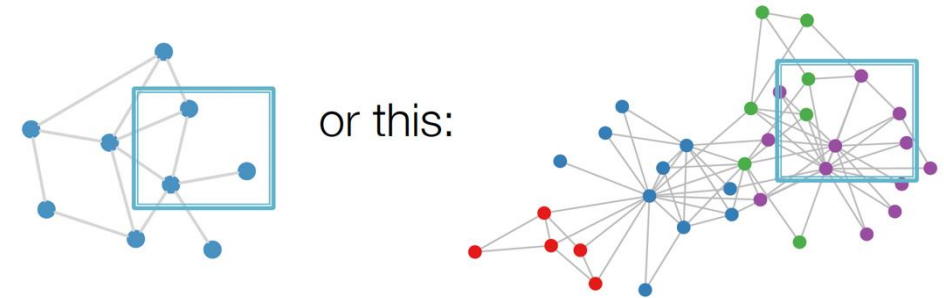
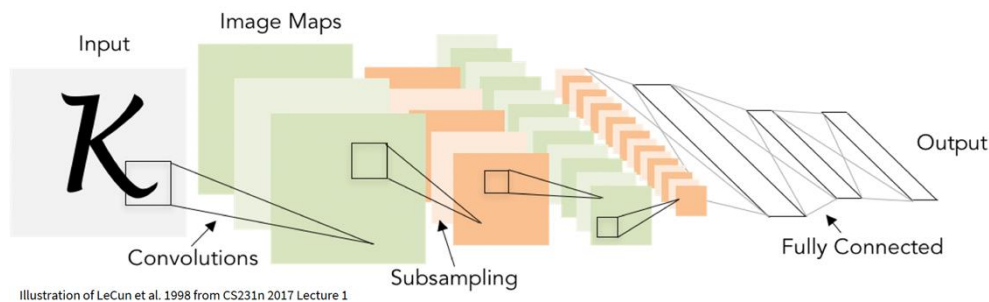
The Propagation Module

- Facilitates message passing between nodes to integrate structural and feature information.
- Key operations:
 - ❖ **Convolution Operators:** Aggregate neighbor information.
 - ❖ **Recurrent Operators:** Maintain temporal dependencies in dynamic graphs (e.g., Graph GRU, Graph LSTM).
 - ❖ **Skip Connections:** Mitigate over-smoothing by retaining historical representations.



Jie Zhou, et al. (2020). AI Open

Permutation Equivariance and Invariance

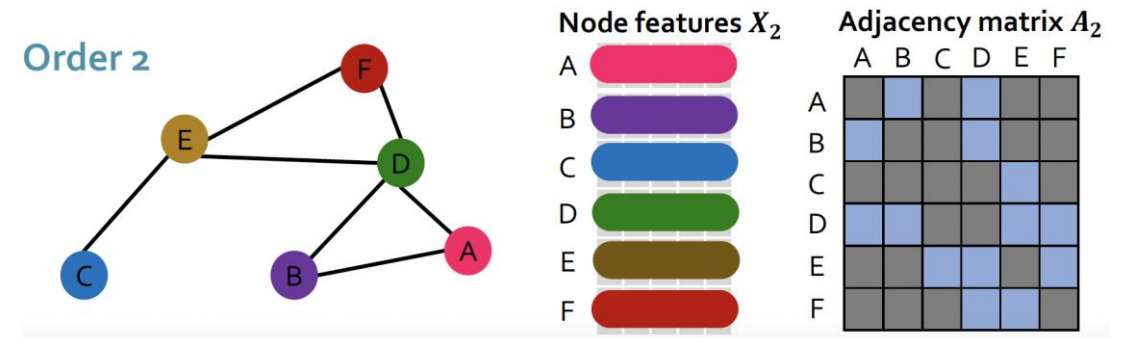
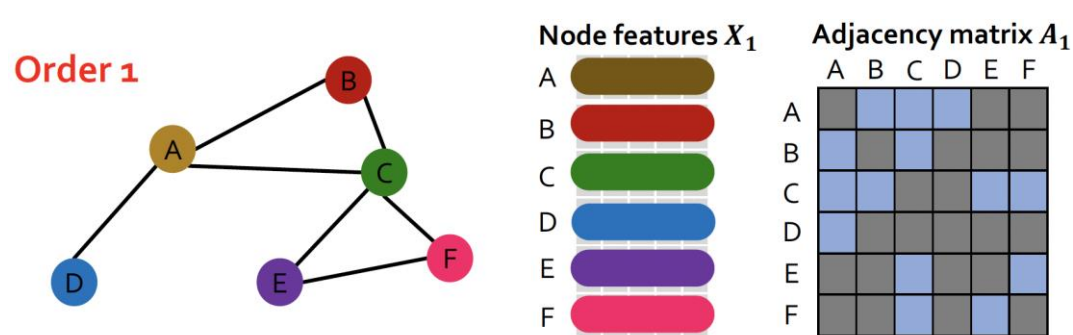


Key Observation:

- ▶ A graph does not have a fixed, canonical ordering of its nodes.
- ▶ Any permutation of node indices can still represent the same underlying graph.

Implication:

- ▶ The labeling or numbering of nodes is arbitrary.
- ▶ Reorder node IDs without changing the graph's structure.



Definition of PE and PI

Permutation Invariance:

- ▶ A function $f(\cdot)$ is *invariant* if permuting inputs does not change the output:

$$f(\pi(A, X)) = f(A, X), \quad \forall \pi \in S_n.$$

- ▶ Typical for **graph-level** tasks (e.g., entire graph classification).

Permutation on Graphs:

- ▶ Let P be an $n \times n$ permutation matrix.
- ▶ Then $A \mapsto PAP^T$, $X \mapsto PX$.
- $f(A, X) = \mathbf{1}^T X$: **Permutation-invariant**
 - Reason: $f(PAP^T, PX) = \mathbf{1}^T PX = \mathbf{1}^T X = f(A, X)$
- $f(A, X) = AX$: **Permutation-equivariant**
 - Reason: $f(PAP^T, PX) = PAP^T PX = PAX = Pf(A, X)$
- $f(A, X) = X$: **Permutation-equivariant**
 - Reason: $f(PAP^T, PX) = PX = Pf(A, X)$

Permutation Equivariance:

- ▶ A function $g(\cdot)$ is *equivariant* if the output is permuted in the same way:

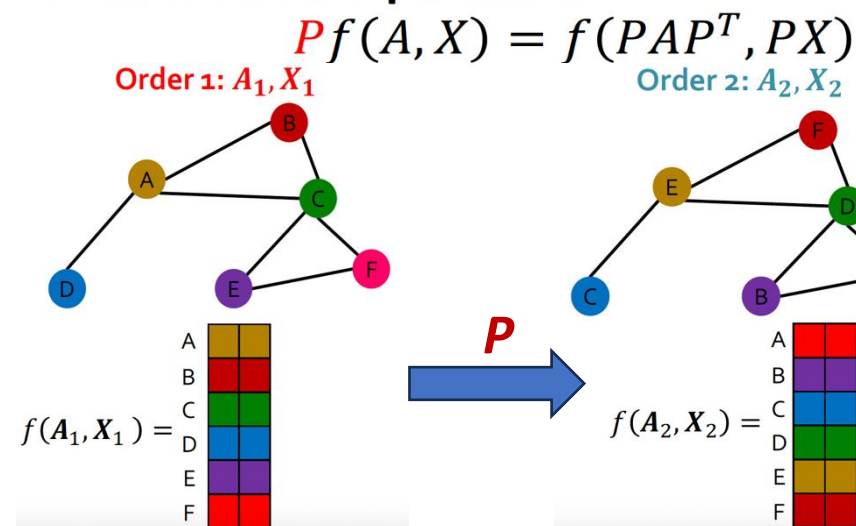
$$g(\pi(A, X)) = \pi(g(A, X)), \quad \forall \pi \in S_n.$$

- ▶ Typical for **node-level** tasks (e.g., node embeddings, node classification).

