APS360 Summary Notes

Ashley Leal

Summer 2023

0 Contents

1 General Terminology

Idea

- Introduction to AI terminology and approaches: **symbolic** and **connectionist**.
- **Symbolic Approach**: Logic-based problem-solving with strengths in transparency and reasoning, but struggles with complexity and contextual understanding.
- **Connectionist Approach (Neural Networks)**: Brain-inspired networks for pattern recognition with adaptability, but challenges in interpretability and overfitting.
- AI covers fields like **ML**, **CV**, **NLP**.
- Machine learning **learns from data** without explicit programming.
- Deep learning (DL) is a **subset of ML with multi-level abstractions**.
- DL **excels** in translation, image generation, gaming; faces **challenges** in interpretability and bias.
- Learning categories: **supervised**, **unsupervised**, **reinforcement**.
- **Bias-Variance Tradeoff** balances model complexity for better generalization.
- Splitting datasets for **training, validation, testing** aids model performance and selection.

1.1 Artificial Intelligence

- AI was coined in 1956 with the hope to **reproduce human intelligence** with machines.
- It captures the notion of developing computer systems that can perform tasks normally only humans can.
- AI is a **poorly defined** term and its meaning has changed with time
- Historically, there have been two distinct approaches to AI:

1. **Symbolic Approach**

Definition

The **symbolic** approach to AI is based on the idea that intelligent behavior can be achieved by **manipulating symbols or abstract representations of concepts and their relationships**. It draws inspiration from **formal logic** and relies on **rules, logic, and symbolic representation of knowledge** to solve problems. In this approach, knowledge is typically represented using symbols, such as predicates, rules, and frames, and reasoning involves manipulating these symbols using logical inference.

Figure 1.1: Symbolic Approach

(a) **Key Characteristics**:

i. **"Model the knowledge of an adult"**

ii. **Symbol Manipulation**: The approach focuses on manipulating symbols according to predefined rules and logic.

iii. **Knowledge Representation**: Information is represented using explicit symbols, which can represent concepts, relationships, and facts.
iv. **Rule-Based Systems**: AI systems in this approach often use rule-based systems to infer conclusions from given premises.
v. **Logic and Inference**: Logical reasoning is a core component of this approach, and it's used to derive new information from existing knowledge.
vi. **Expert Systems**: Early AI applications like expert systems used the symbolic approach, where human expertise was encoded in the form of rules and facts.

(b) **Strengths**:
i. **Transparency**: Symbolic systems provide a clear and interpretable representation of knowledge and reasoning processes.
ii. **Logical Reasoning**: Well-suited for problems that require logical inference and deductive reasoning.
iii. **Human-Readable**: The representations used are often human-readable and understandable.

(c) **Weaknesses**:
i. **Complexity**: Symbolic approaches can struggle with managing and representing the complexity of real-world domains.
ii. **Knowledge Acquisition**: Constructing a comprehensive set of rules and knowledge for complex domains can be challenging.
iii. **Lack of Contextual Understanding**: Symbolic systems may struggle with tasks that require understanding context and handling ambiguous or incomplete information.

2. **Connectionist Approach**

> **Definition**
>
> The **connectionist** approach, also known as the **neural network** approach, draws inspiration from the **structure and function of the human brain**. It involves building **artificial neural networks** that consist of **interconnected nodes (neurons)** that process information in a parallel and distributed manner. These networks learn patterns and relationships from data through **training processes** that adjust the strengths of connections (weights) between neurons.

Figure 1.2: Connectionist Approach

(a) **Key Characteristics**:
i. **"Simulate the learning of a baby"**
ii. **Parallel Distributed Processing**: Information processing occurs in parallel across interconnected nodes, leading to emergent properties.
iii. **Learning from Data**: Neural networks learn patterns and relationships from large datasets through processes like supervised learning, unsupervised learning, and reinforcement learning.
iv. **Feature Extraction**: Neural networks can automatically learn relevant features from raw data, reducing the need for hand-crafted feature engineering.
v. **Nonlinear Mapping**: Neural networks can capture complex nonlinear relationships in data.

(b) **Strengths**:
i. **Pattern Recognition**: Excellent at tasks such as image recognition, natural language processing, and other tasks involving pattern recognition.
ii. **Adaptability**: Neural networks can learn from data and adapt to different tasks with appropriate training.
iii. **Robustness**: Neural networks can generalize well to new data, even in the presence of noise or variability.

(c) **Weaknesses**:

i. **Black Box Nature**: Neural networks can be challenging to interpret, and understanding their decision-making processes can be difficult.

ii. **Data Dependency**: Neural networks require substantial amounts of data for effective training.

iii. **Overfitting**: There's a risk of overfitting to training data if not properly regularized.

AI researchers tend to avoid the term as much as possible, and talk about much more specific, relatively well-defined fields:

- **Machine Learning (ML)**: developing algorithms that learn from data rather than being hard-coded to solve a task
- **Computer Vision (CV)**: making computers capable of seeing
- **(NLP)**: making computers capable of understanding our languages

Until recently, these fields were almost completely separate. All of these are now dominated by Deep Learning.

Machine Learning enables computers to **learn** from data, **rather than being explicitly programmed**, to solve a task. ML is a very broad term. There are many methods of learning from data. We need machine learning because almost every rule will have some counter-example in the real world. This makes it difficult to cover all conditions in an algorithm. High-dimensional input space, like colored images are hard to understand, and require us to first learn easier representations. We require a way to learn from a lot of examples to solve a problem.

General Machine Learning Approach:

1. **Identify information** we want to find
2. **Collect data** that will hold that information
3. Design and run **algorithms to extract** as much information possible from data

Idea

"A computer program is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**." (Mitchell et al. 1997)

1.2 Deep Learning

Deep Learning is the latest version of **Artificial Neural Networks (ANN)**, or **Connectionism (an old ML method)**. Neural networks were inspired by the **brain**.

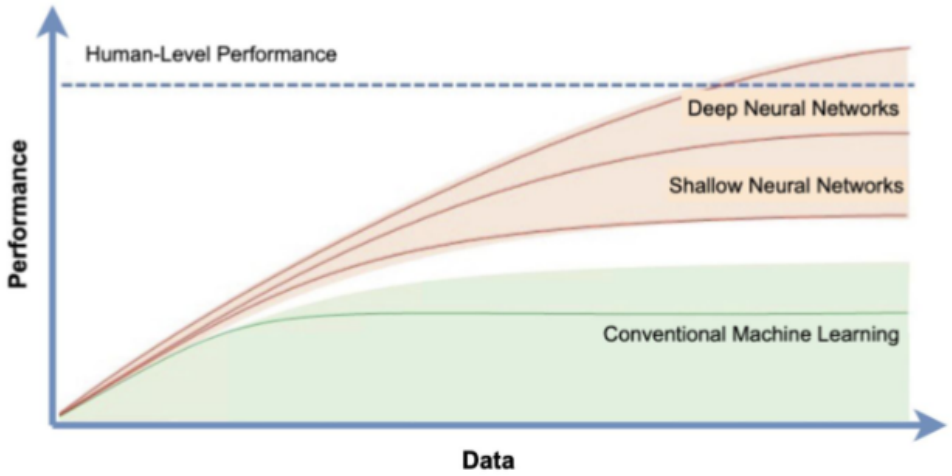


Figure 1.3: Deep Learning

Definition

"**Deep learning** is a subset of machine learning that allows **multiple levels of representation**, obtained by composing simple but **non-linear modules** that each transform the representation at one level **(starting with the raw input)** into a representation at a higher, slightly more **abstract level**." (LeCun et al. 2015)

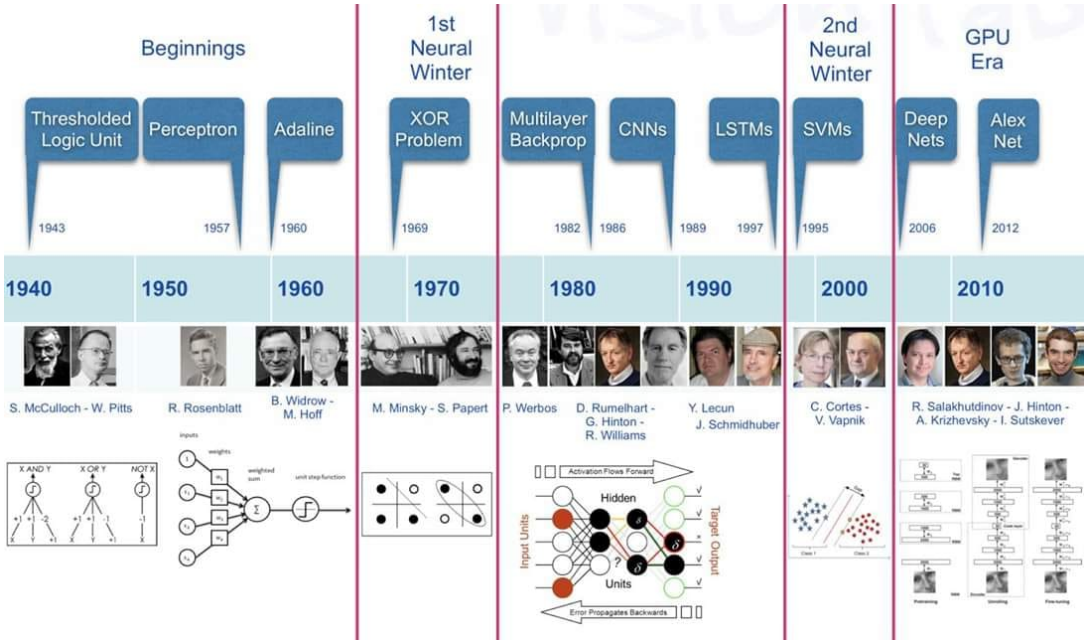


Figure 1.4: History of Deep Learning

Definition

Artificial Intelligence (AI): broad and poorly defined concept of developing computer systems that can **perform tasks normally only humans could do**.

Definition

Machine Learning (ML): computers **learn by example, from data**, rather than being explicitly programmed, to solve a task.

Definition

Deep Learning (DL): a machine learning method that learns **multiple levels of abstractions** over data end-to-end.

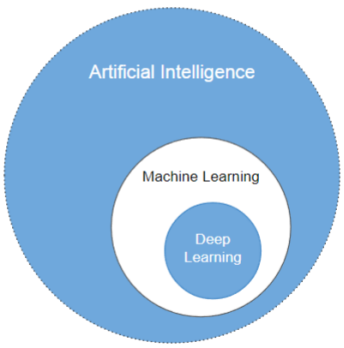


Figure 1.5: Artificial Intelligence vs. Machine Learning vs. Deep Learning

Deep Learning: Successes

- Machine Translation (Google Translate)
- Drug Discovery (antibiotics)
- Speech Recognition (auto-generated subtitles)
- Image Generation (generate image from prompts)
- AlphaFold (protein structures)
- AlphaGo
- Mathematics (pattern discovery)
- Code Generation
- Language Modelling
- Simulators

Deep Learning: Caveats

- Interpretability (black box)
- Adversarial Examples (noise filters over images)
- Causality (while deep learning models excel at identifying patterns in data, they fall short in distinguishing between correlation and causation)
- Fairness and Bias

Definition

Bias: refers to **systematic and consistent errors or inaccuracies in the predictions or outcomes** produced by a machine learning model. These biases can arise from **various sources** and can significantly **impact the model's performance and fairness**. Bias in deep learning can **occur at different stages** of the model development process, including **data collection**, **preprocessing**, **algorithm design**, and **decision-making**.

Example

A 2016 arXiv paper, claimed to be able to predict whether someone was a convicted criminal or not solely from a driver's license-style photo with 90% accuracy. When looked at in detail, images of criminals were collected from government IDs, while non-criminal face images were collected from the internet.

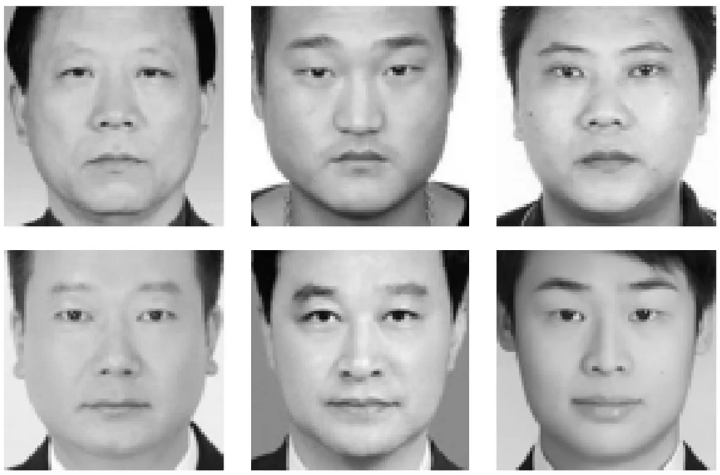


Figure 1.6: Top: ”criminal” images. Bottom: ”non-criminal” images.

1.3 Machine Learning Basics

There are three categories that describe how a Neural Network ”learns”:

1. **Supervised Learning**

Definition

Supervised Learning: training by feeding **labeled data** to the computer, leading it to **predict the correct label** for inputted data (relationship between input and label)

- Involves two main tasks:
  (a) **Regression**: the model predicts a continuous or real-valued output
  (b) **Classification**: the model assigns inputs to specific categories or classes
- Performance is often measured using metrics like **mean squared error** (for regression) or **accuracy, precision, and recall (true positive rate)** (for classification)
- Requires data with **ground-truth labels**

Definition

Mean Squared Error (MSE): metric used for measuring the **average squared difference** between the **predicted values** and the **actual (true) values** in a regression problem. It provides a measure of how well a regression model's predictions align with the **ground truth values**. The lower the MSE value, the better the model's predictions match the data points.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where:

- n is the number of data points (samples) in the dataset.
- y_i represents the actual target value (ground truth) for the ith data point.
- $\hat{y}_i$ represents the predicted value for the ith data point by the regression model.

Definition

Accuracy: metric that measures the **proportion of correct predictions** made by a classification model/ It gives an overall view of the model's correctness across all classes.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

Warning

Accuracy can be misleading in cases where **classes are imbalanced**. If one class is significantly more prevalent than others, a model might achieve high accuracy by simply predicting the majority class for all instances.

Definition

Precision: metric that measures the **proportion of positive predictions** that were actually correct. It measures the model's **ability to avoid false positives** (instances tat were predicted as positive but are actually negative).

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives + False Positives}}$$

Definition

Recall (True Positive Rate): metric that quantifies the **proportion of actual positive instances** that were correctly predicted by the model. It helps identify the model's ability to **capture all instances of the positive class**, minimizing false negatives (instances that are actually positive but are predicted as negative).

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives + False Negatives}}$$

Example

Fitting a polynomial (Regression): Given a noise sample data (blue), we want to find the polynomial that generated the data. However, many possible curves fit. Which of these curves is the best fit and why? This is where knowledge of the problem, **domain knowledge**, can be used to select an appropriate modelling technique.

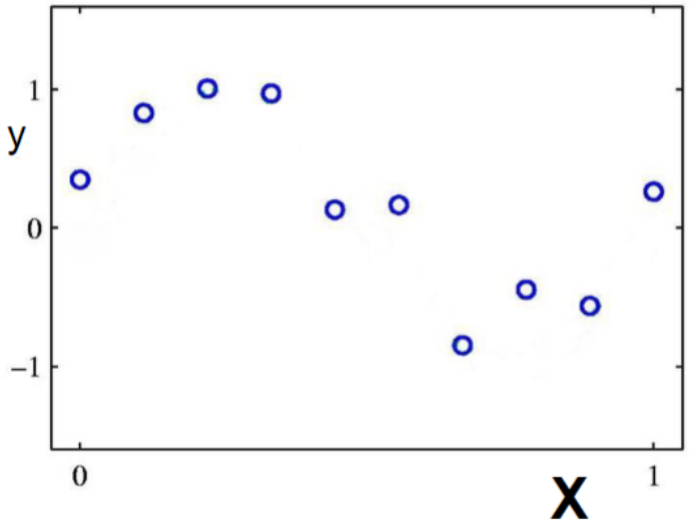


Figure 1.7: Noisy Sample Data

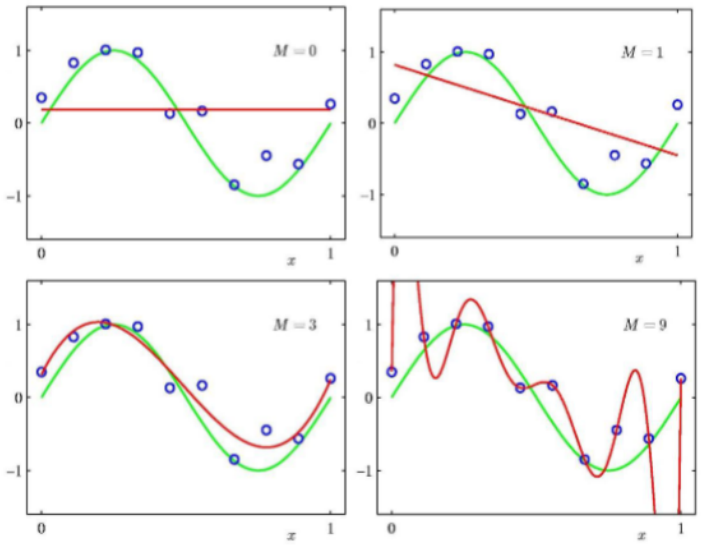


Figure 1.8: Possible Curves

Definition

Inductive Bias (Learning Bias): set of **assumptions** used for modelling, or prior knowledge that a learning model **uses to generalize** from the training data to new, unseen data. It's the foundational principles and constraints that guide a model's learning process and influence the types of patterns it's more likely to learn. Applies to all types of learning.

Theorem

No Free Lunch Theorem: concept that **there is no universal optimization** of learning algorithm that performs best for all possible problems. There is **no one-size-fits-all approach** that can excel across all domains and problem instances.

2. **Unsupervised Learning**

Definition

Unsupervised Learning: training by feeding **unlabeled data**; hence, relationships are made between **elements of input** (Clustering, grouping algorithm)

- Requires observations **without human annotations**
- Also known as Self-supervised Learning or Semi-supervised Learning

3. **Reinforcement Learning**

Definition

Reinforcement Learning: when we want to train a computer to perform a task but **we don't know the optimized way** so the computer **finds the best steps by itself** (we rank the computer's methods relative to each other and this way the computer learns the best method to perform a task)

- Has an **agent**, which is an entity capable of perceiving its environment, making decisions, and taking actions in order to achieve specific goals. This may be a software program, a robot equipped with sensors, or any **system that interacts with its environment** to acheive desired outcomes.
- **Sparse rewards** from environment. For example, winning or losing can only occur at the end of a game, making it difficult for an agent to learn to specific actions that lead to victory.
- **Dynamic nature**: agent's actions directly influence the environment. **Feedback loop** where agent's actions influence the environment, which in turn affects the agent's future options and decisions.

Machine Learning is a game of balance, with the objective being to generalize training data and future data.

Definition

Bias: refers to the **error introduced by approximating a real-world problem**, which may be complex, with a simplified model. In the context of a machine learning model, bias is the **error due to overly simplistic assumptions in the learning algorithm**. If a model has high bias, it means that it doesn't capture the underlying patterns in the training data well, leading to systematic errors. In other words, it is "**underfitting**" the data.

Definition

Variance: refers to the model's **sensitivity to small fluctuations or changes** in the training data. A model with high variance tends to fit the training data very closely, even capturing noise or randomness in the data. This can lead to a situation where the model performs well on the training data but poorly on new, unseen data, because it **hasn't generalized well**. This is called "**overfitting**."

Theorem

Bias-Variance Tradeoff: as you increase the complexity of a model, you generally reduce bias but increase variance, and vice versa. Finding the right level of complexity that minimizes both bias and variance is crucial for building models that generalize well to new data.

- If you **increase model complexity** (more features, more layers in neural networks, etc.), the model becomes more flexible and capable of fitting the training data closely. However, it also becomes more sensitive to fluctuations and noise in the data, **increasing the risk of overfitting** (high variance).
- If you **decrease model complexity**, the model becomes more constrained and less likely to fit the training data perfectly. While it might generalize better to new data, it could struggle to capture complex patterns, **resulting in underfitting** (high bias).

Idea

Generally, **more data** leads to a better model.

We need to split our dataset so that it can be use to **train, validate, and test** the model. We need to ensure the model does not see the testing (holdout) data during the training process, as that leads to overfitting to the test data and poor generalization on new data. The purpose of validation data is for assessing performance and making informed decisions about model selection and hyper-parameter tuning.

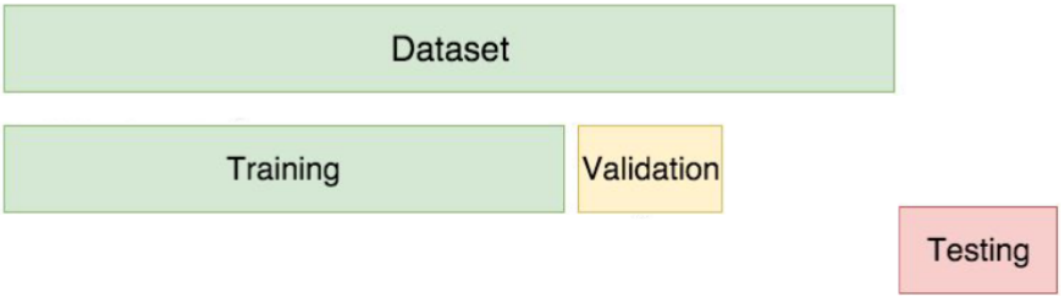


Figure 1.9: Dataset split into Training, Validation, and Testing Datasets

2 Artificial Neural Networks (ANNs)

Idea

- **Artificial Neural Networks (ANNs):** Computational models inspired by the brain's structure and function.
- **Neurons:** Processing units in ANNs that apply weights to inputs and pass through activation functions.
- **Activation Functions:** Introduce non-linearity to neuron outputs, enabling complex pattern capturing.
- **Loss Function:** Measures the discrepancy between predicted and actual outputs.
- **Training:** Process of adjusting weights to minimize the loss function.
- **Gradient Descent:** Optimization algorithm that updates weights in the direction of loss reduction.
- **Simple Fully Connected Network:** Neurons in one layer connected to all in the next; input passes through hidden layers to produce output.
- **Hyperparameters:** Settings chosen prior to training, such as learning rate, batch size, and network architecture.
- **Optimizers:** Algorithms that control weight updates during training (e.g., SGD, Adam).
- **Normalization:** Techniques like Batch Norm and Layer Norm stabilize training by normalizing input or activation distributions.
- **Regularization:** Techniques like Dropout and Weight Decay prevent overfitting by adding constraints or penalties.

2.1 Neuron

Simplified Biological Neuron:

- **Dendrites** recieve information from other neurons
- **Cell body** consolidates information from the dendrites
- **Axon** passes information to other neurons
- **Synapse** is the area where the axon of one neuron and the dendrite of another connect

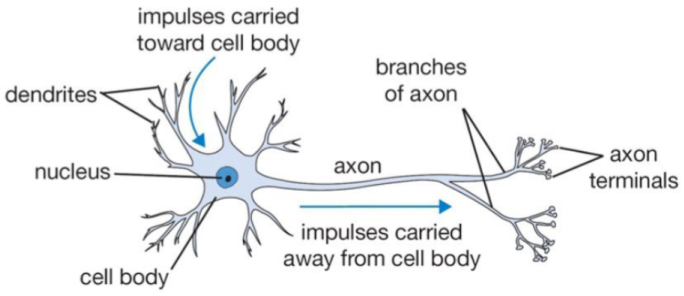


Figure 2.1: Simplified Biological Neuron

Neurons **fire** on stimuli: edges, lines, angles, movements, familiar faces, etc. regardless of scale, rotation and translation.

Artificial Neuron:

- $\mathbf{x}_i$ is the **input** such as a pixel in an image
- $\mathbf{w}_i$ is the **weight** for input $\mathbf{x}_i$ that we learn for this particular input
- $\mathbf{b}$ is the **bias**, a weight we learn with no input
- $\mathbf{f}$ is the **activation function** that determines how our output changes with the sum of all weight-input products

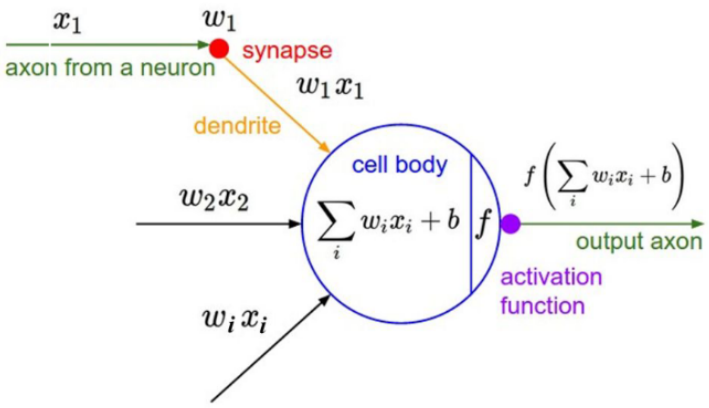


Figure 2.2: Mathematical Representation of Artificial Neuron

- y is the **output** such as the class an image belongs to
- Basically, we take the **weighted sum of all inputs**, **add the bias term**, then pass the result to the **activation function**. The result of the activation function is the **output**.

$$y = f(w \cdot x + b)$$

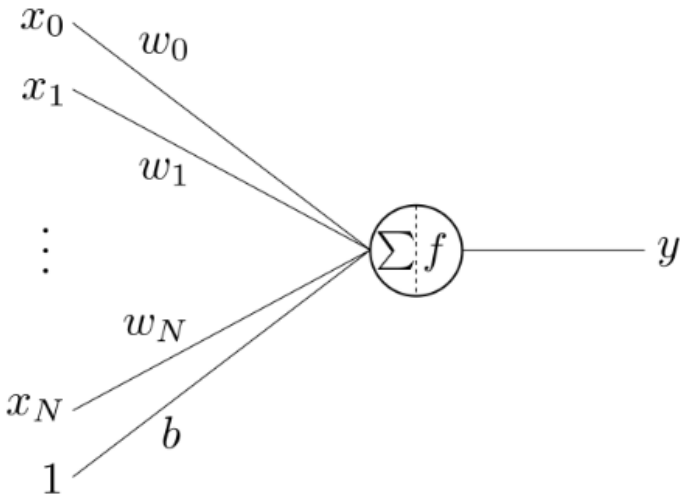


Figure 2.3: Vector Representation of Artificial Neuron

2.2 Activation Function

Linear Activation Function

Definition

Linear Activation Function: a mathematical function that produces an **output directly proportional to its input**. In other words, for any given input, the output increases or decreases at a constant rate without any bending or non-linear behavior.

- When the activation function is a simple linear function:

$$y = f(w \cdot x + b) \longrightarrow y = w \cdot x + b$$

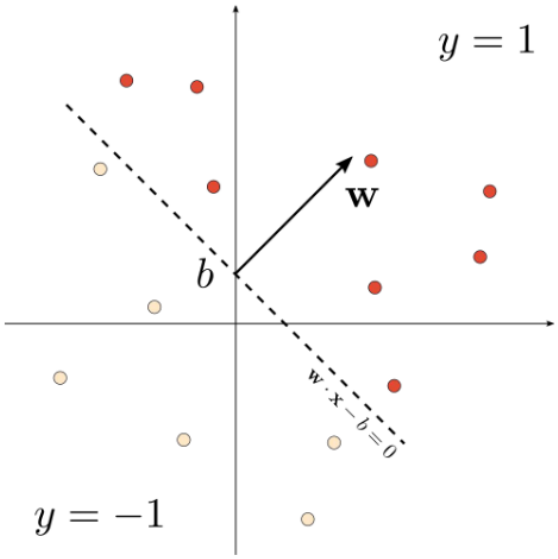


Figure 2.4: Linear Activation Function

- This is a special equation and becomes more clear if we write it out for 2D:

$$y = w_0 x_0 + w_1 x_1 + b$$

- Recall general equation of line:

$$Ax + By - C = 0$$

- The bias is related to the offset of the line from the origin and allows us to create a boundary between data that doesn't have to go through the origin

> **Definition**
>
> **Bias Term:** a **constant** value that's added to the weighted sum of inputs in a neuron or model, allowing the neuron to **shift the activation function's output** up or down, effectively **adjusting the decision boundary** of the neuron's response. The bias term is **learnable** during the training process, just like the weights associated with the inputs

- $y = w \cdot x + b$ is a generalized line for any dimension, known as **hyperplane**, splitting the n-dimensional input space into 2 parts
- Linear activation functions are **not usually practical** since most real datasets are not linearly separable (we can't usually find a straight line that separates classes well in a classification problem)
- We can learn **non-linear transformations** of our data to help

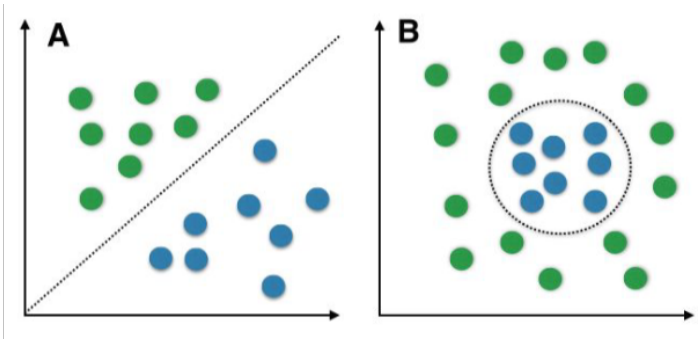


Figure 2.5: Most datasets are not linearly separable like A

- Multiple layers with non-linear transformations are useful, while **there is no advantage from multiple linear layers**, since the composite is a linear layer and reduces the network's ability to capture complex patterns in data

Perceptrons (Binary Activation Function)

Definition

Perceptron Activation Function: a decision rule that outputs **1 when the weighted sum of inputs exceeds a threshold, and outputs 0 otherwise**. This decision guides the activation of a perceptron in a basic neural network unit.

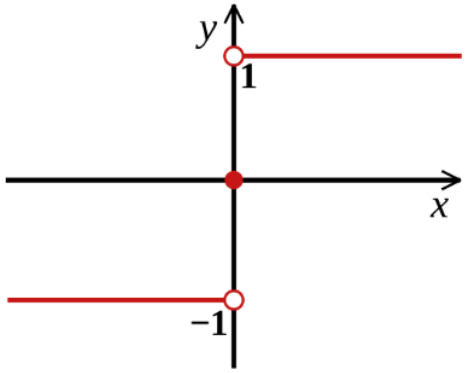


Figure 2.6: Perceptrons

- The first artificial neurons (1943-1970s) use a simple binary activation function based on which side of the hyperplane the input is
- Examples of these functions include the **Sign function** and the **Heaviside (unit) step function**

$$f(x) = sign(x)$$

$$f(x) = \begin{cases} 0, & \text{if } x < 0, \\ 1, & \text{if } x \geq 0. \end{cases}$$

- The **decision boundary** is the hyperplane, or the **threshold that separates the classes** in the binary classification problem.
- These functions are **not differentiable, contiuous or smooth**

Sigmoid Activation Function

Definition

Sigmoid Activation Function: a mathematical function that **smoothly** maps input values to a specified output range. It is characterized by an S-shaped curve and is commonly used in neural networks to **introduce non-linearity**, allowing the network to capture complex relationships in data. The sigmoid function is particularly useful for transforming raw scores into probabilities or for controlling the strength of neuron activations.

- Sigmoid activation functions were most common before 2012
- **Easily differentiable, smooth, continuous**
- Range between [-1, 1] or [0, 1]
- There are many sigmoid functions, the most common are **Hyperbolic tangent** and **Logistic functions**:

$$f(x) = \tanh(x)$$

$$f(x) = \frac{1}{1 + e^{-x}}$$

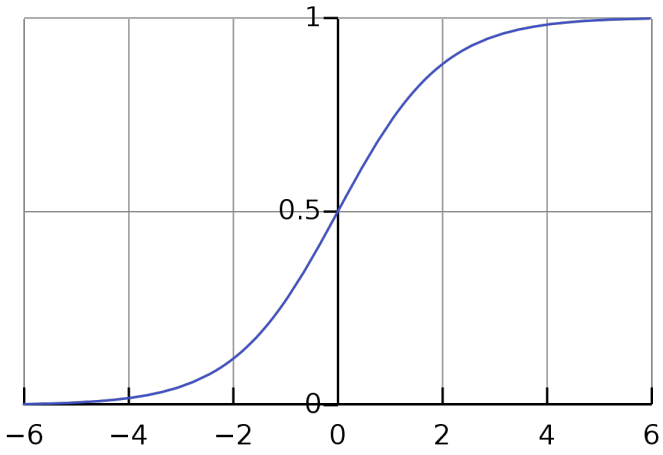


Figure 2.7: Sigmoid Function

- Saturated neurons ”kill” the gradients

Definition

Gradients: represent the directions and magnitudes by which the network's internal settings (weights and biases) should be adjusted to minimize the difference between its predictions and the actual target values. Gradients guide the network during training, helping it learn how to make better predictions by fine-tuning its parameters based on the errors it makes.

- Gradients become vanishingly small very quickly away from x = 0

Theorem

Vanishing Gradient Problem: pertains to the phenomenon wherein the **gradients of the loss function** with respect to the parameters of the earlier layers **become extremely small** as they are back-propagated through the network during the learning process. This reduction in gradient magnitude **diminishes the weight updates applied to the earlier layers** during optimization, causing these layers to **learn at a significantly slower rate or even stagnate** in terms of learning progress. As a consequence, the **network struggles to capture complex relationships** in the data, hindering its ability to generalize and make accurate predictions.

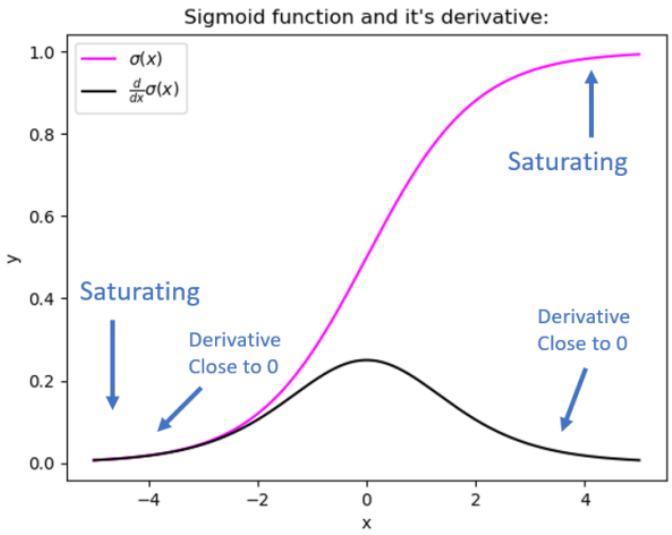


Figure 2.8: Vanishing Gradient Problem

ReLU Activation Function

Definition

ReLU Activation Function: turns **on for positive input values**, allowing signals to pass unchanged. For **negative inputs, it outputs zero**, which helps the network introduce non-linearity and learn complex patterns in data. ReLU helps with the vanishing gradient problem, though variations like Leaky ReLU address potential issues of inactive neurons during training.

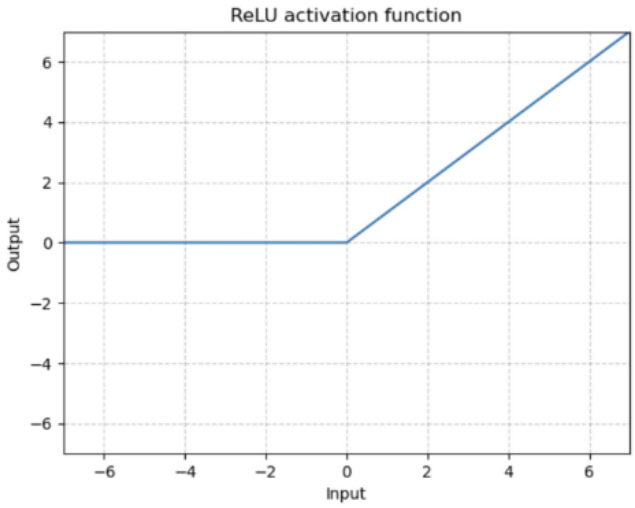


Figure 2.9: ReLU Activation Function

- Modern deep learning typically uses the **Rectified Linear Unit (ReLU)** based activation functions:
  - **ReLU**:

$$\text{ReLU}(x) = (x)^+ = \max(0, x)$$

  - **Leaky ReLU**:

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative\_slope} \cdot x, & \text{otherwise} \end{cases}$$

  - **Parametric ReLU:**

$$\text{PReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ a \cdot x, & \text{otherwise} \end{cases}$$

  - These functions have very **easy derivaties**: 0 or 1; use 0 at x = 0
- We can also approximate ReLU activation by **continuous functions**:
  - **SiLU (Swish)**:

$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

  - **SoftPlus**:

$$\text{SoftPlus}(x) = \frac{1}{\beta} \cdot \log(1 + e^{\beta x})$$

  - These work **on par or better** than ReLU functions due to mitigating issues like the **vanishing gradient problem**, dead neurons, lack of smoothness, and noise sensitivity that can arise with the original ReLU function. These approximations offer improved stability, controlled activation behavior, and better optimization dynamics for specific tasks and network architectures.

2.3 Training Neural Networks

How do we learn the **weights** and **bias** of a neural network?
Input: x, Predicted Output: y, Ground Truth Label: t, Neuron $M(w; x)$

1. **Prediction:** Make a prediction for some input data x, with a known correct output t:

$$y = M(w; x)$$

2. **Comparison:** Compare the correct output with our predicted output to compute loss:

$$E = \text{Loss}(y, t)$$

3. **Adjustment:** Adjust the weights/bias to make the prediction closer to the ground truth, i.e., minimize error.
4. **Repetiton:** Repeat until we have an acceptable level of error.

This process gradually improves a network's ability to solve a problem.

Definition

Forward Pass: input data is fed into a neural network, and it **flows through the layers**, undergoing computations using the network's parameters (weights and biases). This process generates predictions or output values based on the input data. Used for both **training and inference**.

Definition

Backward Pass: also known as **backpropagation**, this step follows the forward pass. It involves **calculating the gradients of the loss function** with respect to the network's parameters. These gradients indicate how much each parameter needs to be **adjusted** to reduce prediction errors. The gradients are then used to update the parameters using optimization algorithms. Used only for **training**.

The forward pass is the initial step that **produces predictions** based on input data. The backward pass is crucial for **training** the network. By computing gradients during the backward pass, the network's parameters are adjusted in a way that **reduces prediction errors**. This adjustment is **iteratively repeated** through multiple forward and backward passes, refining the network's parameters to make better predictions over time.

In summary, the **forward pass generates predictions**, while the **backward pass calculates gradients to fine-tune the network's parameters**, enabling it to learn and improve its performance.

Idea

All activation functions used in modern neural networks trained with backpropogation are both **differentiable** and **non-linear**.

2.4 Loss Function

Definition

Loss Function: computes **how bad predictions are** compared to the ground truth labels. Large loss means the network's predictions differ from the ground truth. Small loss means the network's prediction matches the ground truth.

We want to calculate the error over **all training samples** (to get the average error).