■ Permutation-invariant

$$f(A, X) = f(PAP^T, PX)$$

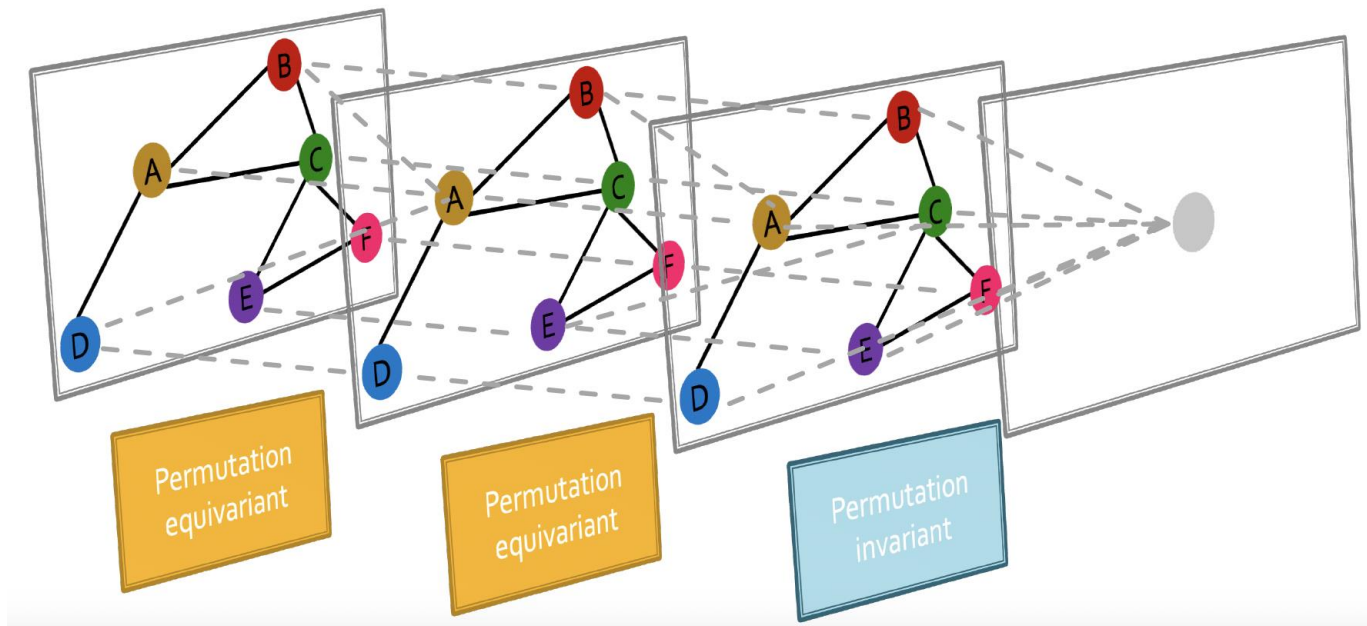
■ Permutation-equivariant



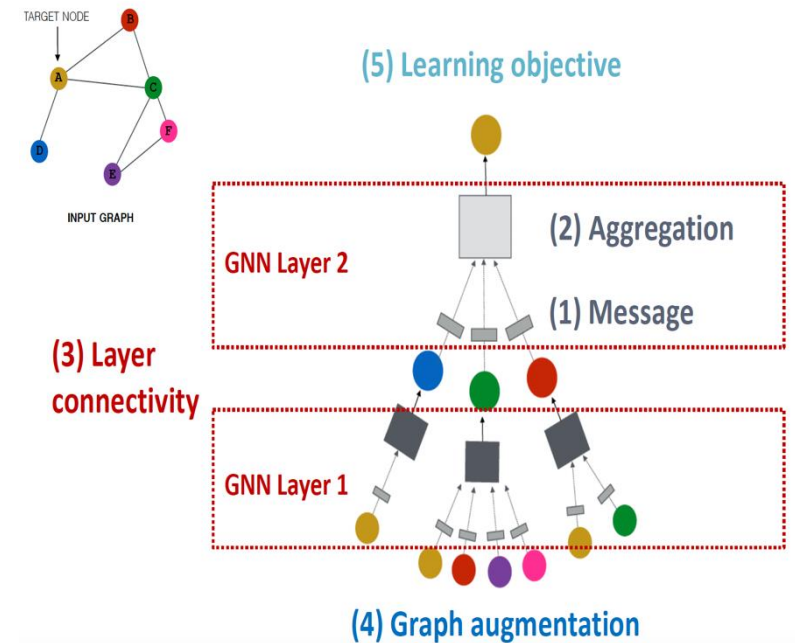
Designing GNN

- **Designing GNN Layers** must preserve or respect permutations at each update step. PI and PE are crucial for robust GNN models that handle node reorderings gracefully.
- ❖ **Sampling + Approximation:** Avoid violating permutation properties in large-scale graphs (random sampling, etc.).
- ❖ **Pooling Mechanisms:** Summation/average pooling ensures invariant graph-level outputs.
- ❖ **Challenges:** Hierarchical pooling, dynamic graphs, and advanced aggregator designs can complicate these properties.

GNN consist of multiple permutation equivariant / invariant functions.

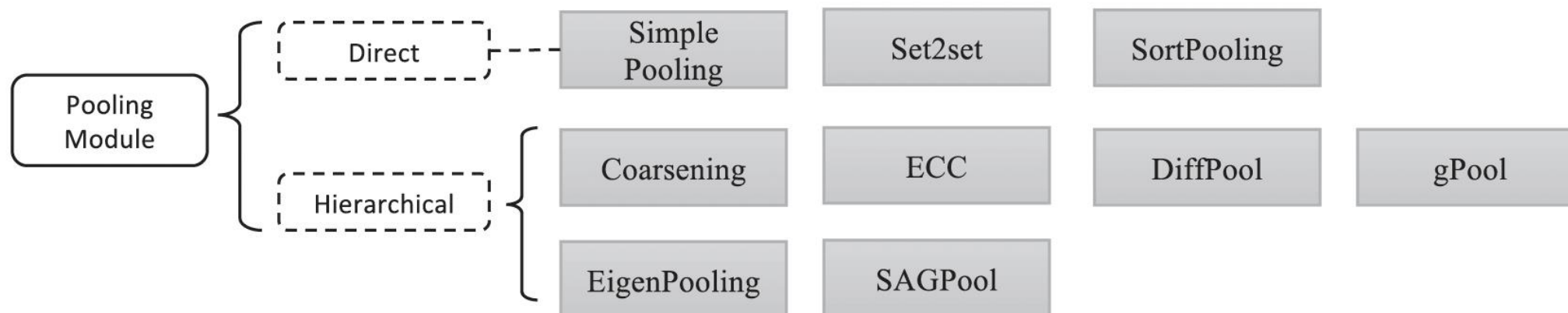


A general GNN framework



The Pooling Module

- Extracting High-Level Representations
 - ❖ Generates compact representations of subgraphs or entire graphs.
 - ❖ Essential for tasks like graph classification and hierarchical learning.
- Key pooling techniques:
 - ❖ Node Dropout Pooling: Drops less informative nodes (e.g., Top-K pooling).
 - ❖ Cluster-based Pooling: Merges similar nodes into clusters (e.g., DiffPool).
 - ❖ Attention-based Pooling: Assigns weights to nodes based on learned importance.
 - ❖ Maintaining Permutation Invariance: Ensures that graph representations remain unchanged.
- Two main categories:
 - ❖ Direct (Readout) Pooling Modules: Aggregate node embeddings into a single graph-level embedding in one step.
 - ❖ Hierarchical Pooling Modules: Iteratively coarsen (or cluster) the graph, creating a hierarchy of smaller graphs or subgraphs.



Jie Zhou, et al. (2020). AI Open

GNN Training Framework

Training Approaches

- ❖ **Supervised Learning:** Uses labeled data to train GNNs for node/graph classification.
- ❖ **Semi-supervised Learning:** Uses both labeled and unlabeled data to improve training.
- ❖ **Unsupervised Learning:** Uses self-supervision (e.g., contrastive learning) to learn node embeddings.

Prediction Tasks in GNNs

- ❖ **Node-focused:** Predicts node labels (e.g., node classification) using an MLP or softmax layer.
- ❖ **Edge-focused:** Predicts relationships between nodes (e.g., link prediction) using similarity functions or MLPs.
- ❖ **Graph-focused:** Generates graph embeddings using pooling layers for tasks like graph classification.

Cross-Entropy Loss: Example

$$L = \sum_{i \in V_{train}} y_i \log(\sigma(h_i^T \theta)) + (1 - y_i) \log(1 - \sigma(h_i^T \theta))$$

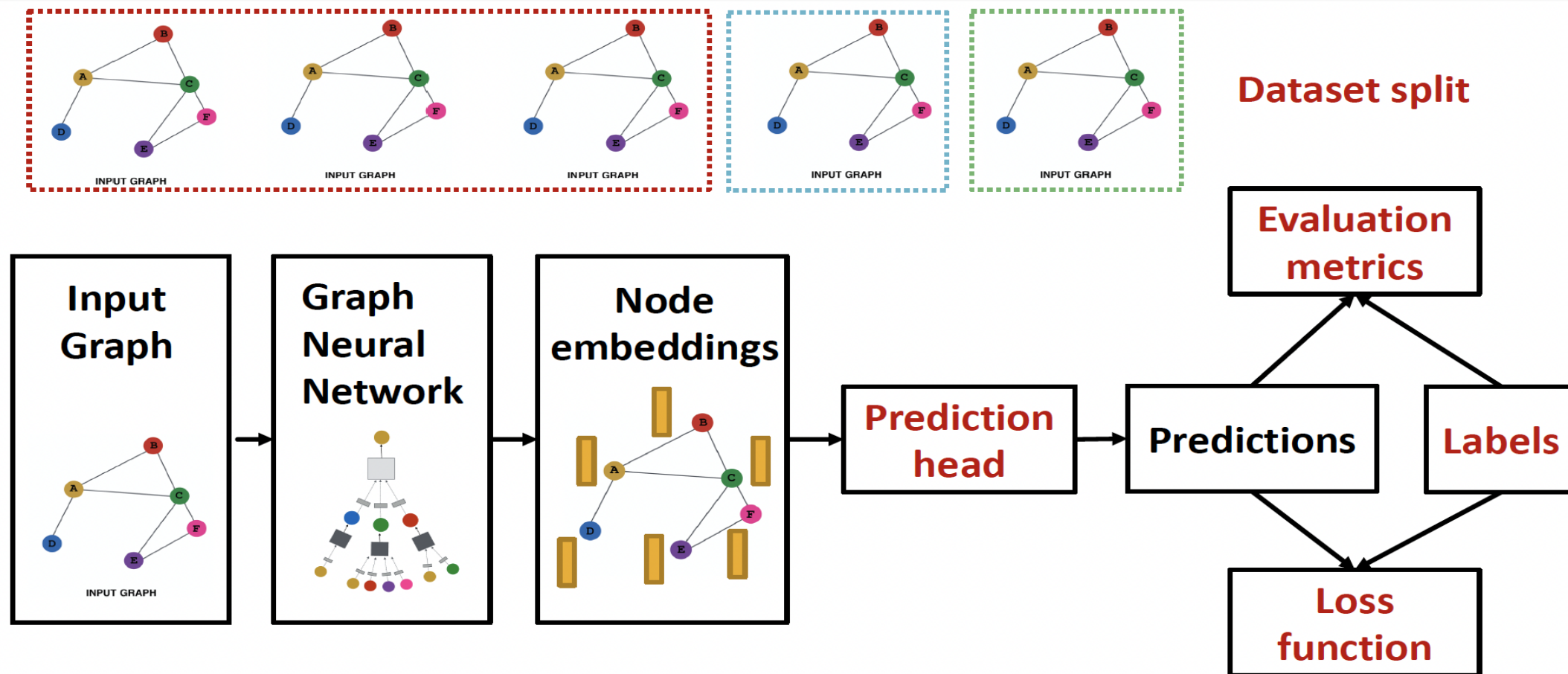
where:

- ▶ h_i is the node embedding at the last GNN layer.
- ▶ y_i is the true class label of node i .
- ▶ θ represents the classification weights.
- ▶ σ is the sigmoid function.

Types of Nodes in GNN Training

- ❖ **Training Nodes:** Used in loss computation.
- ❖ **Transductive Test Nodes:** Processed in GNN but not included in loss computation.
- ❖ **Inductive Test Nodes:** Not included in GNN computation or loss function.