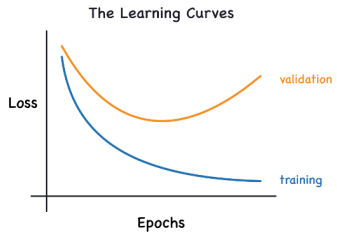


Figure 2.10: Learning Curves

Example

Suppose we want to train a linear neuron to differentiate images into three classes:

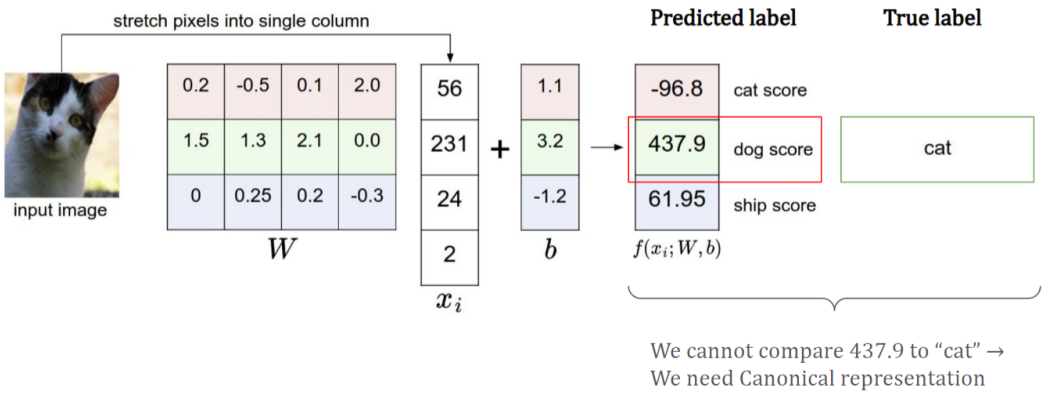


Figure 2.11: Why we need Softmax

We need a way that will allow us to **represent the output** in a way that can be **compared to the true label** so that we may see if the model's predictions are accurate or not. Enter: **Softmax**.

Definition

Softmax function: normalizes the logits into a categorical probability distribution over all possible classes. Produces normalized probabilites for multiple classes.

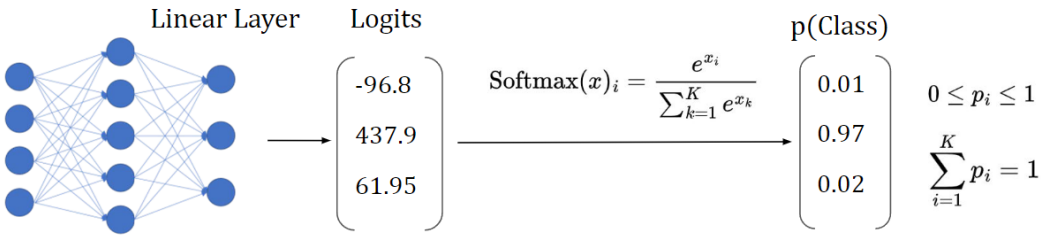


Figure 2.12: Softmax Function

We still need to map "cat" to a **vector** so that it can be compared with the softmax output. A way to map categories to vector representation is **One-hot encoding**.

Definition

One-hot encoding: a **binary representation** technique used to convert categorical variables into a **numerical format**. Each category is represented as a binary vector where a single element is set to **1** (indicating the **presence** of that category) while all other elements are set to **0** (indicating the **absence** of other categories).

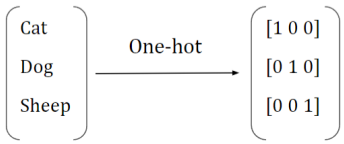


Figure 2.13: One-hot encoding

Now that the output of the model and the classes have been converted to comparable forms, how do we determine the accuracy of the model? We use **Loss functions**:

1. **Mean Squared Error (MSE)**: mostly used for **regression** problems (predicting continuous numeric output)

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_{\text{predicted},i} - y_{\text{actual},i})^2$$

Where:

n : Number of data points (training samples)
$y_{\text{predicted},i}$: Predicted value for the ith data point
$y_{\text{actual},i}$: Actual target value for the ith data point

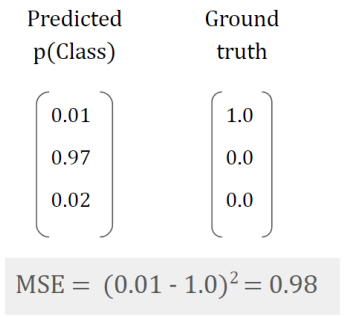


Figure 2.14: Mean Squared Error

2. **Cross Entropy (CE)**: mostly used for **multi-class classification** problems

$$CE = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_{\text{actual},n,k} \cdot \log(y_{\text{predicted},n,k})$$

Where:

N : Number of data points (training samples)
K : Number of classes
$y_{\text{actual},n,k}$: Actual target value (1 or 0) for the nth data point and kth class
$y_{\text{predicted},n,k}$: Predicted probability for the kth data point belonging to the kth class

3. **Binary Cross Entropy (BCE)**: mostly used for **binary classification** problems

$$BCE = -\frac{1}{N}\sum_{n=1}^{N}\left(y_{\text{actual},n} \cdot \log(y_{\text{predicted},n}) + (1 - y_{\text{actual},n}) \cdot \log(1 - y_{\text{predicted},n})\right)$$

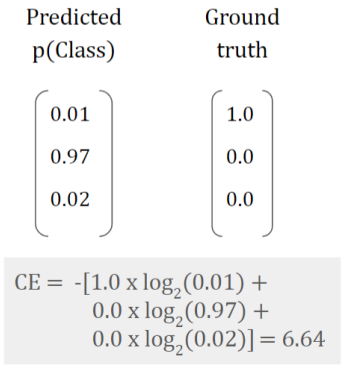


Figure 2.15: Cross Entropy Error

Where:

N : Number of data points
$y_{\text{actual},n}$: Actual target value (0 or 1) for the nth data point
$y_{\text{predicted},n}$: Predicted probability for the nth data point

2.5 Gradient Descent

Definition

Gradient Descent: an optimization algorithm used in machine learning to **iteratively** adjust the parameters of a model in the **direction that decreases the model's error**, guided by the **gradient** of the error with respect to those parameters.

A neural network layer with two neurons:

$$y_1 = f(w_1 \cdot x + b_1)$$
$$y_2 = f(w_2 \cdot x + b_2)$$

can represent a neural network easier with a **matrix** where each neuron's weight vector is a **row of the weight matrix W** and the input is a **column vector x**:

$$y = f(Wx + b)$$

This is relevant for debugging the dimensions of tensors.

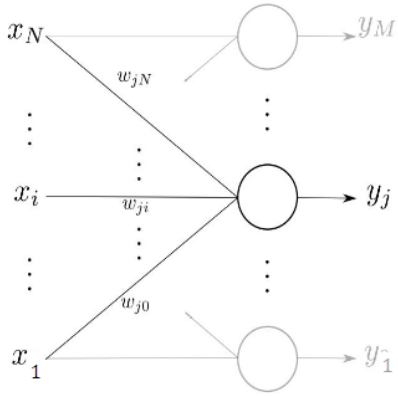


Figure 2.16: NN layer represented by vectors

In order to improve our network, we want to know how to change each of our neuron's weights $w_j i$ to **reduce error** E. Basically, we want to find the weights that are **increasing the error** and change them in the **opposite direction** of the current function to **correct** them. To do this, we will use **Gradient Descent**.

Idea

In the context of this course, **gradients and derivatives are the same thing**. The **vector** of partial derivatives for all weights is the **gradient**. The **direction** of the gradient is the **direction the function is increasing**. The **magnitude** of the gradient is the **rate of increase**.

First, we need to know how our error changes with **each individual weight**:

$$\frac{\partial E}{\partial w_{ji}}$$

This is relatively simple to calculate adjacent to the output layer. Adjusting weights according to the slope (gradient) will guide us to the minimum (or maximum) error. We do this by computing the **gradient of loss with respect to each weight**. That will tell us the direction in which the weight is increasing the loss.

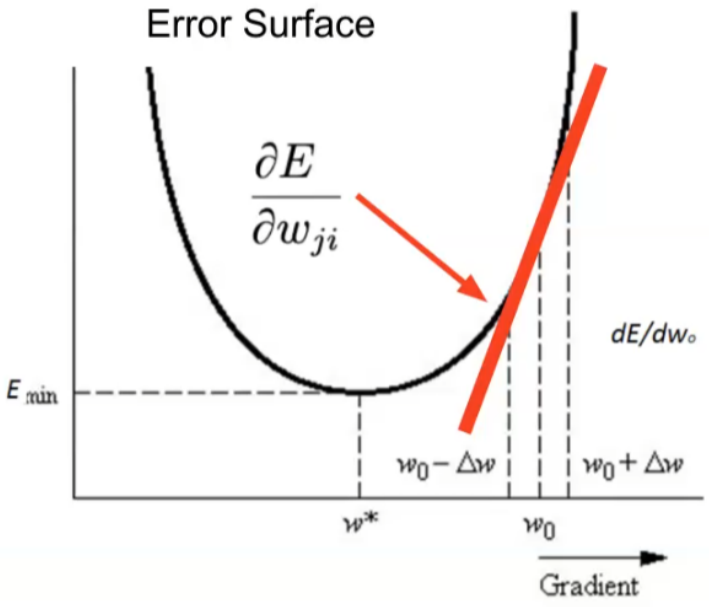


Figure 2.17: Gradient Descent. The graph represents the Error (loss) function and the red tangent line is gradient of loss.

Gradient Descent Formula: $w_{ji}^{t+1} = w_{ji}^{t} - \eta \frac{\partial E}{\partial w_{ji}}$

$$\Delta w_{ji} = \eta \frac{\partial E}{\partial w_{ji}}$$

Where η is the learning rate (step size).

Idea

A **positive** gradient at a weight means that the weight is **contributing** to the loss. This is why we negate it in the Gradient Descent Formula

Example

Delta Rule for Single Weight/Training Example

$$E = (y - t)^2 \text{ and } f(x) = \frac{1}{1 + e^{-x}}$$

To solve this, we need to use the **Chain Rule**:

$$\frac{dE}{dw_p} = (\frac{dE}{dy})(\frac{dy}{da})(\frac{da}{dw_p})$$

- $a = \sum_p w_p x_p + b$ (summation of inputs)
- $y = f(a)$
- $\frac{dE}{dw_p}$ represents the derivative of the error E with respect to the weight w_p.
- $\frac{dE}{dy}$ represents the derivative of the error E with respect to the output y.
- $\frac{dy}{da}$ represents the derivative of the output y with respect to the input a.
- $\frac{da}{dw_p}$ represents the derivative of the input a with respect to the weight w_p.

Let's calculate each of these derivatives:

$$\frac{dE}{dy} = (\frac{d(y - t)^2}{dy}) = 2(y - t)$$

$$\frac{dy}{da} = (\frac{d\frac{1}{1+e^{-a}}}{da}) = (1 - y)(y) \text{ (useful property of sigmoid functions)}$$

$$\frac{da}{dw_p} = x_p$$

$$\frac{dE}{dw_p} = 2(x_p)((y - t)((1 - y)(y)))$$

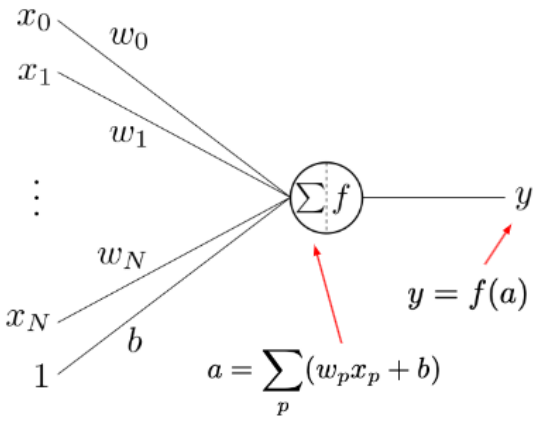


Figure 2.18: Delta Rule Example

2.6 Neural Network Architectures

XOR Problem

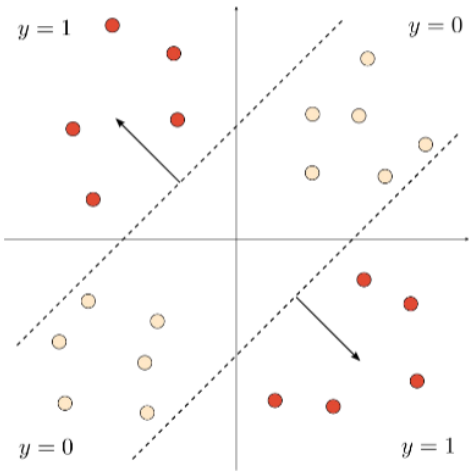


Figure 2.19: XOR Function

Having a single decision boundary (a single NN layer) is not enough to solve many problems. The most famous such problem is the **XOR function**, which **needs two decision boundaries** to solve (more than one line).

Definition

XOR Function: short for ”exclusive OR,” is a logical operation that takes in two inputs and produces an output. The output is **true (or 1) when the inputs are different from each other**, and **false (or 0) when the inputs are the same**. In other words, if one input is true and the other is false, the XOR function gives a true output. If both inputs are the same (both true or both false), the XOR function gives a false output.

We solve this by having **at least one hidden neural network layer**. In fact in the limit of an infinitely-wide neural network with at least one hidden layer, **NN is a universal function approximator**. This means that if this hidden layer is made extremely wide (with many neurons), and there's at least one hidden layer, the neural network becomes capable of **approximating almost any kind of function**.

Definition

Hidden Layer: a set of neurons that process information **between the input and output layers**, capturing complex patterns in the data. It's called ”hidden” because it's **not directly connected to the input or output** of the network.

Credit Assignment Problem

Neural networks up until the 1970s were not very useful for two main reasons:

1. Not clear how to train a NN of more than 1 layer (known as the **credit assignment problem**)

2. A neural network of only **one layer cannot describe complex functions**, two or more can represent any function (in theory with infinite width).

Definition

Credit Assignment Problem: the challenge of figuring out **how much each neuron in a network contributed to the final prediction or output**. It's about understanding which parts of the network were most **responsible for the outcome** and how changes in **individual neurons affect the overall performance**.

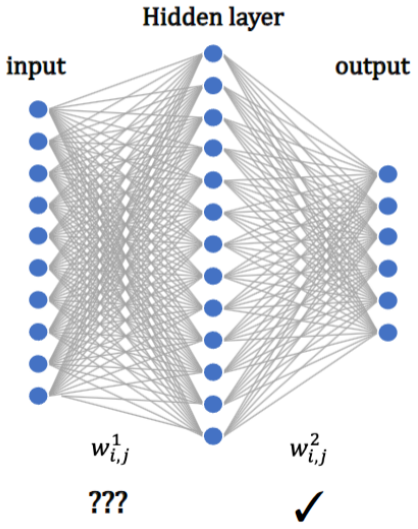


Figure 2.20: Backpropogation

Idea

Solving the credit assignment problem is essential for training neural networks effectively, as it allows us to identify which parts of the network **need adjustment or improvement**. Without a clear understanding of credit assignment, it's challenging to fine-tune networks, diagnose issues, and ensure consistent learning and performance improvement.

The credit assignment problem was solved by **backpropagation**, a method that describes how to distribute errors to neurons **not adjacent to the output layer**.

Definition

Backpropagation: a method in training neural networks, where the **network adjusts its internal parameters (weights and biases) by computing the gradient of the loss function** with respect to these parameters. This allows the network to learn and improve its predictions over time.

Backpropagation uses a **dynamic programming**-like approach to calculate gradients and adjust weights in neural networks. The combination of **gradient descent** and **dynamic programming** is called backpropogation. It propagates error information **backward** through the layers, updating the weights in a way that **optimizes the network's performance over time.**

Multiple Layers with Non-Linearity

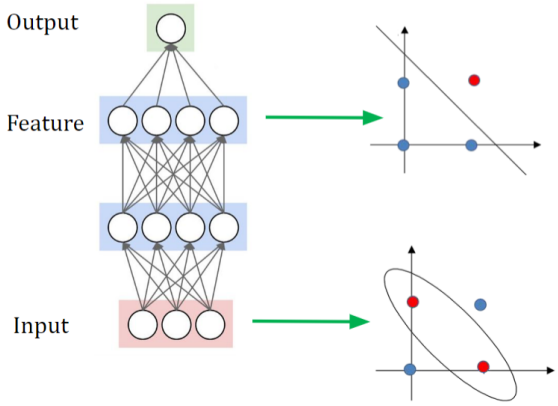


Figure 2.21: Multiple Layers with Non-Linearity

- Neural networks can be viewed as a way of learning features directly and end-to-end from raw input data.
- You can use the activations of the layer before the last layer as high-level features representing the input data.
- The goal being that the final layer is presented with a **linear separation**.

Idea

Essentially, the network **maps (or projects) input, which is not linearly separable, to other dimensions (spaces) where it does become linearly separable**. The output layer is a linear layer because by the time the data reaches it, **it is already linearly separable**. In these hidden layers, the network learns features in the input data.

Neural Network Architecture
An architecture of a neural network describes the neurons and their connectivity. Architecture selection will greatly affect model performance. It is an example of a very significant **inductive bias**.

Definition

Feed-Forward Network: information only flows forward from **one layer to a later layer**, from the input to the output **without any loops or cycles**. It's designed to make predictions or classifications based on the input data **without internal feedback connections**.

Definition

Fully-Connected Network: neurons between adjacent layers are **fully connected**. Each neuron in a given layer is connected to every neuron in the subsequent layer. This creates a **dense** and **interconnected structure** that enables **information to flow freely** between all layers.

Definition

Number of Layers: number of **hidden layers + output layer** (not including input "layer")

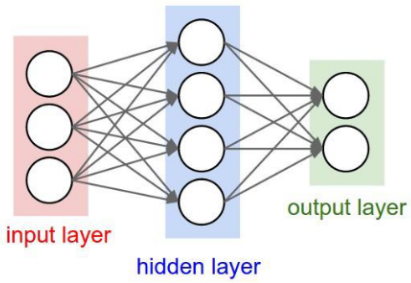


Figure 2.22: 2-Layer Neural Network

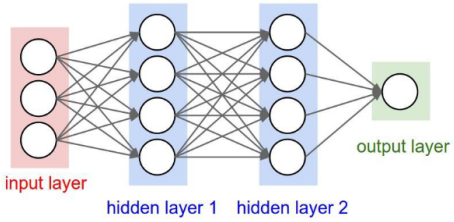


Figure 2.23: 3-Layer Neural Network

Example

Given the code below, for a binary neural network classifier, answer the following questions.

```
def __init__ ( self ) :
    super ( MLP , self ) . __init__ ()
    self . firstLayer = nn . Linear (20 , 10)
    self . middleLayer = nn . Linear (10 , 10)
    self . lastLayer = nn . Linear (10 , 1)
def forward ( self , input ) :
    activation1 = self . firstLayer ( input )
    activation1 = F . relu ( activation1 )
    activation2 = self . middleLayer ( activation1 )
    activation2 = F . relu ( activation2 )
    activation3 = self . middleLayer ( activation2 )
    activation3 = F . relu ( activation3 )
    activation4 = self . lastLayer ( activation3 )
    return activation4
```

What is the input size of the network?:
(20, B) Where B is the batch size.

How many layers are there in the following network?:
There are 4 layers in the network. Recall that the output layer is also considered a layer, while the input layer is not considered.

What is the total number of parameters in the network (including biases)?:
441 = (20*10 + 10) + (10*10 + 10) + (10*10 + 10) + (10*1 + 1)

Idea

To calculate the **number of parameters** for a fully-connected network layer:

Number of weights $=$ Input size (entering layer) $\cdot$ Number of Neurons (in layer) $+$ Bias Term (for each neuron in layer)

Only include bias term if it is assumed to exist.

Example

Calculating number of parameters (weights) for Fully-Connected Networks:

- Input: 200 x 200 pixels = 40000
- 1st hidden layer: 500 neurons
- 2nd hidden layer: 200 neurons

(Weights = 40 000 x 500 + 500 x 200) + (Bias = 500 + 200) = 20100700 parameters

Example

Calculating number of parameters (weights) for Fully-Connected Networks:

- self.fc1 = nn.Linear(16, 100)
- self.fc2 = nn.Linear(32, 100)

(Weights = 16 x 100 + 32 x 100) + (Bias = 100 + 100) = 5000 parameters

2.7 Hyperparameters

While neural network **parameters**, such as weights, are updated through **Gradient Descent (Inner loop of optimization)**, **hyperparameters** are tuned by methods such as **Grid Search** and **Random Search (Outer loop of optimization)**. Most of the time, **Random Search is all we need** as **Grid Search is very time consuming and costly**.