GNN Training Pipeline

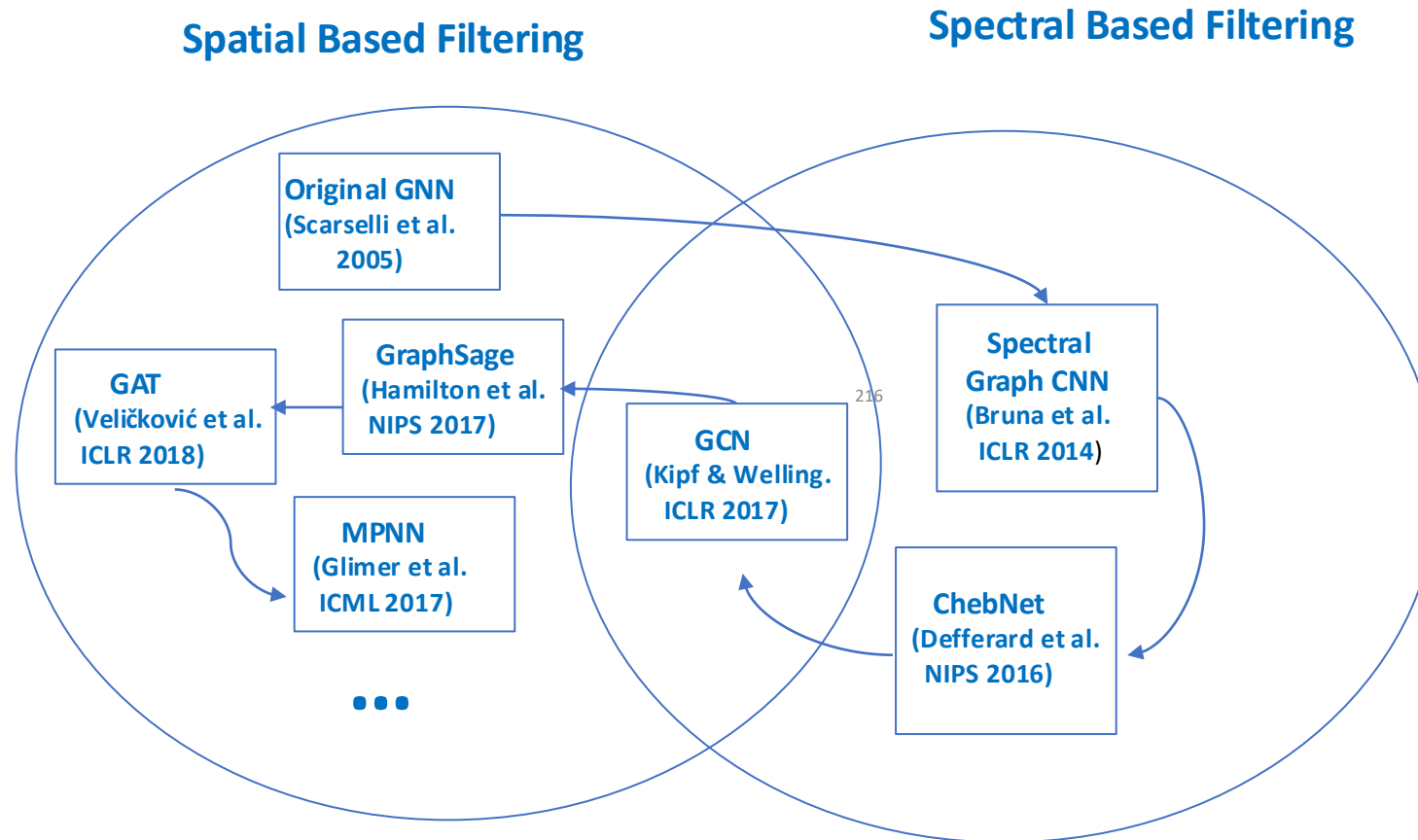


Implementation resources:

[PyG](#) provides core modules for this pipeline

[GraphGym](#) further implements the full pipeline to facilitate GNN design

Spectral and Spatial GNN Framework

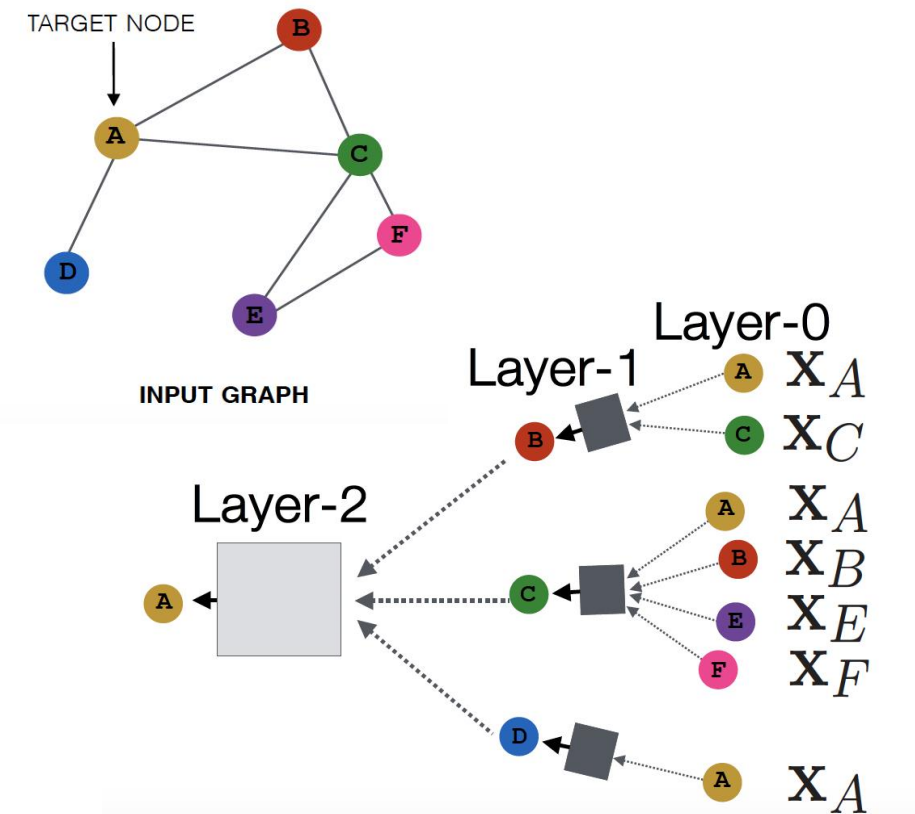
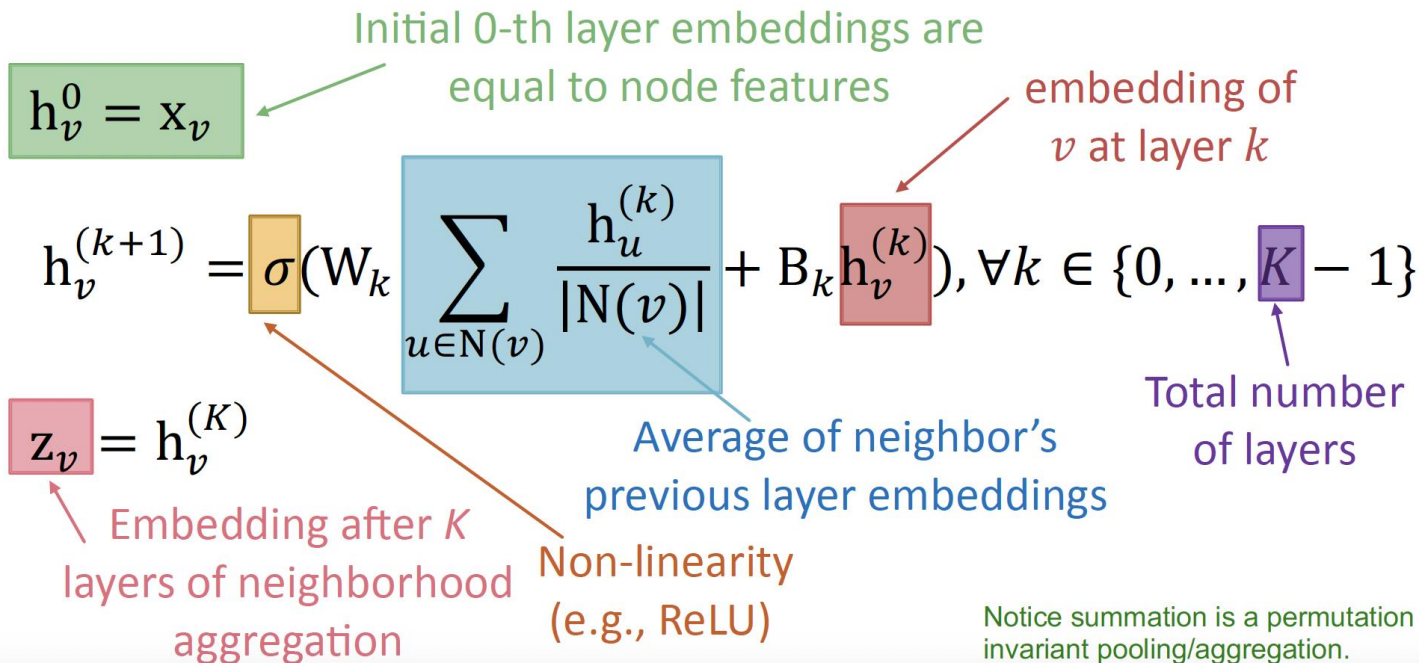