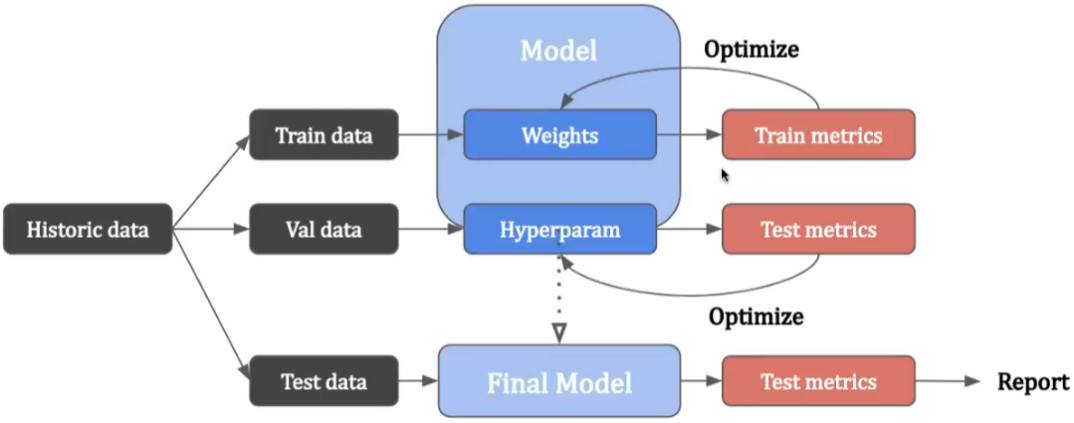


Figure 2.24: How split data affects optimization

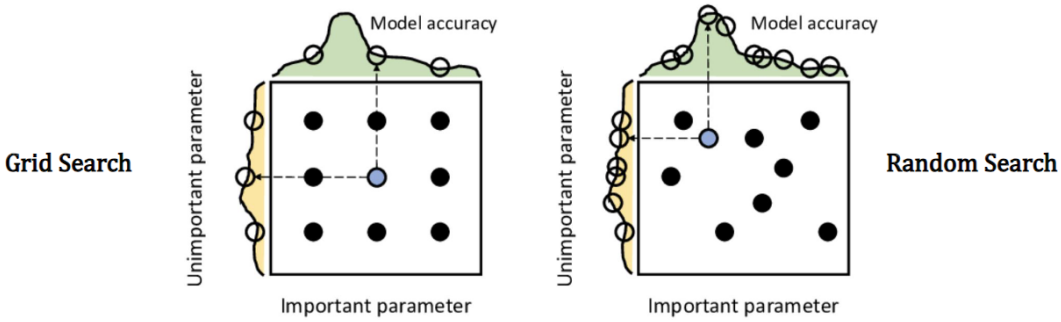


Figure 2.25: Grid Search and Random Search

Definition

Grid Search: involves creating a **predefined** grid of hyperparameter values to explore. It **exhaustively evaluates all possible combinations** of these values. It systematically searches through the entire parameter space, testing each combination to find the **best-performing set** of hyperparameters. Grid search is ideal when the hyperparameter space is relatively small and the interactions between hyperparameters are not too complex.

Pros of Grid Search:

- Exhaustive search that **guarantees optimal hyperparameters** within the specified search space.
- Systematic approach that **covers all combinations**.

Cons of Grid Search:

- **Computationally expensive** when the search space is large.
- Inefficient when many **hyperparameters are not influential**.

Definition

Random Search: randomly samples hyperparameters from a specified distribution or range. It performs a **certain number of random trials**, independently of each other, and evaluates the model's performance with each set of hyperparameters. Random search is particularly useful when the search space is large or when the impact of individual hyperparameters is less clear. **Generally better than Grid Search**.

Pros of Random Search:

- More **efficient** when the search space is large or when some hyperparameters are less influential.
- **Less computationally demanding** compared to grid search for large search spaces.

Cons of Random Search:

- There's no guarantee of finding the optimal hyperparameters, as it **relies on chance**.
- It might require more trials to converge to the best solution compared to grid search.

2.8 Optimizers

Defining a loss function turns a **learning problem into an optimization problem**. An optimizer determines, based on the value of the loss function, **how each parameter (weight) should change**. The optimizer solves the **credit assignment problem**. We assign credit to the parameters based on how the network performs by using optimizers that are based on **gradient descent**.

Stochastic Gradient Descent (SGD)

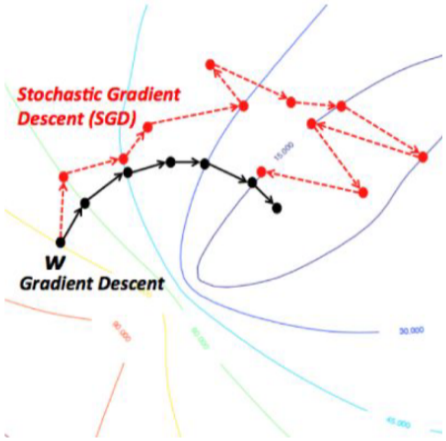


Figure 2.26: Stochastic Gradient Descent

- For each iteration, evaluate a training sample from the dataset taken at random.
- Computing the gradient takes less time, but... may not actually be faster...
- Optimization path that looks rather erratic.
- SGD allows you to do more of a global search for an optimum, often resulting in a better set of weights for your model.
- Gradient Descent (GD) on the entire training data!

Definition

Stochastic Gradient Descent: an optimization algorithm used to iteratively (**one at a time**) update the parameters of a model by computing the gradient of the loss function using a **randomly selected subset of the training data** in each iteration.

Mini-Batch Gradient Descent

Instead of working with one sample at a time... can apply batching...

1. Use our network to make predictions for n samples
2. Compute the average loss for those n samples
3. Take a "step" to optimize the average loss of those n samples

Theorem

Set **batch size** close to **GPU memory** to use as much memory as possible!

Batch size: Number of training examples used per optimization "step".
Iteration: One step - The parameters are updated once per iteration.
Epoch: Number of times all the training data is used once to update the parameters.
Suppose there are 1000 samples in the training data, if we set the batch size to 10, then 1 epoch will contain 100 iterations.

Definition

Mini-Batch Gradient Descent: an optimization algorithm used to update the parameters of a model by computing the gradient of the loss function using a small subset (mini-batch) of the training data in each iteration, striking a **balance between the efficiency of Stochastic Gradient Descent (SGD) and the stability of Batch Gradient Descent**.

Warning

What happens when the **batch size is ineffective**?

- Too small:
  - We optimize a (possibly very) different function loss at each iteration
  - Noisy
- Too large:
  - Expensive
  - Average loss might now change very much as batch size grows
  - The true gradient is not always the best gradient for optimization; some amount of noise in your gradients can help training (converge faster), so larger batch size is not always better.

A deep neural network has millions or billions of parameters. Real gradient descent of a deep network is optimization in millions of dimensions! Most points of zero gradients are saddle points. Plateaus are a problem but can be addressed using specialized variants on gradient descent.

Stochastic Gradient Descent (SGD) with Momentum

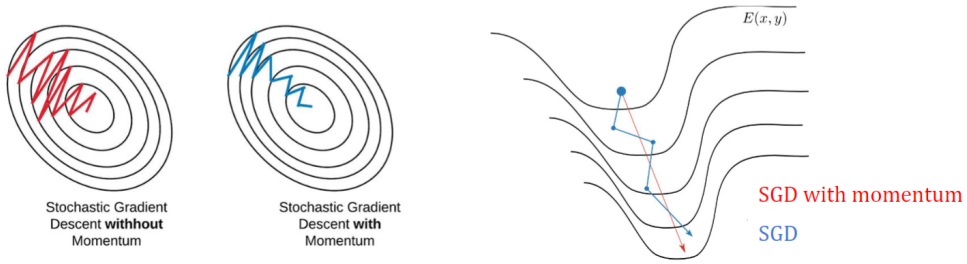


Figure 2.27: SGD with Momentum

Definition

Ravines: areas where the surface curves much more steeply in one dimension than in another, common around local optima.

- SGD has trouble navigating ravines $\rightarrow$ it oscillates across the slopes of the ravine.

- Momentum helps accelerate SGD in the relevant direction and dampens oscillations.
- The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions
- Analogy $\rightarrow$ we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way until it reaches its terminal velocity

Definition

Stochastic Gradient Descent with Momentum: an optimization algorithm used to iteratively update the parameters of a model by incorporating a moving average of past gradients. This momentum term helps accelerate the convergence process by reducing oscillations and smoothing out the optimization path, leading to faster and more stable convergence.

$$v_{t+1} = \mu v_t - \alpha \nabla J(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$

Where:

- v_{t+1}: Updated velocity at time step $t + 1$
- μ: Momentum coefficient (typically between 0 and 1)
- v_t: Velocity at time step t
- α: Learning rate
- $\nabla J(\theta_t)$: Gradient of the cost (loss) function J with respect to the parameters θ at time step t
- θ_{t+1}: Updated parameters at time step $t + 1$
- θ_t: Parameters at time step t

Adaptive Moment Estimation (Adam)

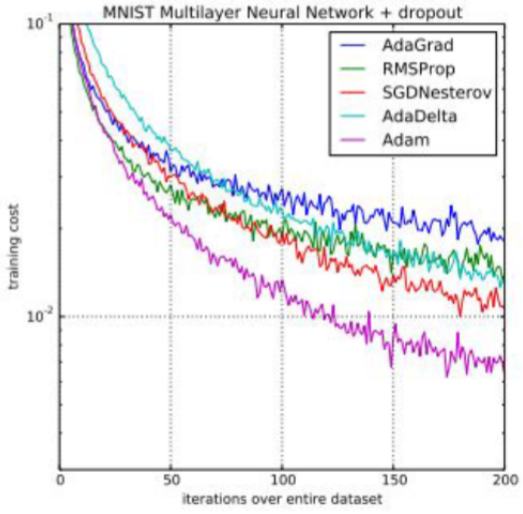


Figure 2.28: Optimizers relative to each other in terms of training cost over iterations

Definition

Adaptive Moment Estimation: a popular optimization algorithm used for tuning the parameters of a model during training. It combines concepts from both Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp) to adaptively adjust learning rates for each parameter. This helps achieve efficient and stable convergence by individually adapting the learning rates while also incorporating momentum-based updates.

- Adaptive learning rates $\rightarrow$ each weight has its own rate

- Incorporates momentum and adaptive learning rate:

  - rapid convergence

  - requires minimal tuning

  - commonly used optimizer

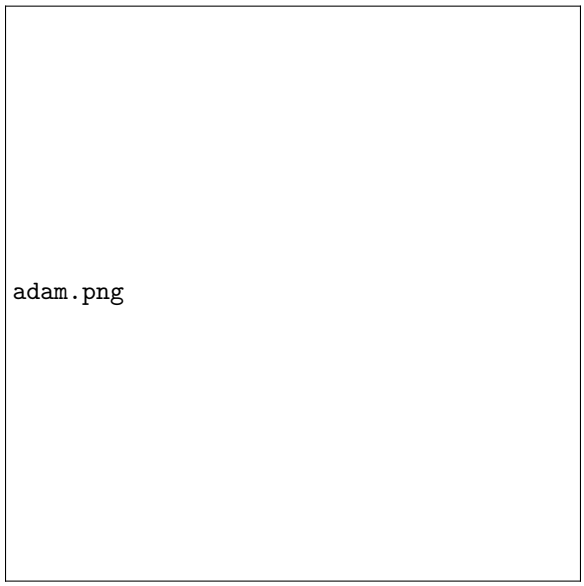


Figure 2.29: Adam Optimizer Equations

Idea

Optimizer Summary:

1. **SGD (Stochastic Gradient Descent)**:
   - **Use when:** Training on large datasets with limited computational resources.
   - **Scenario:** Updates model parameters using the gradient of a single randomly selected data point.
   - **Pros:** Faster updates, can escape local minima, suitable for online learning.
   - **Cons:** Noisy updates, slower convergence on some problems.
2. **SGD with Momentum**:
   - **Use when:** Training needs faster convergence and better handling of noisy gradients.
   - **Scenario:** Incorporates a moving average of past gradients to accelerate convergence.
   - **Pros:** Faster convergence, reduced oscillations, better handling of noisy data.
   - **Cons:** May overshoot in some cases.
3. **Adam (Adaptive Moment Estimation)**:
   - **Use when:** Training deep neural networks with diverse architectures.
   - **Scenario:** Combines adaptive learning rates for each parameter and momentum-like behavior.
   - **Pros:** Fast convergence, adaptive learning rates, well-suited for complex models.
   - **Cons:** Can require tuning, memory-intensive.
4. **Mini Batch Gradient Descent**:
   - **Use when:** Training on medium to large datasets efficiently.
   - **Scenario:** Computes gradient on a small subset (mini-batch) of data.
   - **Pros:** Faster convergence than pure SGD, benefits from vectorized operations.
   - **Cons:** Batch size selection matters, noise in updates, needs more memory.

2.9 Learning Rate

The learning rate determines the **size of the step** that an optimizer takes during each iteration. Larger step size means a bigger change in the parameters (weights) in each iteration.

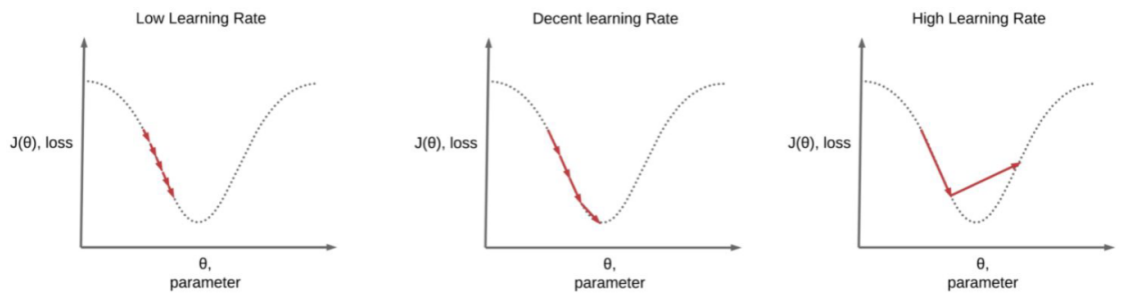


Figure 2.30: Learning Rate Size

Warning

What happens when the **learning rate size is ineffective**?

- Too small:
  - Very small parameter change
  - Longer training time
- Too large:
  - Noisy
  - Detrimental to training

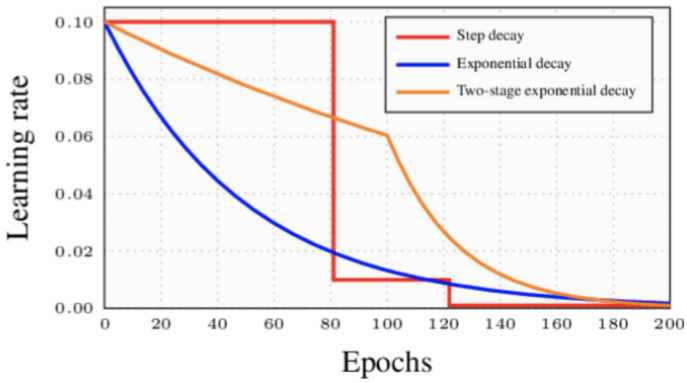


Figure 2.31: How learning rate changes over epochs

The appropriate learning rate depends on:

- The learning problem
- The optimizer
- The batch size
  - **Large batch**: larger learning rates
  - **Small batch**: smaller learning rates
- The stage of training
  - Reduces as training progresses

2.10 Normalization

Definition

Normalization: refers to the process of **scaling and shifting** input data or intermediate activations to ensure that they have a **consistent and suitable range**. This aids in improving the stability and convergence of training by mitigating issues related to varying magnitudes of data across features or layers.

We always normalize the inputs to prevent the model from paying attention to the features with larger range.

$$X_i = \frac{X_i - \mu_i}{\sigma_i}$$

Where:

- X_i is the original value of the ith element in the data vector.
- μ_i is the mean (average) of all ith elements across a batch of data.

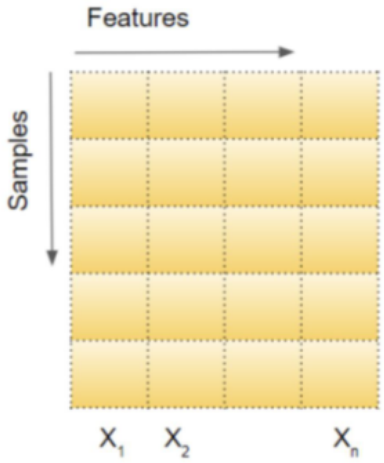


Figure 2.32: Normalization

- σ_i is the standard deviation of all ith elements across the same batch of data.

- However, this only normalizes data for the first layer. How do we normalize activations of each layer for the next layer?

Batch Normalization

Definition

Batch Normalization: a technique used to normalize the activations of a layer by adjusting them to have zero mean and unit variance across a mini-batch of training examples. This improves training stability, speeds up convergence, and enables higher learning rates by reducing the internal covariate shift problem.

Theorem

Internal Covariate Shift Problem: the change in the distribution of the input to a given layer during training. As the network learns and its parameters get updated, the distribution of activations in each layer may shift, making the optimization process more difficult. This can slow down training and require the use of lower learning rates to prevent the network from diverging or getting stuck.

Batch normalization essentially combats the internal covariate shift problem by normalizing the inputs of each layer within a mini-batch during training.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

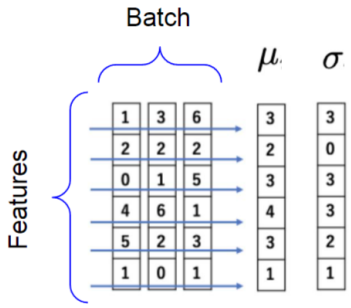


Figure 2.33: Batch Normalization

Definition

Inference Time: refers to the phase when a trained machine learning model is applied to new, unseen data to make predictions or produce outputs based on the knowledge it gained during its training phase.

Batch Normalization primarily affects the **training phase** of a neural network, and its operation during inference time differs slightly from its behavior during training. During training, Batch Normalization computes the mean and standard deviation of each feature within a mini-batch of data, normalizes the features, and then scales and shifts them based on learnable parameters. These normalization statistics can change from one batch to another, helping with faster convergence and better generalization.

However, during inference time, the model processes individual examples or small batches of examples one at a time, rather than in the larger training batches. This raises the question of how to handle the normalization statistics, as there is no longer a batch of data to compute them from.

A way to solve this issue is **Moving Average**.

Definition

Moving Average: addresses the challenge of Batch Normalization during inference by maintaining running statistics, including the **mean** and **standard deviation** of features, over the course of training. These running statistics are computed by **aggregating the statistics from all training batches**. During inference, these **accumulated statistics are used to normalize the input features**, providing a consistent behavior that aligns with the model's learned behavior during training. This method helps ensure that the benefits of Batch Normalization are retained when making predictions on new data while avoiding the need to compute statistics on small inference batches.

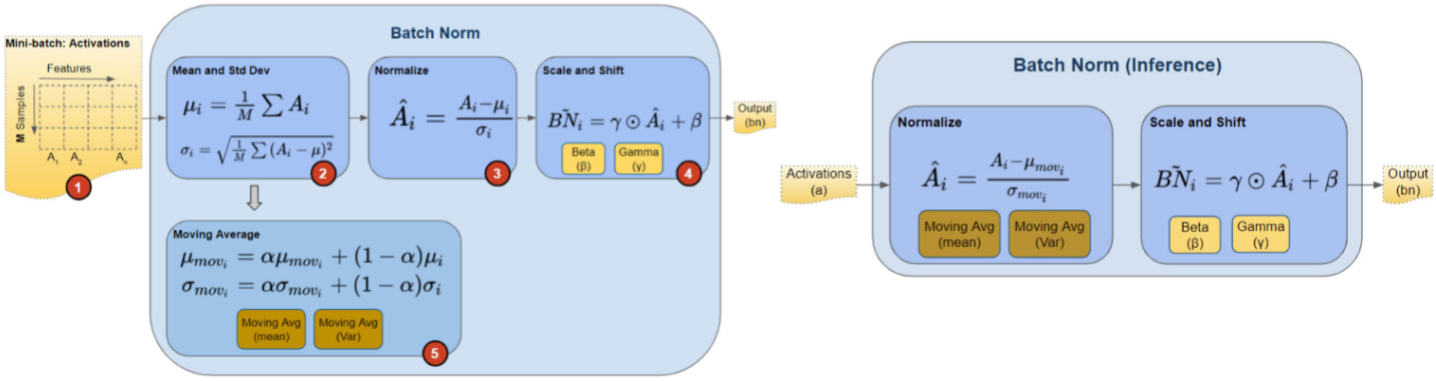


Figure 2.34: Inference Time

Pros:

- Higher learning rate → speeding up the training
- Regularizes the model
- Less sensitivity to initialization

Cons:

- Depends on batch size → No effect with small batches
- Cannot work with SGD

Warning

If the batch size is 1, we cannot use batch normalization!

Layer Normalization

Definition

Layer Normalization: a technique used to normalize the activations of a layer across the entire batch, focusing on the mean and standard deviation of each feature independently. This helps stabilize training, improve convergence, and reduce sensitivity to weight initialization, enhancing the network's performance.

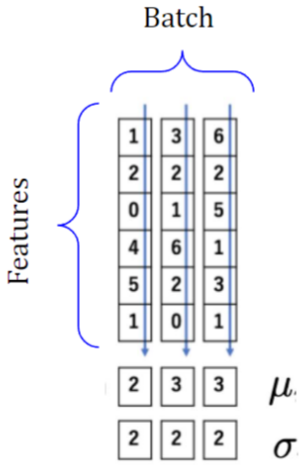


Figure 2.35: Layer Normalization

- Normalization is applied on the neuron for a single instance across all features
- Simpler to implement, no moving averages or parameters
- Not dependent on batch size
- Works on par with batch normalization or even better

2.11 Regularization

Definition

Regularization: refers to techniques that mitigate overfitting by adding constraints or penalties to the model's training process. These techniques discourage the model from fitting noise in the training data and encourage it to learn more generalizable patterns, ultimately enhancing its ability to perform well on unseen data. Essentially, **techniques that make it hard for the model to memorize.**

Dropout

Definition

Dropout: involves randomly **deactivating** (dropping out) **a proportion of neurons** during each training iteration. This helps prevent overfitting by encouraging the network to learn more robust and independent features, improving its generalization on new data.

- Forces a neural network to learn more robust features by forcing to learn the **underlying distribution** of the data rather than memorizing it
- A very **straightforward** and **efficient** way to regularize deep models
- During training → Drop activations (set to 0) with probability p
  - Implementation wise, neurons are muliplied by 0 to drop, and 1 to be kept
- During inference → multiply weights by (1-p) to keep the same distribution

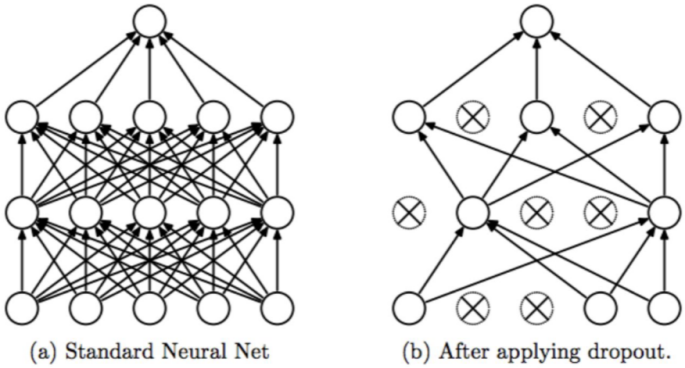


Figure 2.36: Dropout

Weight Decay (L2)

Definition

Weight Decay (L2): a technique that involves adding a **penalty term (summation of all weights of the network) to the loss function** during training. This penalty discourages the model from assigning excessively large weights to its parameters, promoting simpler and more generalizable solutions, which reduces the risk of overfitting.

$$E(W;y,t) = E(W;y,t) + \frac{\alpha}{2} \parallel W \parallel_2^2 \longrightarrow \frac{\partial E}{\partial W} = \frac{\partial E}{\partial W} + \alpha W$$

$$W_{t+1} = W_t - \gamma \left(\alpha W_t + \frac{\partial E}{\partial W} \right)$$

Figure 2.37: Weight Decay

- When a model ”wants” to overfit or memorize, it tends to **inflate** weights, which causes the space to be warped in a strange way, leading to memorization
- Prevents the weights from growing too much $\rightarrow$ Lowering variance
- Weight reduction is multiplicative and proportion to the scale of W

Early Stopping with Patience

Definition

Early Stopping with Patience: refers to **monitoring** the model's performance on a validation set during training and **halting** the training process when the performance stops improving for a certain number of consecutive epochs (patience). This technique prevents overfitting by selecting a model that achieves good validation performance before it starts to degrade due to excessive training.

- In each training iteration observe the validation loss
- As soon as validation loss starts to increase, start a counter
- If the validation loss decreases, reset the counter
- Otherwise, wait for a fixed iterations (patience) and then stop the training

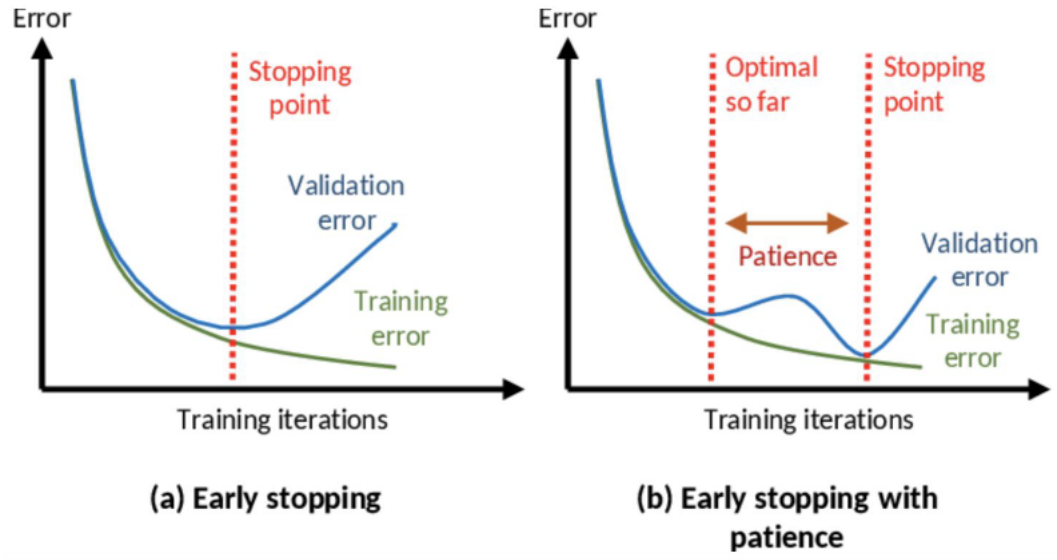


Figure 2.38: Early Stopping with Patience

2.12 Evaluation and Debugging

Debugging Tips

- Make sure your model **can overfit**
  - Make sure you can get loss to decrease w.r.t training data
  - If model cannot even learn the training data, it's not going to be able to predict validation or test data
- Make sure that your network is training: i.e. loss is going down.
  - Sanity check!
- Ensures that you are using the right variable names, and rule out other programming bugs that are difficult to discern from architecture issues.
- Confusion Matrix
  - True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN)
- 2D Projections of Data (visualizing class clusters)
  - PCA, t-SNE

Confusion Matrix

Definition

Confusion Matrix: tabular representation used to display the performance of a classification model. It summarizes the predicted class labels against the actual class labels, showing counts of true positive, true negative, false positive, and false negative predictions.

Warning

Accuracy is only valid when class distributions are equal (**balanced dataset**)! **F1 Score** should be used in cases where the **dataset is unbalanced**.

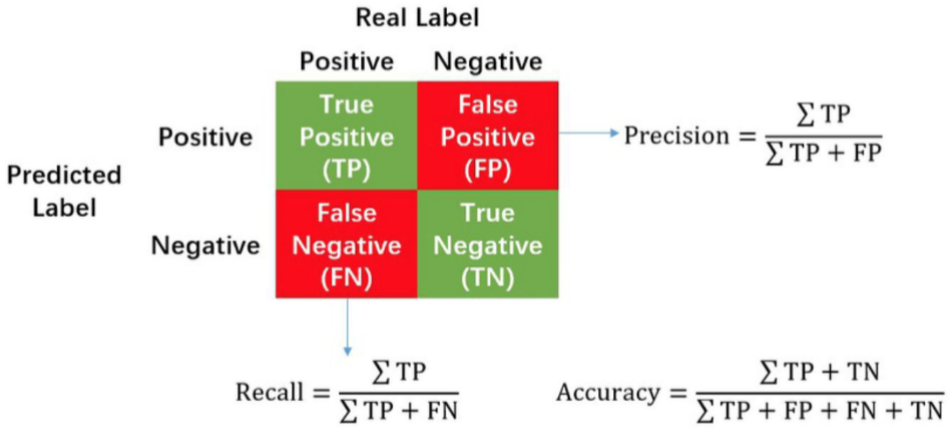


Figure 2.39: Confusion Matrix

$$F1\ Score = 2 \times \frac{recall \times precision}{recall + precision}$$

Figure 2.40: F1 Score

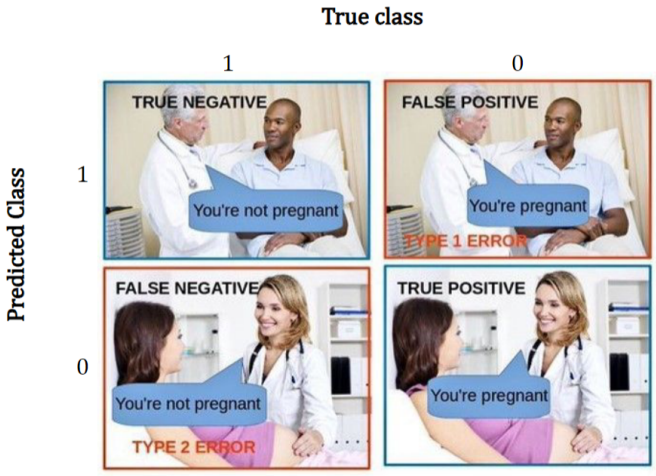


Figure 2.41: Confusion Matrix Example

MNIST 2D Visualization

Idea

Data that appears neatly clustered in groups by class indicates a **well-performing model**.

Figure 2.42: t-SNE for 2D projection and visualization of data structure

3 Convolutional Neural Networks (CNNs)

3.1 Motivation

Fully connected networks become useless when the dataset is not nicely preprocessed, centered, etc. When the size of the input is incompatible with the dimensions of the model, the fully connected will no longer work because it needs to be retrained.

Using a large fully connected network has some downsides:

- **Computation complexity (exponentially) grows** $\rightarrow$ harder to train
- **Larger capacity** $\rightarrow$ more data to generalize
- **Bad inductive bias** $\rightarrow$ Ignores geometry of image data (relative distances between objects in an image)
  - Recall that in fully connected networks, the pixels of an image are stacked to make a line of pixels (1D) instead of a 2D image
- **Not flexible** $\rightarrow$ Different image sizes require different models
  - When we want to put a new image size as input in a model, we need to train the model from scratch

3.2 Convolution Operator

Definition

Convolution: Convolution is a mathematical operation on two functions f and g that expresses how the shape of one is modified by the other. The function f is the input function and the function g is known as the **kernel**.

$$(f * g)(t) = \sum_{\tau=-\infty}^{\infty} f(\tau) \cdot g(t - \tau)$$

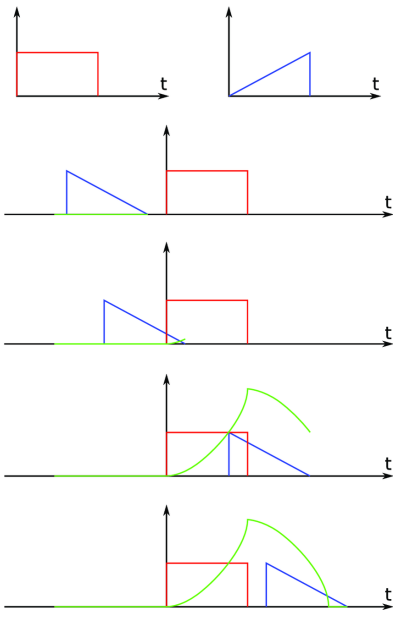


Figure 3.1: Convolution

3.3 Convolution in 2D for Images

To apply the convolutional operator to 2D space, we simply **add another dimension** to the equation.

Convolution of Image I with filter kernel K:

1. Multiply each pixel in the range of the kernel by the corresponding element of the kernel.
2. Sum all these products and write the result to a new 2D array.
3. Slide the kernel across all areas of the image until you reach the image's edges.
4. Once the edge is reached, go back horizontally to the start but slide down one pixel

$$y[m,n] = I[m,n] * K[m,n = \sum_{j=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I[i,j] \cdot K[m-i, n-j]$$

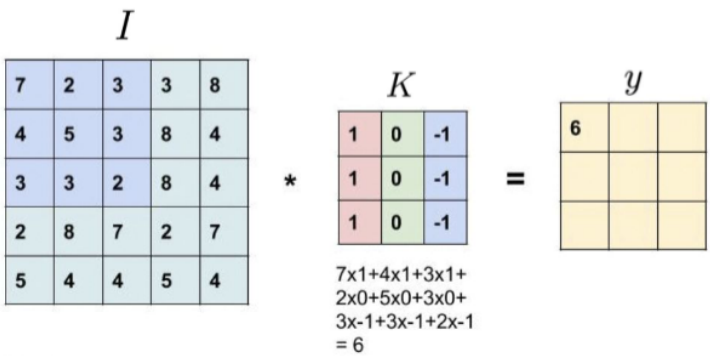


Figure 3.2: Step 1

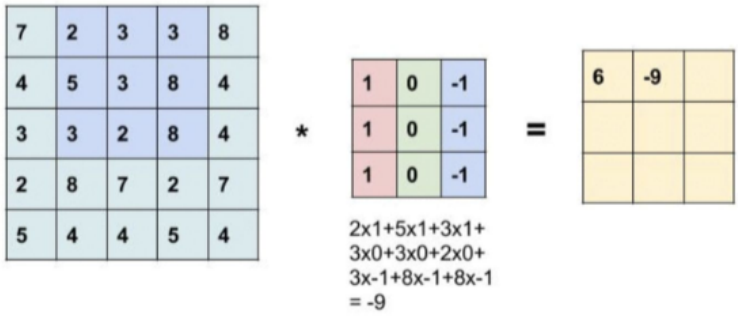


Figure 3.3: Step 2

Idea

Convolution of an image is applying a kernel to the image while sliding the kernel across the image

Why use convolution?

- In signal processing, convolution is used to **extract important features** of the input signal
- Instead of trying to make prediction directly on image pixels, such as what we've done in fully connected networks, we are applying a kernel to the image pixels to extract interesting features, then do predictions based on those features (which are the most important assets of the image)
- The output of the kernel applied to the image is the most important assets of the image extracted by the kernel

Definition

Kernel: a kernel, also known as a filter, is a small matrix used to extract specific features from an input image or data. It's applied through the process of convolution to scan the input and produce feature maps that highlight particular patterns, such as edges, textures, or other relevant characteristics.

Examples of kernels (filters) applied to images

- **Blurring:** averages out pixel intensities in an image

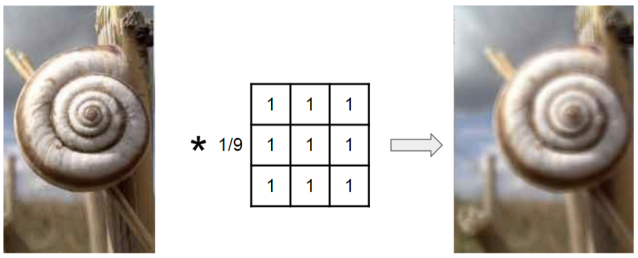


Figure 3.4: Blurring Filter

- **Vertical Edge Detector**

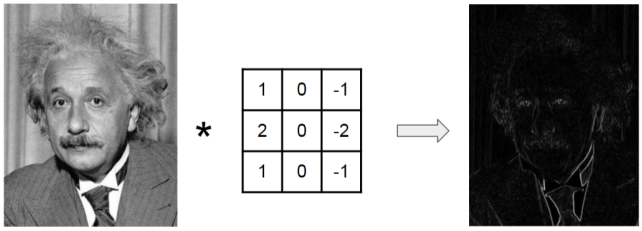


Figure 3.5: Vertical Edge Detector

- **Horizontal Edge Detector**

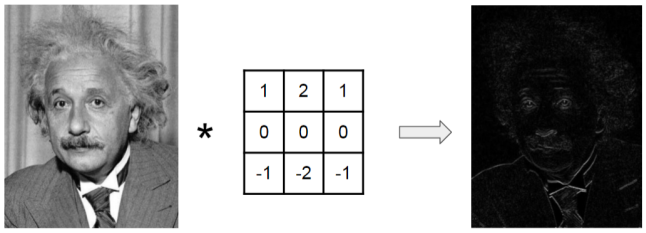


Figure 3.6: Horizonal Edge Detector

- **Blob Detector:** detect regions that differ in properties, such as brightness or color, compared to surrounding regions

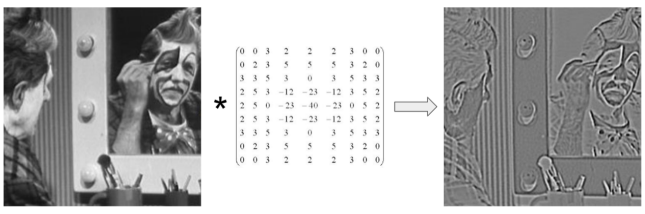


Figure 3.7: Blob Detector

Where do these kernels come from?

- Before deep learning, people would spend years coming up with these kernels (hand-crafted)
- Classic computer vision used multi-stage feature (kernel) engineering

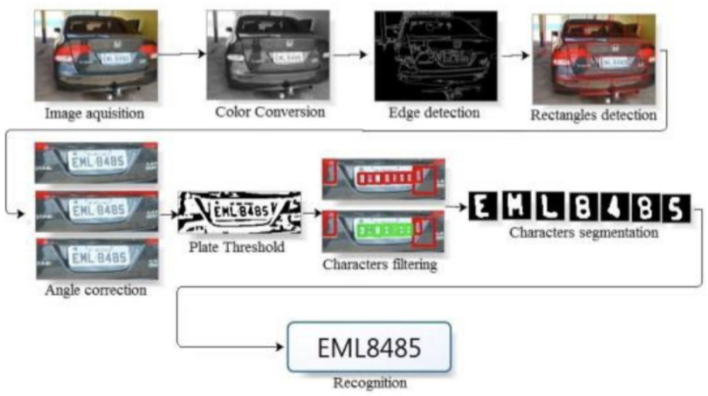


Figure 3.8: Multi-stage feature (kernel) engineering of a licence plate recognition system

- There were two main problems with this method:
  1. If one step fails, the subsequent steps wil also fail
  2. Hand-written algorithms don't work in real world scenarios due to fringe cases
- This is where convolutional neural networks become useful

3.4 Convolutional Neural Networks

Definition

Convolutional Neural Network: a type of deep learning model specifically designed for processing and analyzing grid-like data, such as images and sequences. It uses layers of learnable filters (kernels) to perform convolutions on the input data, enabling the network to automatically learn and extract hierarchical features. CNNs are highly effective for tasks like image recognition, object detection, and image generation due to their ability to capture local patterns and spatial relationships within the data.

Convolutional neural networks were **inspired by the Hubel and Wiesel Cat Experiments (1958-1959)**:

- Individual neurons respond to stimuli only in a restricted region of the visual field known as the **Receptive Field**.
- Collection of such fields overlap to cover the entire visual area
- Some neurons react only to images of horizontal lines, while others react line orientations
- Higher-level neurons are based on the outputs of neighbouring lower-level neurons

Introduce convolutional filters into neural networks so that we don't have to hand craft the features!