Spatial GNN Framework

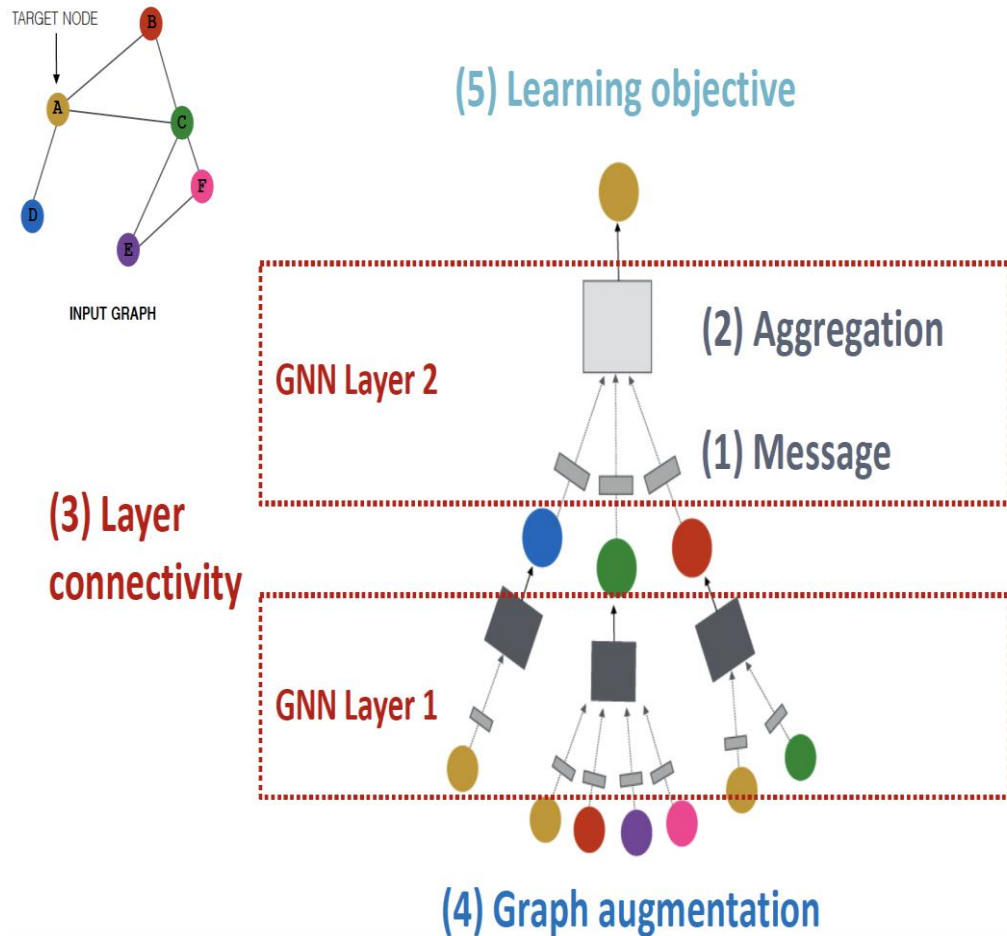
Key Concepts:

- ❖ Spatial approaches define convolutions directly on the graph using graph topology.
- ❖ Unlike spectral methods, these approaches operate in the node domain without eigen-decomposition.
- ❖ The challenge lies in handling variable neighborhood sizes and preserving local invariance.

General Spatial Convolution:



Neural Message Passing



The defining feature of a GNN is that it uses a form of *neural message passing*.

During each iteration k , a hidden embedding $h_u^{(k)}$ for node u is updated according to the information **aggregated from its neighborhood $N(u)$** , which can be expressed as follows:

$$h_u^{(k+1)} = \text{update}^{(k)} \left(h_u^{(k)}, \text{aggregate}^{(k)} \left(\{ h_v^{(k)}, \forall v \in N(u) \} \right) \right)$$

We often denote $m_{N(u)} = \text{aggregate}^{(k)} \left(\{ h_v^{(k)}, \forall v \in N(u) \} \right)$ as the “**message**” aggregated from neighborhood. The initial embeddings at $k = 0$ are set to the input features for all nodes, i.e., $h_u^{(0)} = x_u$. After running K iterations of the GNN message passing, we can use the **output of the final layer to define the embeddings for each node**, i.e., $z_u = h_u^{(K)}, \forall u \in V$.

Neural Message Passing: Intuition

Intuition Behind Message-Passing Framework

- ❖ The core idea of message passing is simple:
 - At each iteration, every node aggregates information from its 1-hop neighbors.
 - As iterations progress, nodes encode information from progressively farther regions of the graph.
- ❖ This allows nodes to capture both local and global structures over time.

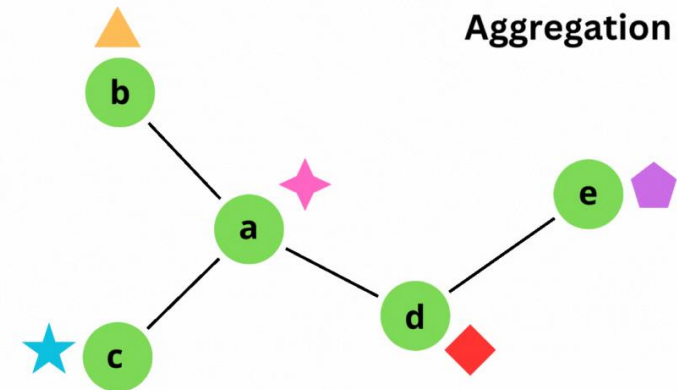
What Do Node Embeddings Encode?

Node embeddings contain two main types of information:

- **Structural Information:** Local connectivity patterns; Higher-order graph structures; the importance of a node based on its graph position (e.g., centrality measures).
- **Feature Information:** Numerical attributes (e.g., temperature, population density in spatial graphs); Categorical attributes (e.g., user preferences in recommendation systems); Learned representations from deep neural networks.

Why is Message Passing Powerful?

- ❖ Combines local and global information efficiently.
- ❖ Enables deep learning models to capture rich relational patterns.
- ❖ Supports various tasks like node classification, link prediction, and graph generation.



GNN: Basic Form

The basic GNN message passing is defined in **node-level**:

$$h_u^{(k)} = \sigma \left(W_{self}^{(k)} h_u^{(k-1)} + W_{neigh}^{(k)} \sum_{v \in N(u)} h_v^{(k-1)} + b^{(k)} \right)$$

where W_{self}, W_{neigh} are trainable parameter and σ denotes an elementwise non-linearity such as ReLU. Alternatively, it can also be succinctly defined in **graph-level**:

$$H^{(t)} = \sigma(H^{(k-1)} W_{self}^{(k)} + A H^{(k-1)} W_{neigh}^{(k)})$$

The basic GNN message passing can be simplified by omitting the explicit update step:

$$h_u^{(k+1)} = aggregate \left(\{ h_v^{(k)}, \forall v \in N(u) \cup \{u\} \} \right)$$

where now the aggregation is also taken over the node u itself. Adding self-loops is equivalent to sharing parameters between self and neighbor transformations.