- **Locally connected layers**: local features in small regions of the image
- **Weight sharing:** detect the same local features across the entire image
- Neural Network **learns the kernel values** (or weights)
- More efficient than fully-connected networks since **outputs of kernels are connected to neurons** instead of each individual pixel of an image (weight sharing).
- **Detecting:** the output (activation) is high if the feature is present
- **Feature:** something in the image, like an edge, blob, or shape

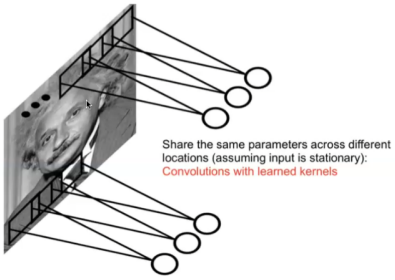


Figure 3.9: Weight Sharing

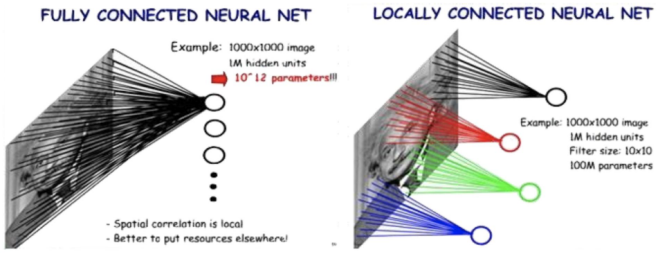


Figure 3.10: Fully Connected vs. Convolutional

While fully-connected networks require us to preprocess the input,

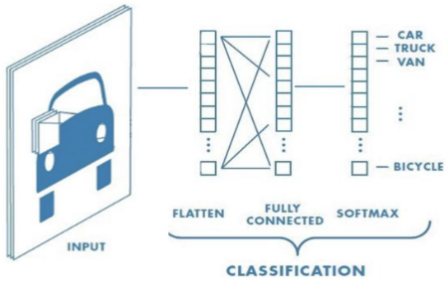


Figure 3.11: Fully connected network

convolutional neural networks apply convolution to image tensors and allow the network to learn and extract important low and high level features.

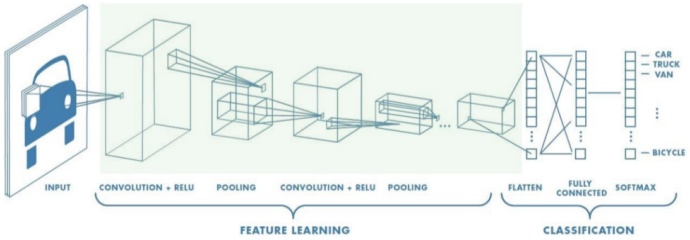


Figure 3.12: End-to-end network

Thus, in a model that uses both fully connected and convolutional layers, the **CNN part is the encoder** and the **FC part is the classifier**. The CNN part extracts features, which are passed to the FC part that classifies the input.

Idea

Because everything is **jointly connected, the weights** of the convolutional and fully connected layers are **jointly optimized** according to the loss function. This means that the **end-to-end** network is not going to face the same problem as the multi-stage algorithm described earlier.

The numbers in the kernel are learned through gradient descent.

Forward and Backward Pass (kernels):

- Initialize the kernels randomly
- In forward pass, convolve the image with the kernel
- In backward pass, update the kernel using gradients

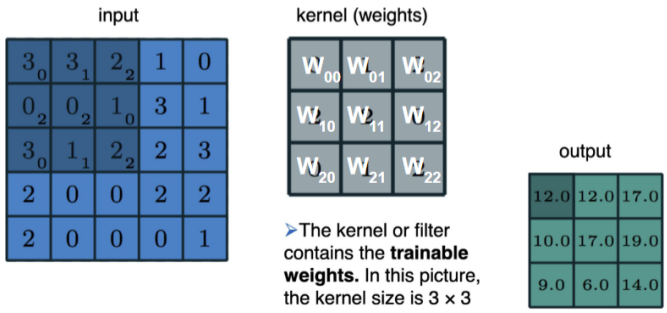


Figure 3.13: Kernel as a matrix of weights

General CNN Terminology:

Definition

Zero Padding: adding zeros around the border of the image before convolution.

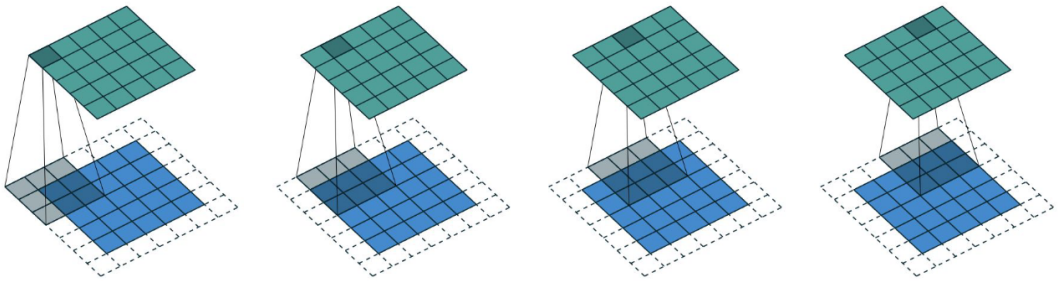


Figure 3.14: Zero Padding

Zero padding is applied for two main reasons:

1. Keep width and height consistent with the previous layer
2. Keep the information around the border of the image

Definition

Stride: distance between two consecutive positions of the kernel.

Stride allows us to control the **output resolution**.

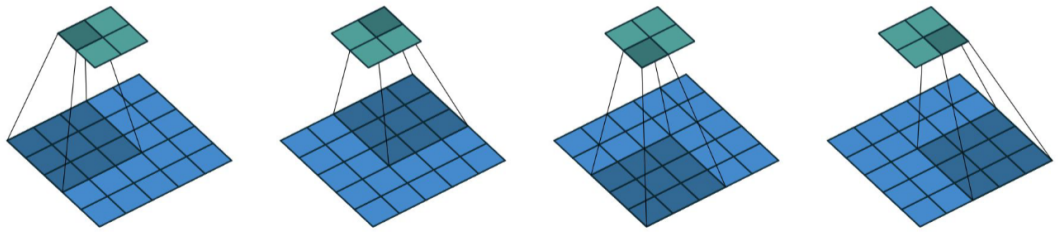


Figure 3.15: Stride

Definition

Feature Map: a 2D or 3D array of values that represents the activations of specific features detected within an input data volume, generated by applying filters through convolutional operations in neural networks, particularly in convolutional neural networks (CNNs).

Theorem

Computing the output size of the feature map (from convolving an image with a kernel)

For each dimension of an input image with:

- Image dimension of size **i**
- Kernel of size **k**
- Padding of size **p**
- Stride of size **s**

The size of output dimension is computed by:

$$o = [\frac{i + 2p - k}{s}] + 1$$

This equation assumes the kernel and image sizes are square!

In the case that the sizes are not square:

- Image dimension of size **i x j**
- Kernel of size **k x l**
- Padding of size **p**
- Stride of size **s**

We need to calculate the output of each dimension separately:

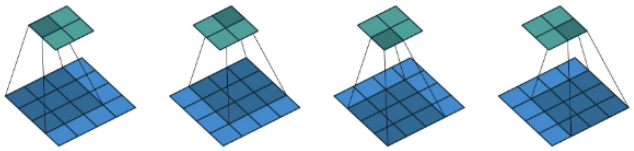
$$o_1 = [\frac{i + 2p - k}{s}] + 1$$

$$o_2 = [\frac{j + 2p - l}{s}] + 1$$

- No padding, unit stride → (the output size < the input size)

$i = 4,\ k = 3,\ s = 1$ and $p = 0$

$$o = \left\lfloor \frac{4 + 2 \times 0 - 3}{1} \right\rfloor + 1 = 2$$



- Arbitrary padding, unit strides

$i = 5,\ k = 4,\ s = 1$ and $p = 2$

$$o = \left\lfloor \frac{5 + 2 \times 2 - 4}{1} \right\rfloor + 1 = 6$$

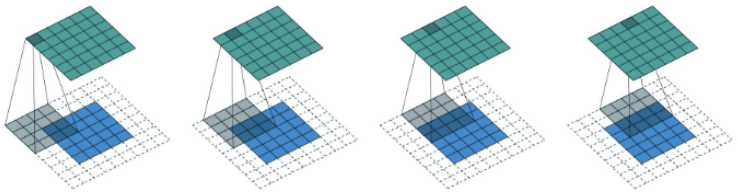
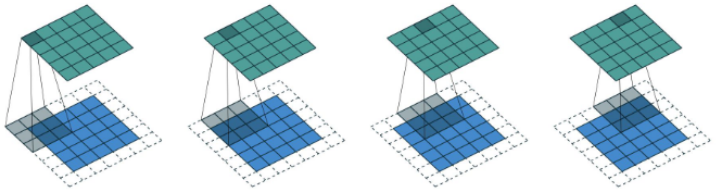


Figure 3.16: Output Calculation Example 1

- Half padding, unit strides → (the output size be the same as the input size)

$i = 5,\ k = 3,\ s = 1$ and $p = 1$

$$o = \left\lfloor \frac{5 + 2 \times 1 - 3}{1} \right\rfloor + 1 = 5$$



- Full padding, unit strides → (the output size >= the input size)

$i = 5,\ k = 3,\ s = 1$ and $p = 2$

$$o = \left\lfloor \frac{5 + 2 \times 2 - 3}{1} \right\rfloor + 1 = 6$$

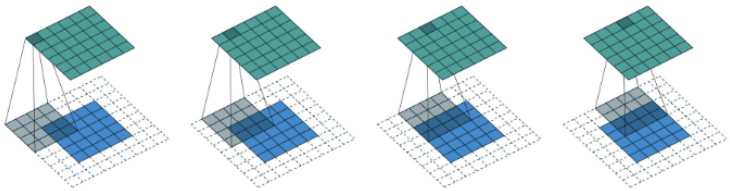
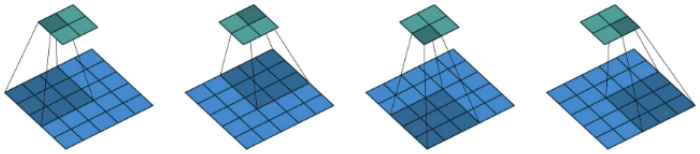


Figure 3.17: Output Calculation Example 2

- No padding, arbitrary strides

$i = 5,\ k = 3,\ s = 2$ and $p = 0$

$$o = \left\lfloor \frac{5 + 2 \times 0 - 3}{2} \right\rfloor + 1 = 2$$



- Arbitrary padding and strides

$i = 6, k = 3,\ s = 2$ and $p = 1$

$$o = \left\lfloor \frac{6 + 2 \times 1 - 3}{2} \right\rfloor + 1 = 3$$

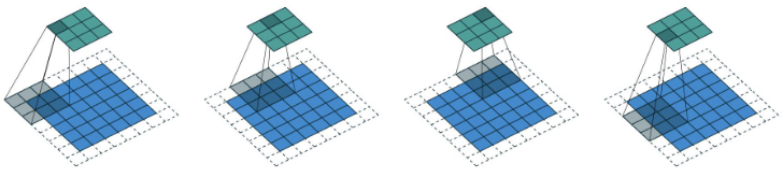


Figure 3.18: Output Calculation Example 3

3.5 Convolution in 3D for RGB input

In 3D, the kernel becomes a **3D tensor.**

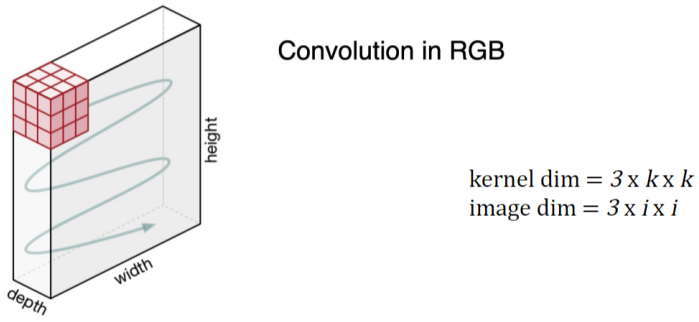


Figure 3.19: Convolution in 3D (RGB)

Example

Calculating the number of trainable weights from an RGB input image and a convolution kernel

- Colored input image: 3×28×28
- Convolution kernel: 3×3×3

The dimension of the kernel is 3 x 3 x 3, therefore there are $3x3x3 = 27$ trainable weights.

Expanding Feature Maps

If we just use **one convolution, or one kernel, we are only able to detect one type of feature**, which isn't enough for information-rich images. The solution is to apply **multiple kernels at the same time to the image in parallel**, which will each extract a different feature, allowing for more than one feature to be extracted at a time.

Each kernel will produce an output, and these outputs will be stacked on top of each other to form the depths of the output.

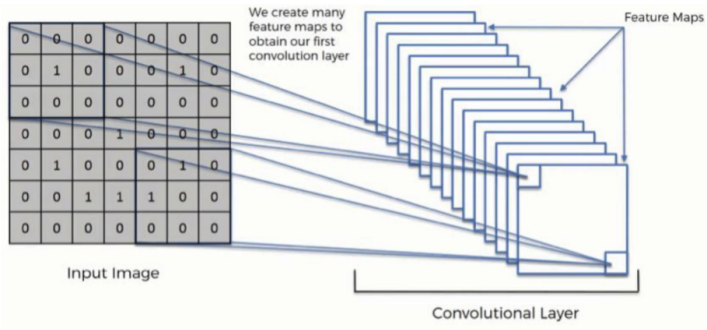


Figure 3.20: Applying multiple kernels

Theorem

The depths of the output = Number of kernels

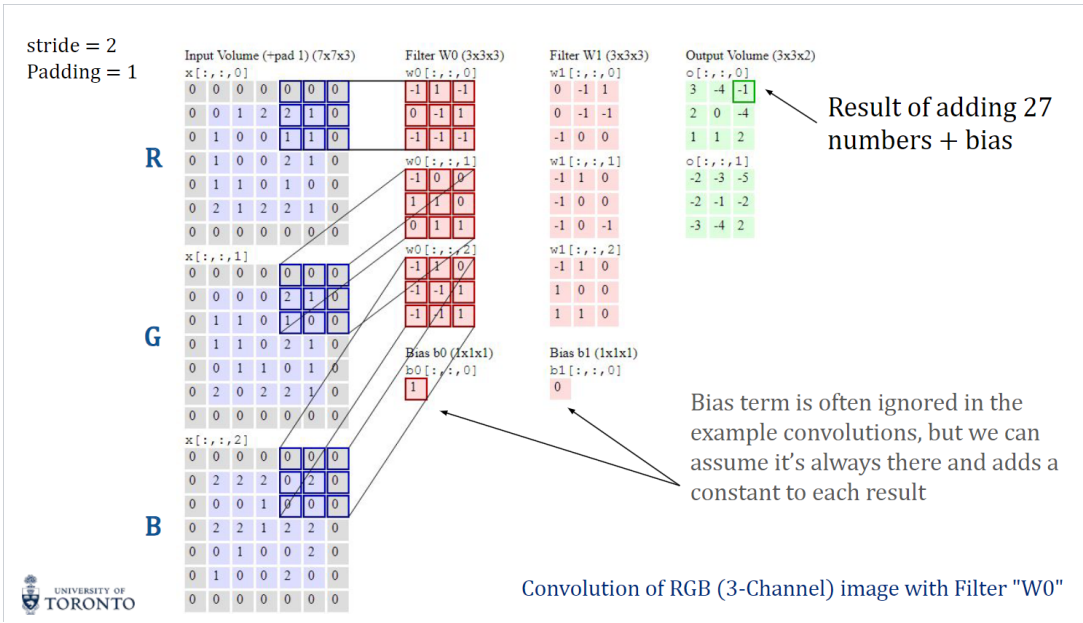


Figure 3.21: Applying multiple kernels

Example

Calculating the number of input channels, output channels, and trainable weights from an RGB input image and a convolution kernel

- **Colored input image: 3×28×28**
- **Convolution kernel: 5x3×8×8**
  - There are 5 kernels being applied in parallel. There are each 3x8x8 in size
- Input channels: 3 input channels because the depth of the input image is 3
- Output channels: 5 output channels because there are 5 kernels
- Trainable weights: 5 x 3 x 8 x 8 = 960 (weights) + 5 (bias for each kernel)

What is preventing these filters from learning the same features?

- **Random Initialization**: we assign random initial weights to the kernels which cause them to all learn different features

3.6 Pooling Operator

In order to **consolidate information and remove information that is not useful** with convolutional layers, we can use pooling operators or strided convolutions. Compressing information allows the network to establish the most **foundational features or "elements" of a class** so that it may **generalize well to unseen data.** As we go deeper into the network, we compress the data more and more.

Definition

Pooling Operator: a **downsampling** technique that reduces the spatial dimensions of the input data by aggregating information from neighboring elements. It aims to capture important features while decreasing the computational complexity and memory requirements of the network.

Max Pooling

- pooling layers provide **invariance to small translations** of the input

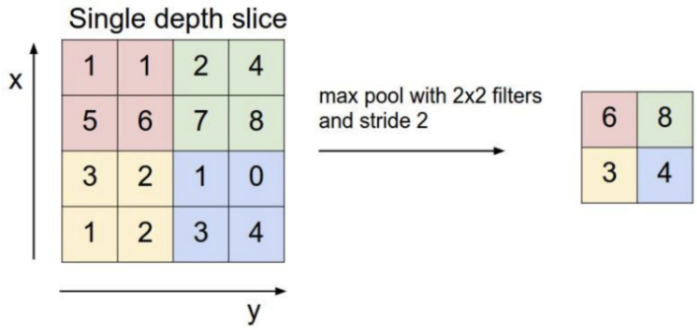


Figure 3.22: Max Pooling

Definition

Max Pooling: a pooling operator used to downsample the spatial dimensions of input data. It involves **dividing the input into non-overlapping regions and selecting the maximum value from each region**, effectively capturing the most prominent feature within that region.

Theorem

The size of max pooling output is computed by:

$$o = [\frac{i-k}{s}] + 1$$

Where:

- Image dimension of size i
- Kernel of size k
- Stride of size s

Average Pooling

- usually **less effective** than max pooling

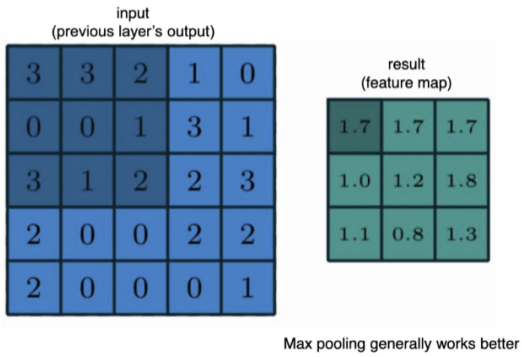


Figure 3.23: Average Pooling

Definition

Average Pooling: a pooling operator used to downsample the spatial dimensions of input data. It involves **dividing the input into non-overlapping regions and calculating the average value** of the elements within each region.

Strided Convolutions

- More recently, people are doing away with pooling operations, using strided convolutions instead

- Shift the kernel by s (eg. s = 2) when computing convolution

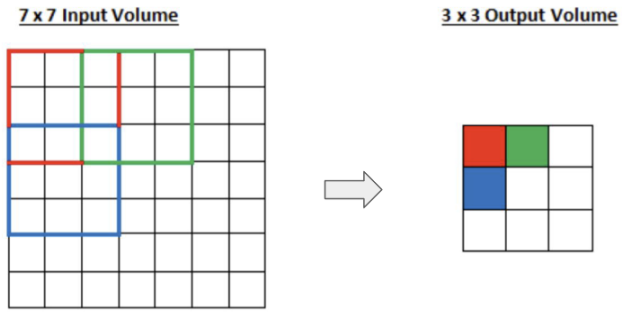


Figure 3.24: Strided Convolutions

Definition

Strided Convolutions: a downsampling technique that combines convolution and pooling. It involves **applying convolutional operations with larger strides** (spacing between filter placements) **than usual**, resulting in reduced spatial dimensions and aggregated feature representation.