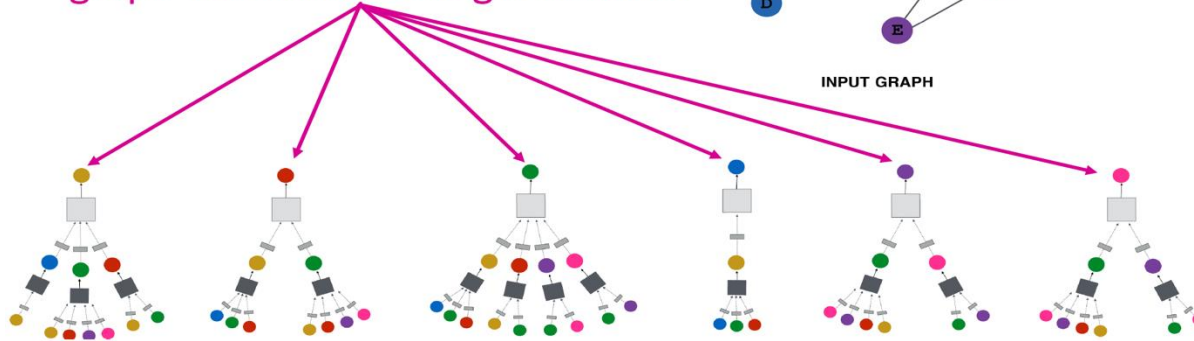
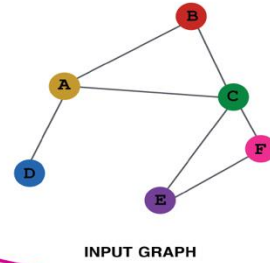
$$\mathbf{H}^{(t)} = \sigma \left((\mathbf{A} + \mathbf{I}) \mathbf{H}^{(t-1)} \mathbf{W}^{(t)} \right)$$

The self-loop GNN approach balances simplicity and efficiency but has some limitations. Self-loops make it harder to differentiate between node and neighbor information. Blurs the distinction between structural and feature information

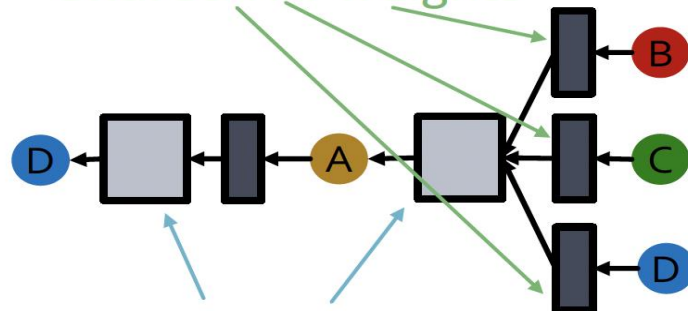
Permutation Invariant and Equivariant

- **Intuition:** Network neighborhood defines a computation graph

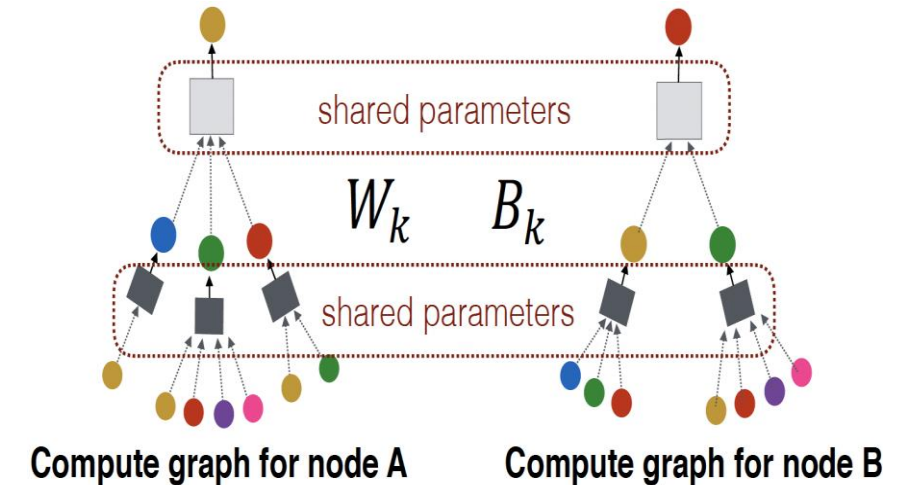
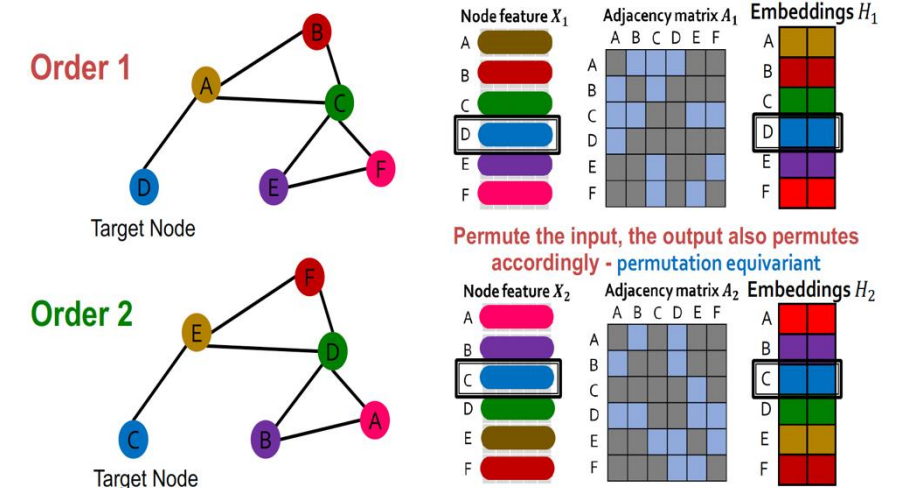
Every node defines a computation graph based on its neighborhood!



Shared NN weights



Average of neighbor's previous layer embeddings - **Permutation invariant**



Neighborhood Normalization

A basic approach is summing neighbor embeddings, but summing neighbor embeddings can create large magnitude differences. Nodes with significantly different degrees may lead to instability and optimization challenges.

$$h_u^{(k+1)} = \text{update}^{(k)} \left(h_u^{(k)}, m(N(u)) \right)$$

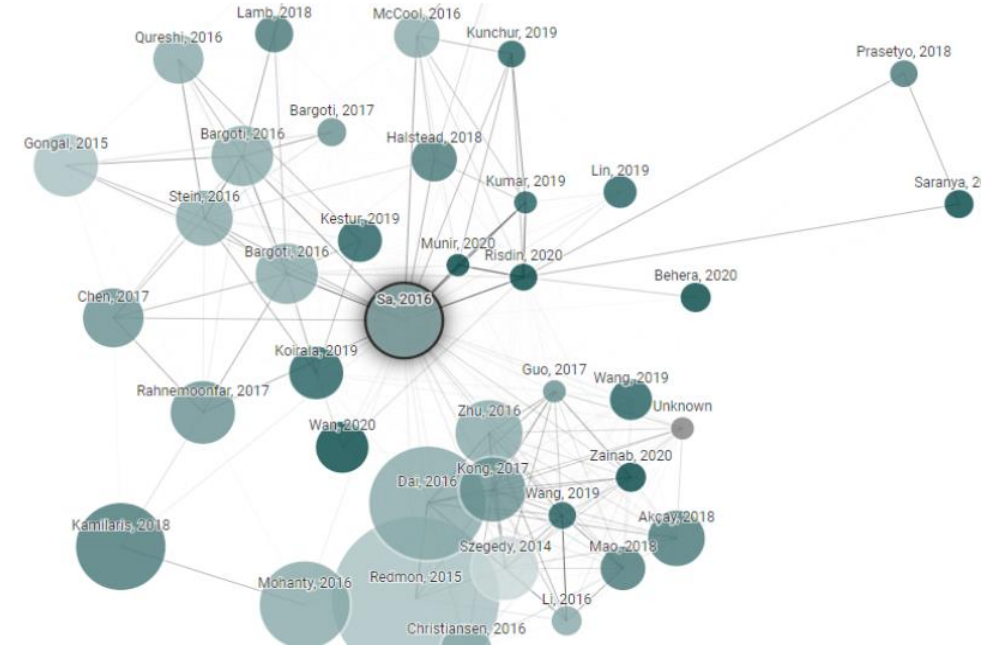
Example: A node with $100\times$ more neighbors than another will have drastically different embedding scales. Leads to numerical instability and difficulties in optimization.

A straightforward solution is **degree-based normalization**:

$$\mathbf{m}_{N(u)} = \frac{\sum_{v \in N(u)} \mathbf{h}_v}{|N(u)|}$$

One solution to this problem is to normalize based upon the degrees of the nodes involved, which is called *symmetric normalization*:

$$m_{N(u)} = \sum_{v \in N(u)} \frac{h_v}{\sqrt{|N(u)| \times |N(v)|}}$$



Graph convolutional networks (GCNs)

$$\mathbf{h}_u^{(k)} = \sigma \left(\mathbf{W}^{(k)} \sum_{v \in N(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|N(u)| |N(v)|}} \right)$$

Generalized Message Passing

As the last attempt to generalize the basic neural message passing framework, now we extend the approach beyond the node level, **leveraging edge and graph-level information** at each stage.

One more generalized message passing approach can be formulized according to the following equations:

$$\begin{aligned}h_{(u,v)}^{(k)} &= \text{update}_{\text{edge}} \left(h_{(u,v)}^{(k-1)}, h_u^{(k-1)}, h_v^{(k-1)}, h_G^{(k-1)} \right) \\m_{N(u)} &= \text{aggregate}_{\text{node}} \left(\{h_{(u,v)}^{(k)}, \forall v \in N(u)\} \right) \\h_u^{(k)} &= \text{update}_{\text{node}} \left(h_u^{(k-1)}, m_{N(u)}, h_G^{(k-1)} \right) \\h_G^{(k)} &= \text{update}_{\text{graph}} \left(h_G^{(k-1)}, \{h_u^{(k)}, \forall u \in V\}, \{h_{(u,v)}^{(k)}, \forall (u,v) \in E\} \right)\end{aligned}$$

The important innovation in this framework is that we generate hidden embeddings not only for each node $h_v^{(k)}$, but also $h_{(u,v)}^{(k)}$ for each edge in the graph as well as an embedding $h_G^{(k)}$ that corresponds to the entire graph. This allows the message passing model to easily integrate edge and graph-level features and have enhanced performances compared to a standard basic GNN. Generating embeddings for edges and the entire graph also makes it trivial to **define loss functions based on the graph or edge-level classification tasks**.

Deepset

A function f transforming a set $X = \{x_1, \dots, x_M\}$ into Y should be:

- ▶ **Permutation invariant:** The output does not change under reordering.

$$f(\{x_1, \dots, x_M\}) = f(\{x_{\pi(1)}, \dots, x_{\pi(M)}\})$$

for any permutation π .

- ▶ **Permutation equivariant:** The output follows the permutation.

$$f([x_{\pi(1)}, \dots, x_{\pi(M)}]) = [f_{\pi(1)}(x), \dots, f_{\pi(M)}(x)]$$

Theorem. A function $f(X)$ is invariant to the permutation of instances in X if and only if it can be decomposed as:

$$f(X) = \rho \left(\sum_{x \in X} \phi(x) \right)$$

where ϕ and ρ are suitable transformations.

The standard neural network layer is represented as:

$$f_{\Theta}(\mathbf{x}) = \sigma(\Theta \mathbf{x})$$

where $\Theta \in \mathbb{R}^{M \times M}$ is the weight matrix.

Theorem. A function $f_{\Theta} : \mathbb{R}^M \rightarrow \mathbb{R}^M$ is permutation equivariant if:

$$\Theta = \lambda I + \gamma(11^T)$$

where:

- ▶ I is the identity matrix,
- ▶ $1 = [1, \dots, 1]^T$,
- ▶ $\lambda, \gamma \in \mathbb{R}$.

de Finetti's theorem states that any exchangeable model can be factored as

$$p(X|\alpha, M_0) = \int d\theta \left[\prod_{m=1}^M p(x_m|\theta) \right] p(\theta|\alpha, M_0).$$

where θ is a latent feature and α, M_0 are hyper-parameters of the prior.

For Exponential Family with Conjugate Priors:

$$p(X|\alpha, M_0) = \exp \left(h \left(\alpha + \sum_m \phi(x_m), M_0 + M \right) - h(\alpha, M_0) \right)$$

PINE

Let $\mathbf{f}$ be a continuous real-valued function defined on a compact set with the following form

$$\mathbf{f}\left(\underbrace{\mathbf{x}_{1,1}, \mathbf{x}_{1,2}, \dots, \mathbf{x}_{1,N_1}}_{G_1}, \underbrace{\mathbf{x}_{2,1}, \dots, \mathbf{x}_{2,N_2}}_{G_2}, \dots, \underbrace{\mathbf{x}_{K,1}, \dots, \mathbf{x}_{K,N_K}}_{G_K}\right),$$

where $\mathbf{x}_{k,n} \in \mathbb{R}^{M_k}$. If function $\mathbf{f}$ is partial permutation invariant, then the PINE framework provides a

Core Representation Theorem as

$$\mathbf{f}(\cdot) = \mathbf{h}\left(\sum_{n=1}^{N_1} \mathbf{g}_1(\mathbf{x}_{1,n}), \sum_{n=1}^{N_2} \mathbf{g}_2(\mathbf{x}_{2,n}), \dots, \sum_{n=1}^{N_K} \mathbf{g}_K(\mathbf{x}_{K,n})\right) + o(1)$$

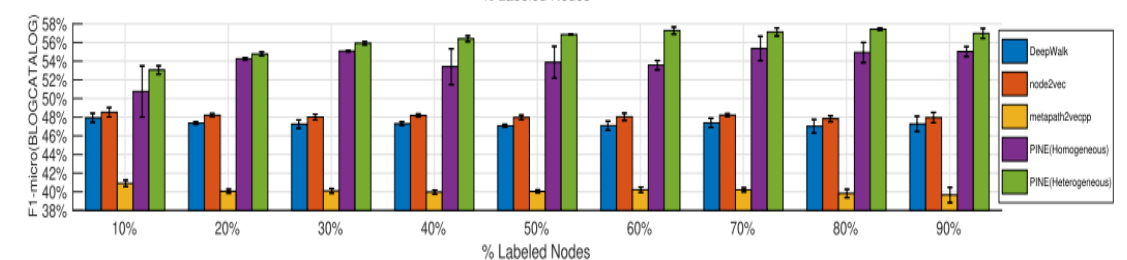
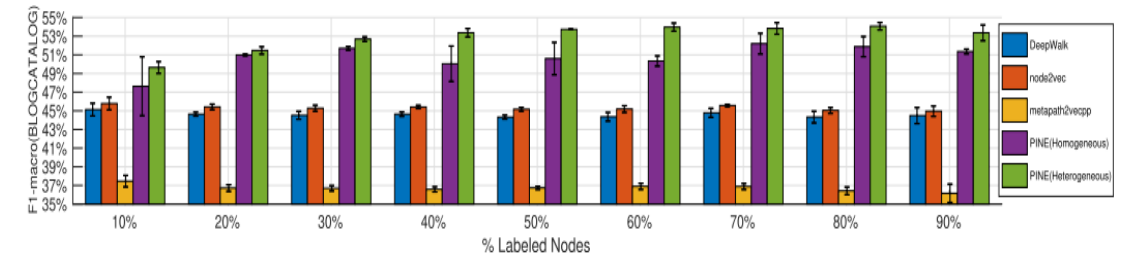
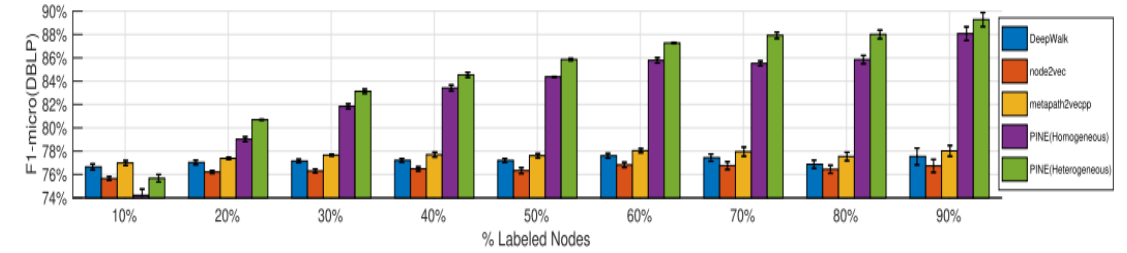
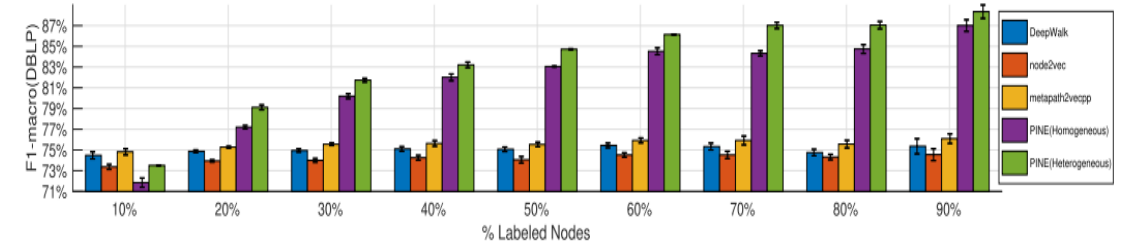
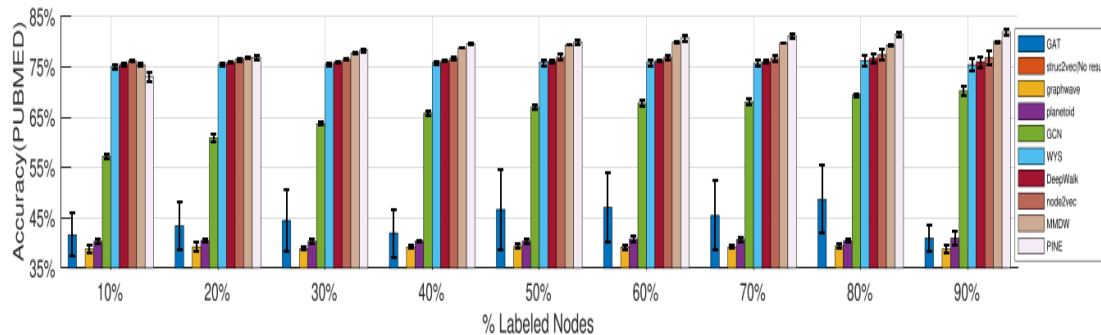
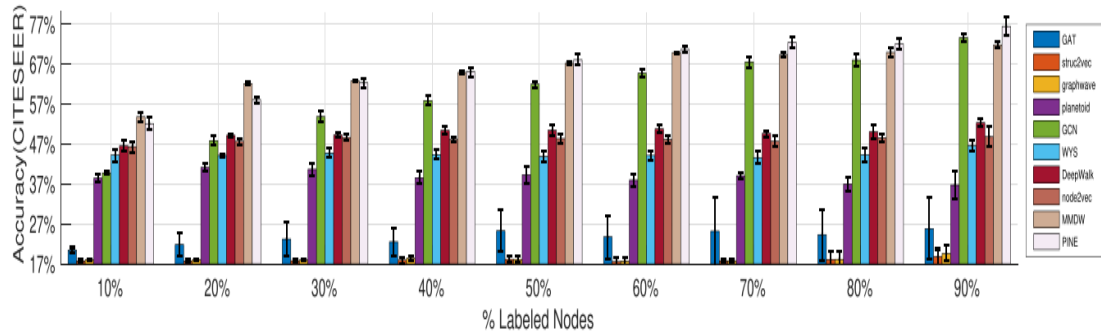
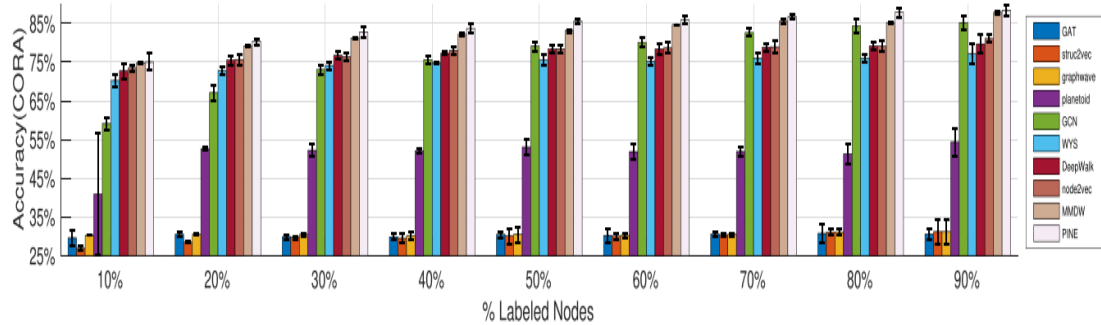
which requires $h(\cdot)$ and $g(\cdot)$ to ensure permutation invariant :

Then, PINE provides specific parameters for $h(\cdot)$ and $g(\cdot)$, which can be trained as follows:

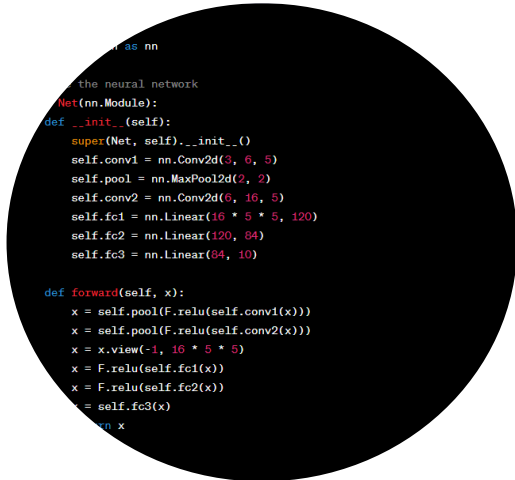
$$\mathbf{h}\left([\mathbf{z}_1^\top, \dots, \mathbf{z}_K^\top]^\top =: \bar{\mathbf{z}} \mid \mathbf{c}, W, \mathbf{b}\right) = \mathbf{c}^\top \sigma(W\bar{\mathbf{z}} + \mathbf{b}) \quad \mathbf{g}\left(\mathbf{x} \mid T, u, \{\mathbf{a}_t, \mathbf{b}'_t\}_{t=1}^T\right) = \begin{bmatrix} \sigma((\mathbf{u} \otimes \mathbf{a}_1)\mathbf{x} + \mathbf{b}'_1) \\ \sigma((\mathbf{u} \otimes \mathbf{a}_2)\mathbf{x} + \mathbf{b}'_2) \\ \vdots \\ \sigma((\mathbf{u} \otimes \mathbf{a}_T)\mathbf{x} + \mathbf{b}'_T) \end{bmatrix}$$

Evaluation

Accuracy (%) of multi-class classification in homogeneous and heterogeneous graphs



How to succeed in this course?



Practice



Explore



Visualize

Discuss



Ask