Idea

As we go through a CNN network layer by layer:

- The filter **depth increases** (as the number of kernels increases)
  - The kernels closer to the input learn low level features
    - Due to the hierarchical nature of information, the number of kernels must increase to extract more detailed (high-level) features
    - That being said, kernel size generally is fixed
- The feature map height and width decreases

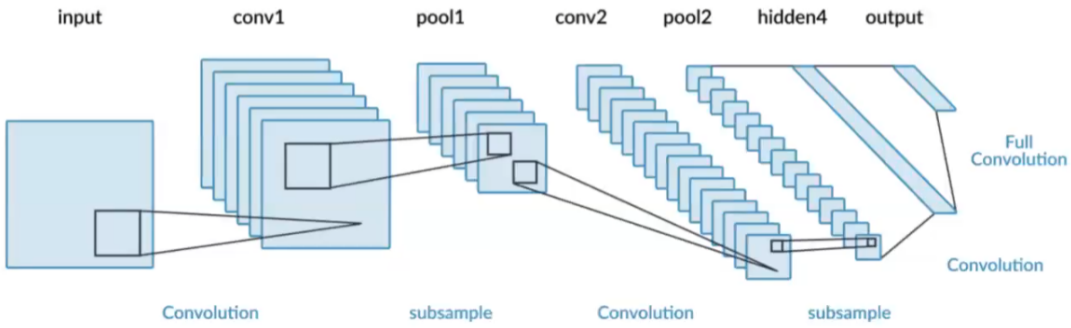


Figure 3.25: CNN

3.7 PyTorch Implementation

Example

Syntax for defining a convolutional layer:

```
self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
```

in_channels and out_channels are int. kernel_size, stride, padding may be either int tuple. By default, stride is 1 and padding is 0.

Example

Syntax for defining a pooling layer:

Max Pool:

```
self.pool = nn.MaxPool2d(kernel_size, stride)
```

Average Pool:

```
self.pool = nn.AvgPool2d(kernel_size, stride)
```

```
class LargeNet(nn.Module):
    def __init__(self):
        super(LargeNet, self).__init__()
        self.name = "large"
        self.conv1 = nn.Conv2d(3, 5, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(5, 10, 5)
        self.fc1 = nn.Linear(10 * 5 * 5, 32)
        self.fc2 = nn.Linear(32,10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 10 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Figure 3.26: Example CNN in PyTorch

3.8 Visualizing Convolutional Filters

What do CNN Filters/Feature maps look like?

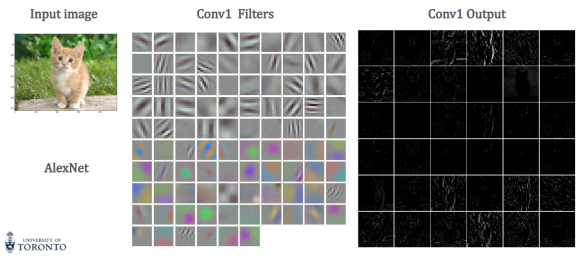


Figure 3.27: Example of a CNN filter

What features do CNNs learn?

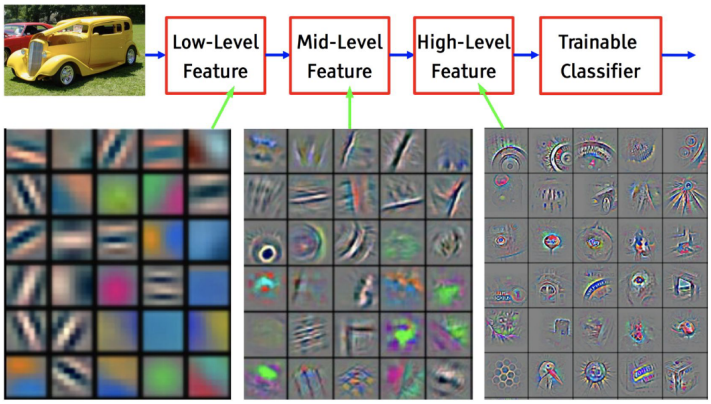


Figure 3.28: Learned Features

Definition

Saliency Maps: are visual representations highlighting the most important and relevant regions or features within an image, indicating where the human visual attention is likely to be drawn.

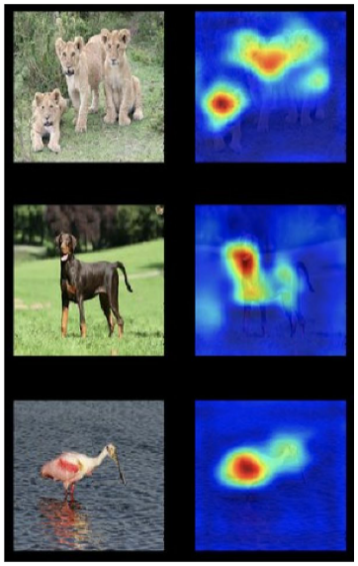


Figure 3.29: Saliency Maps

1. Feed the image to the network
2. Compute the gradients back to the input image
3. Take the maximum value of absolute gradients across channels
4. Visualize

- Saliency maps use gradients of the output over the input to highlight the areas of the images which are relevant for the classification
- Unfortunately outside giving some intuition, these are not practically very useful, and sometimes even misleading
- However, they can be used as a debugging tool to check if model is looking at the right features

3.9 Pre-Deep Learning Era

LeNet

- Convolutional Neural Networks were first introduced by Yann LeCun in 1989
  - Based on earlier ”Neocognitron”, but little used model (Fukushima, 1980)
- Several variants, mostly we refer to LeNet-5 (above, 1998)
  - 7 layers total, 2 convolutional, 2 pooling, 3 fully-connected

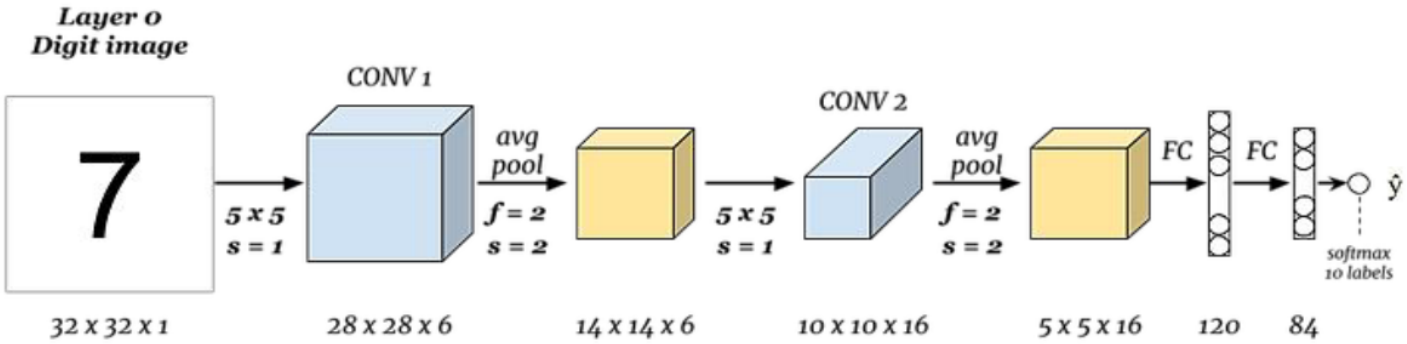


Figure 3.30: LeNet CNN

- Due to heavy hardware constraints at the time, training this network was a pain in the ass. However, once it was working, several useful properties were observed:

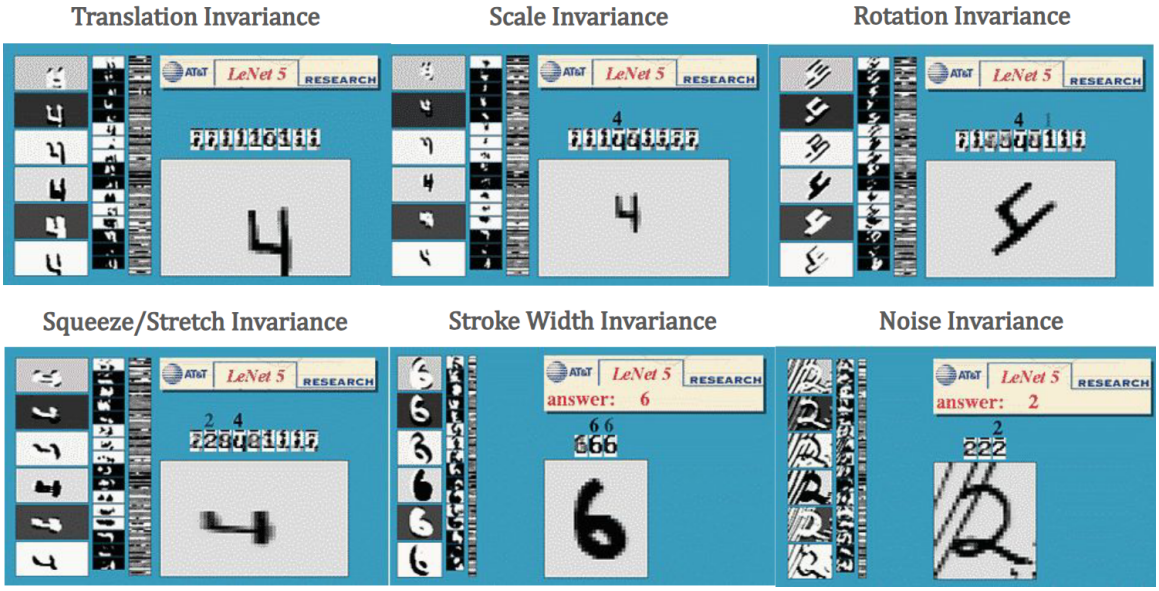


Figure 3.31: LeNet Properties

With PyTorch, implementation of this model is simple.

```
class LeNet5(nn.Module):
    def __init__(self):
        super(LeCun, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(5 * 5 * 16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 5 * 5 * 16)
        x = F.tanh(self.fc1(x))
        x = F.tanh(self.fc2(x))
        x = self.fc3(x)
        return x
```

Figure 3.32: PyTorch Implementation of LeNet

On the Eve of Deep Learning:

- We took a break in the mid-90s! ... let's skip forward to 2010
- Visual Object Classification is holy grail of Computer Vision
  - Classification of object is in an image, e.g. dog v.s. cat
- Best solutions to Object Classification is based on **Deformable Parts Models**
- CNNs Are outperformed on most tasks by using hand-crafted computer vision features (algorithms), and other ML classifiers, e.g. random forests (decision trees) or SVM

An intuitive way of recognizing a face is searching for facial features in an image. The problem with this idea is that if we look for face parts, we will identify a face even if the face parts are in the wrong locations.

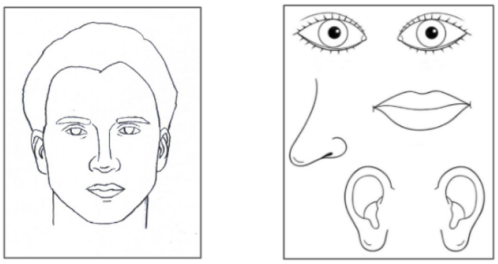


Figure 3.33: Looking for facial features alone isn't good enough

Deformable Parts Models

Definition

Deformable Parts Models: a class of object detection models in computer vision that incorporate flexible part structures within an object to improve accuracy in recognizing complex objects with varying appearances and poses.

- Recognize using parts and locations of parts
- Allow some deformation of part location, as there are difference in facial structure among different populations
- However, this is hard to extend to many different types of objects!

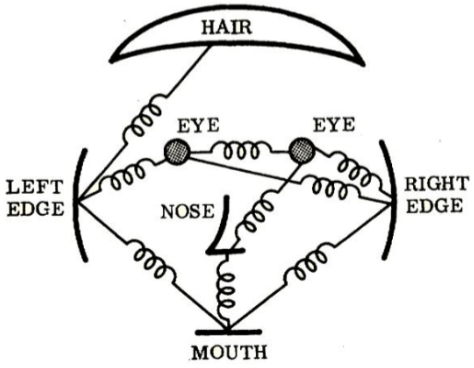


Figure 3.34: Deformable Parts Models

- Also doesn't work well for different viewing angles

Idea

Intuition as to how CNNs work is based on Deformable Parts Models

3.10 Modern Architectures

ImageNet Large Scale Visual Recognition Challenge

Definition

ImageNet Large Scale Visual Recognition Challenge (ILSVRC): is an annual competition in computer vision where participants develop and train machine learning models to classify and detect objects within a large dataset of images, known as ImageNet, containing thousands of categories.

- Pascal VOC was around 20,000 images with 20 classes (2006 - 2009)
  - Around the 2010s, this was the largest dataset available
- ImageNet was the first large-scale image dataset (14 million images)
- ILSVRC dataset based on ImageNet
  - 1 Million training images
  - 1000 different classes!
  - 50k validation, test set (never released)
- When we say ImageNet, we mean ILSVRC

Figure 3.35: Image Net

- ILSVRC challenge ran from 2010.
- Like Pascal VOC, every year saw the new winner improve accuracy by 1-2%
- CNN entry (AlexNet) in 2012 improved accuracy over previous year by 10%
- This is when ”Deep Learning” began

AlexNet

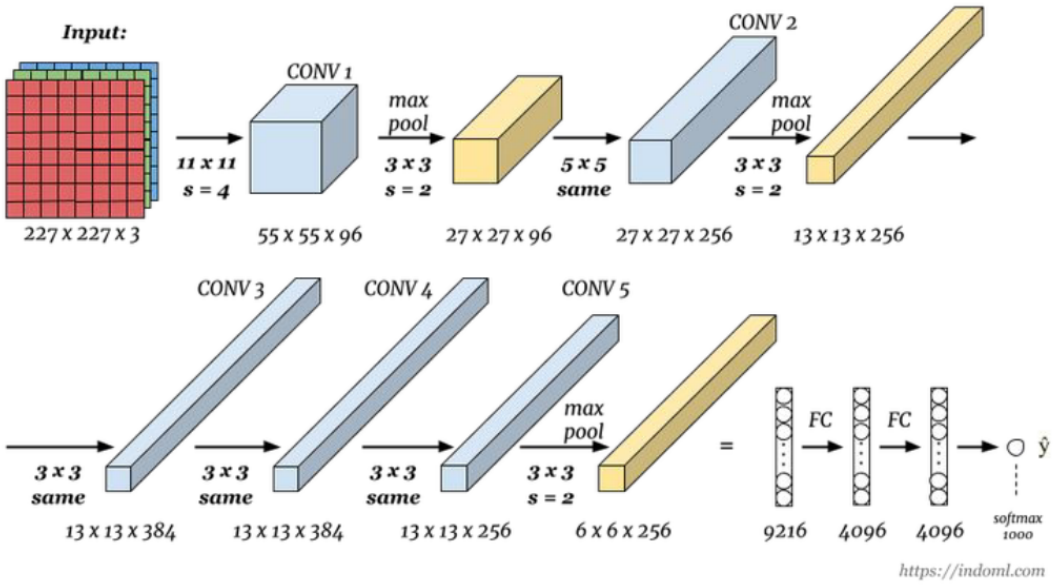


Figure 3.36: AlexNet Architecture

Deep Learning is differentiated from vanilla Neural Networks mostly in the changes between LeNet-5 and AlexNet:

- Much larger training datasets (e.g. ImageNet)
- Vast increase in compute/GPU acceleration (imagine 1989 PC v.s. 2012!)
- Much larger model size/more layers, enabled by both of the above!

AlexNet Training/Architecture Improvements:

- Large number of convolutional layers (i.e. deeper model)
- Use ReLU activation functions instead of sigmoids
- Dropout, data augmentation

About AlexNet Training:

- ∼60 Million parameters!
- Used GPUs to accelerate compute: 2 Nvidia GTX 580 GPUs
- 5-6 days to train over 90 epochs
- Optimized with SGD + Momentum
- Uses weight decay, dropout & data augmentation to improve generalization
- Learning rate schedule decreased learning rate 3 times over training

Data Augmentation

Definition

Data Augmentation: refers to the technique of artificially increasing the diversity of a dataset used for training machine learning models by applying various transformations, modifications, or perturbations to the original data, resulting in augmented samples that enhance the model’s ability to generalize and perform better on unseen data.

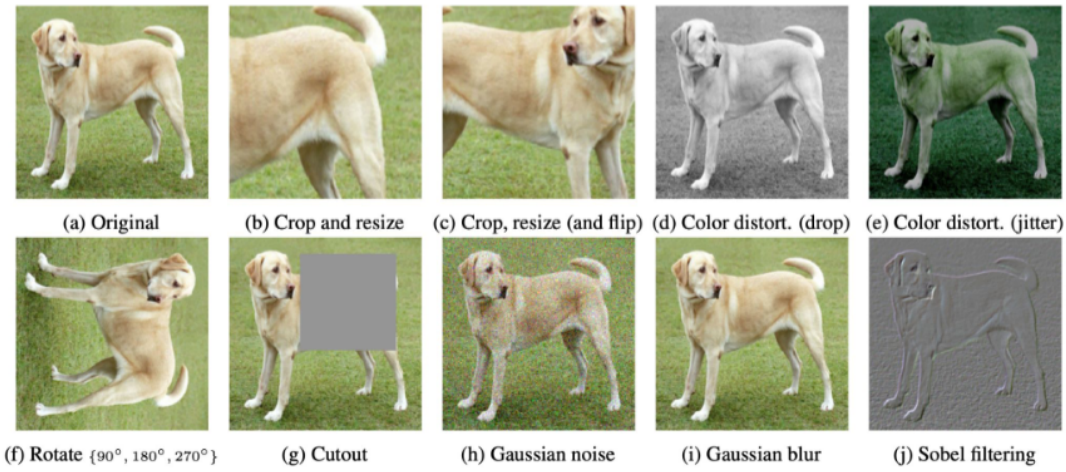


Figure 3.37: Ways to apply Data Augmentation

- Apply class-preserving transformations to the input
- Increases training data
- Helps generalization by learning internal representation of transformations
  - Helps the model learn the appearance in different situations (eg. low lighting, different weather conditions)
- Used by AlexNet (and all other CNNs)

Generalization and Depth

AlexNet and following models showed that increased depth improved generalization on ILSVRC, and other tasks. However, training very deep models often failed! This was due to **vanishing or exploding gradients**. The most important improvements in the past 10 years have been to address this:

- Improved initialization for ReLUs
- Normalization (e.g. Batch Normalization)
- Residual connections

GoogleLeNet (Inception)

People were catching on to the idea that **increased depth improved generalization**. This was the main motivation behind GoogleLeNet.

- 2014 ILSVRC winner, 6.67% Top-5 error
  - Human gets 5.1%
- Primary motivation was to go deeper
  - 22 convolutional layers
- Much more parameter efficient than AlexNet
  - 4 million vs. 60 million for AlexNet

Rather than trying to figure out the optimal kernel size for each layer, they said, ”why not **use all of them at the same time**?”

- The inception block uses a mixture of 3x3, 5x5 and 7x7 filters on one layer
- We don’t need large 7x7 (or 11x11 as in AlexNet) to learn most important filters.
- We can use mostly 3x3, and add a few larger filters

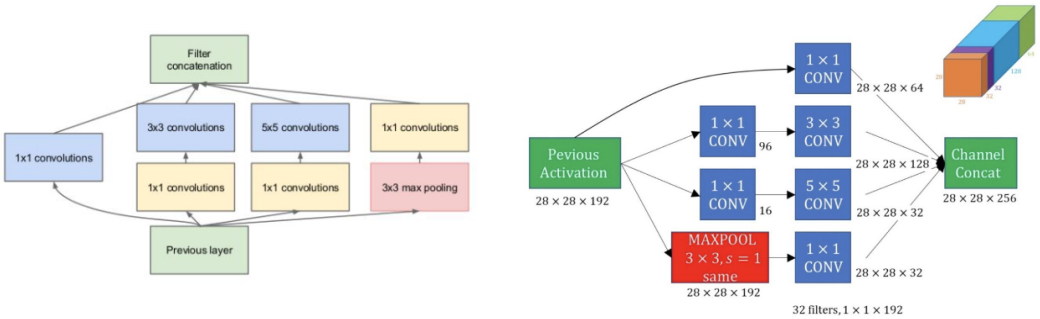


Figure 3.38: Ensemble Learning

Interestingly, they used 1 x 1 convolution, which means each kernel is applied to one pixel.

Pointwise (1x1) Convolution

- **Pixel-wise linear transformations!**
  - Originally used in a model called "Network-in-Network"
  - With a non-linearity, they are non-linear pixel-wise transformations
- Learn to map CNN feature maps into a lower or higher dimensional space
  - Good for learning compact representations/compression
- Efficiently **control the depths** of the network in different layers **while maintaining resolution**
- Used in all modern CNN architectures (except VGG)

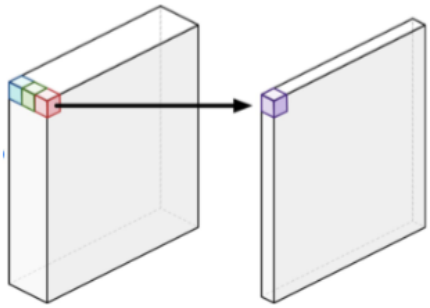


Figure 3.39: Pointwise Convolution

Using pointwise convolution, they managed to reduce the number of parameters to a relatively small number.

Auxiliary Loss

Definition

Auxiliary Loss: refers to an **additional loss function** incorporated into a model's training process alongside the main objective. It aims to provide auxiliary supervision to intermediate layers, helping the network learn relevant features and improve convergence.

- **Inception network is pretty deep** → subject to the vanishing gradient problem.
- **Solution** → intermediate classifiers
- Adding classifiers in the intermediate layers such that the final loss is a combination of the intermediate losses and the final loss.

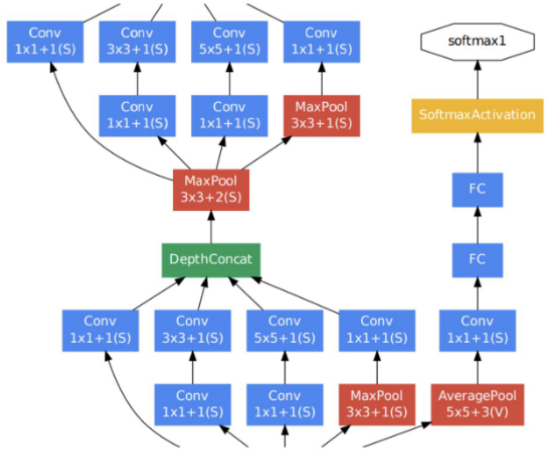


Figure 3.40: Auxiliary Loss

VGG (Visual Geometry Group, Oxford)

- 2014 ILSVRC classification 2nd place, 7.3% Top-5 error
  - However, won parallel ILSVRC localization challenge
- Proposed Models with 11, 13, 16, and 19 layers
- Very simple architecture, easy to understand/extend
- Very large number of parameters: 138 Million v.s. 60 Million for AlexNet!

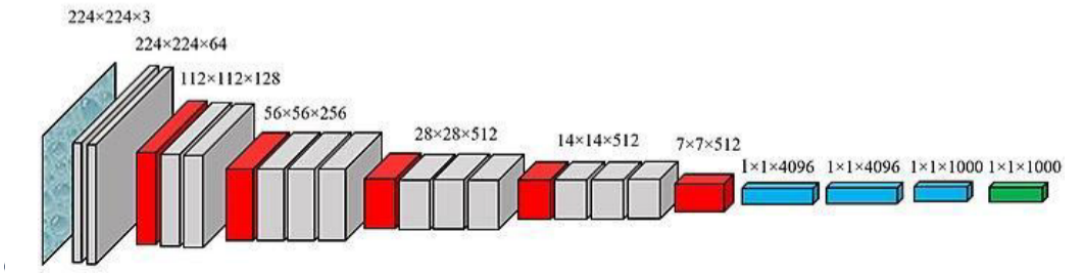


Figure 3.41: Visual Geometry Group

VGG was a very impactful paper:

- Simple architecture made of simple stacked blocks
- **We only need 3x3 filters** (set the modern standard)
  - Authors pointed out that stacked 3x3 filters can approximate any larger-sized convolution, more efficiently
  - Since VGG almost all CNNs use mostly/exclusively 3x3 filters!
  - **The data augmentation used by VGG is very commonly used**

Residual Networks (ResNet)

Even with Batch Normalization and ReLUs, training very deep networks fails. ResNet addresses this by using **skip connections (shortcuts)** to provide deeper layers with more direct access to signals that might otherwise be lost due to vanishing gradients.

ResNet won ILSVRC 2015 with a 3.57% error rate.

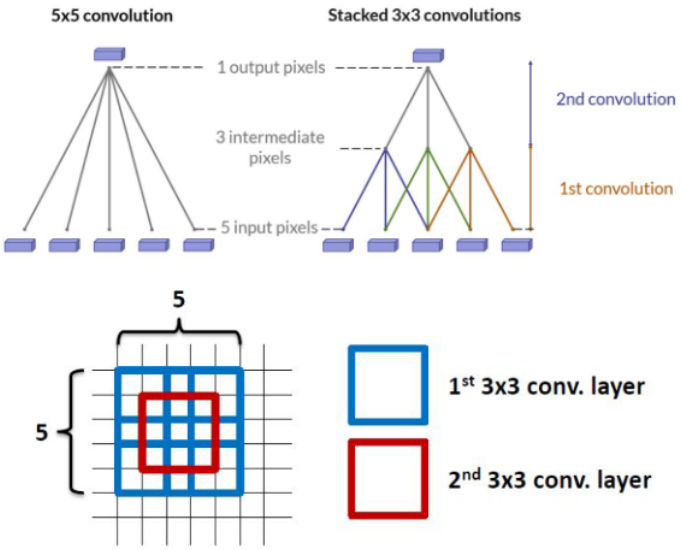


Figure 3.42: VGG's Simple Architecture

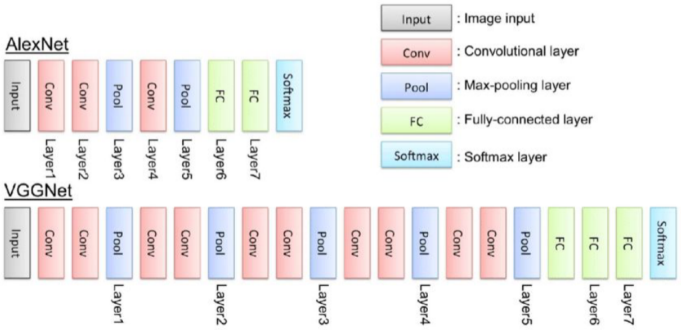


Figure 3.43: AlexNet vs. VGG

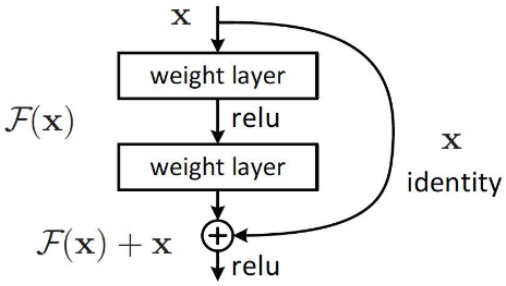


Figure 3.44: ResNet Skip Connections

- Model had 152 layers,
- Better than the Human baseline!

Definition

Skip Connections: a technique where the output of one layer or block of layers is combined with the output of a previous layer. This helps mitigate issues like vanishing gradients and aids in training deeper neural networks by allowing information to flow more directly across layers.

In this diagram, we notice the following:

```
# normal layer:
next_activation = layer(activation)

# residual layer
next_activation = activation + layer(activation)
```

Figure 3.45: PyTorch Implementation of Skip Connections

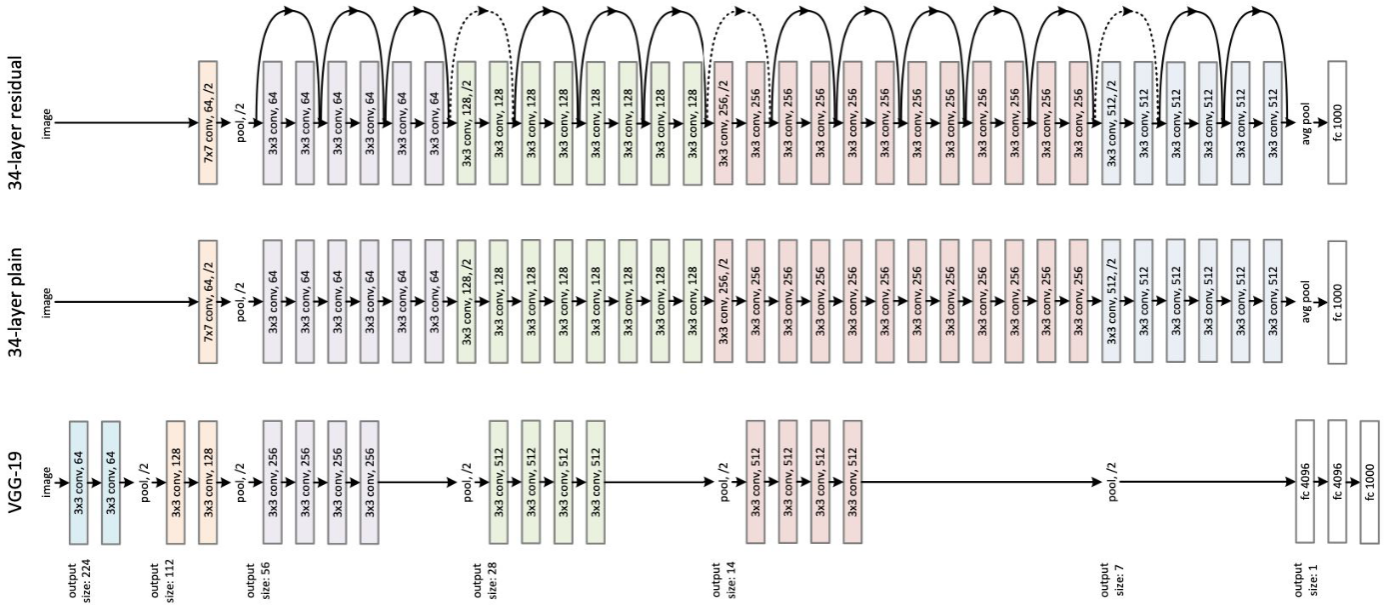


Figure 3.46: ResNet compared to others

- **Residual blocks** (multiple convolutions with skip connections)
- **Downsampling using stride 2**, instead of max/avg pooling (strided convolution)
- **Global average pooling** after last convolutional layers (introduced by Network-in-Network)
  - Means that the embedding has no spatial dimension and is only 512 floats!
- Only **a single fully-connected classification layer**
  - learned embeddings are so good we don't need a complex classifier at end of model
  - allows variable input sizes because it's not constrained by FC layer dimensions

Post-ILVRC

- The last ILVRC competition was held in 2017
- ResNets are still the most popular architecture for a wide variety of problems
- Object recognition in-the-wild is still not solved
  - Not all edge cases can be recognized yet

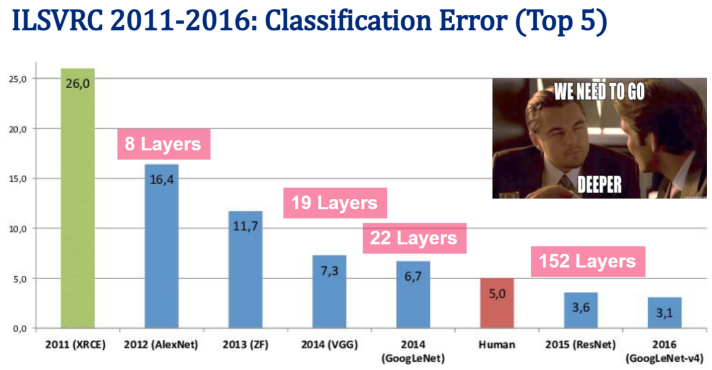


Figure 3.47: ILSVRC Models Ranked

3.11 Transfer Learning

Definition

Transfer Learning: approach where a model trained on one task is re-purposed or fine-tuned for a different but related task. This leverages knowledge gained from the source task to improve learning efficiency and performance on the target task, particularly when data is limited.

Generally, CNN models contain two distinct parts:

- **Convolutional Layers** → Learn filters across spatial and channel dimensions **(encoder)**
- **Fully-connected Layers** → Learn to classify images based on the learned visual features **(classifier)**

The point at which these parts intersect is an **embedding**, which is a learned lower-dimensional set of ”visual feature” representing the image. This embedding encodes everything needed from the image to classify objects!
This means that **everything before the embedding is a general feature extraction that is universal** and can be applied to any model. The only specialized part of the model is the classifier, since it is unique to each problem.

Idea

The larger the encoder, the more universal it is.

Definition

Embeddings: the process of representing objects, such as words, images, or entities, in a lower-dimensional space where their relationships and characteristics are preserved. These representations capture semantic or contextual information, making them suitable for various tasks like similarity measurement, classification, and recommendation.

- These CNN embeddings have proven useful for solving a wide variety of image-based problems
- By being trained on a large image classification dataset, CNNs learn something general about representing images!

We can use these features to transfer our learning to a new problem:

1. Train CNN (e.g. AlexNet) on large image dataset (e.g. ImageNet)
2. Remove ”classification” layers at end of model, freeze remaining weights.
3. Add, and train, new layers at end of model suitable for our new task.

- We froze the original model’s weights, and used our CNN layers as a feature extractor
- Often training some/all of the original model’s weights on the new task at a lower learning rate helps the features ”adapt” to the new task
- This, and variants of it, is often referred to as **fine-tuning**

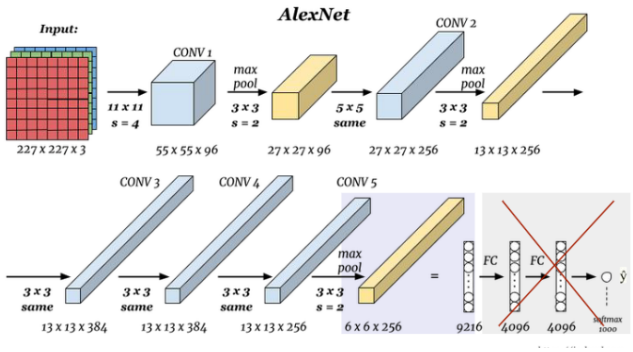


Figure 3.48: Transfer Learning with AlexNet

Idea

Transfer learning also prevents overfitting because the architectures and weights of networks like AlexNet were trained using a larger dataset of over a million images, and were trained to solve a different image classification problem. Nevertheless, transfer learning allows us to leverage information from larger data sets with low computational cost. Effectively acting like a larger training set.

4 Autoencoders

4.1 Motivation

All algorithms up to this point have **required data with ground truth and utilized supervised learning**. This **inductive bias** has come with a list of challenges:

- **Requires large amounts of labeled data**
- **Obtaining labeled data is expensive** → people need to be hired to label data
- **Medical tests are expensive** → require a specialist to review them
- **Chemical data collection** → wet-lab tests are time consuming
- Often there is a lot **more unlabeled data than labeled** → the internet has vast amount of unlabelled data and it would be nice to utilize it
- **Not what we see in biology**

Babies learn by recognizing patterns **without explicit supervisory signals**. This is the idea behind unsupervised learning. Basically, we don't need to know the name of objects to know that they're similar.

Unsupervised Learning

- Our brains are constantly observing the world around us for patterns, or some structure to relate objects.
- Patterns or clusters of similar features can tell us a great deal about the data before we even have a label.

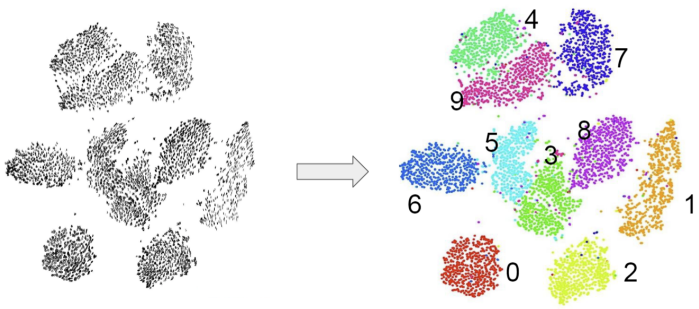


Figure 4.1: Feature Clustering

Definition

Unsupervised Learning: learning patterns from data without human annotations (e.g., clustering, density estimation, dimensionality reduction).

Definition

Self-supervised Learning: use the success of supervised learning without relying on human provided supervision (automatic supervision) (e.g., mask part of the input and predict the masked information).

Definition

Semi-supervised Learning: learning from data that mostly consists of unlabeled samples. A small amount of human-labeled data is available as well.

4.2 Autoencoders

Definition

Autoencoders: neural network architectures designed for unsupervised learning. They consist of an encoder and a decoder network. The encoder compresses input data into a lower-dimensional representation (encoding), while the decoder reconstructs the original data from this encoding. The goal is to learn an efficient representation of the data, typically for tasks like data compression, denoising, or dimensionality reduction.

Find efficient representations of input data that could be used to reconstruct the original input using two components:

- **Encoder**
  - Converts the inputs to an internal representation
  - Dimensionality reduction
- **Decoder**
  - Converts the internal representation to the outputs
  - Generative network

The number of outputs is the same as the inputs. **Hourglass shape creates a bottleneck layer**, lowering dimensional representation. Similar to ANNs and CNNs, we use a loss function to quantify reconstruction error.

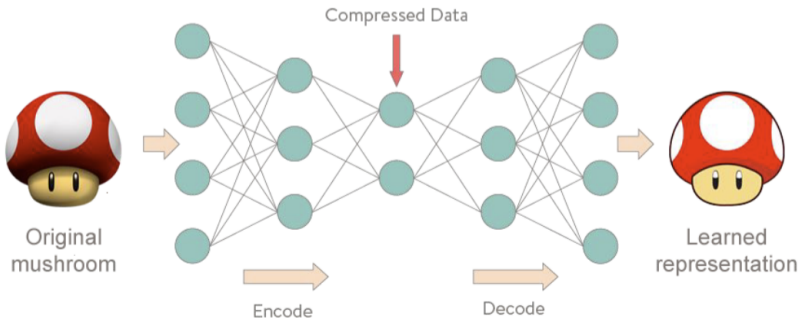


Figure 4.2: Hourglass shape of an Autoencoder

The autoencoder is forced to learn the most important features in the input data and drop the unimportant ones.

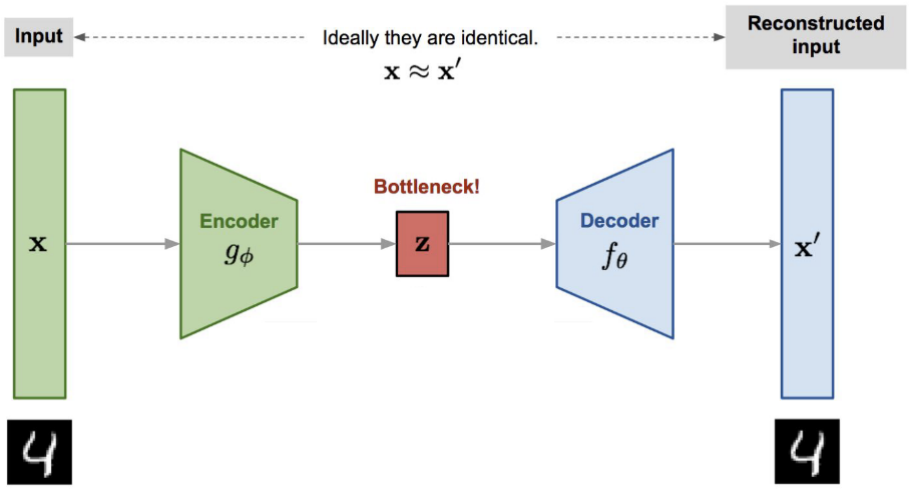


Figure 4.3: Autoencoder structure

Applications

- Feature Extraction
- Unsupervised Pre-training
- Dimensionality Reduction
- Generate new data
- Anomaly detection → Autoencoders are bad at reconstructing outliers

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        encoding_dim = 32
        self.encoder = nn.Linear(28 * 28, encoding_dim)
        self.decoder = nn.Linear(encoding_dim, 28 * 28)

    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        x = self.encoder(flattened)
        # sigmoid for scaling output from 0 to 1
        x = F.sigmoid(self.decoder(x))
        return x

    criterion = nn.MSELoss()
```

Figure 4.4: Simple PyTorch Implementation of an Autoencoder

Stacked Autoencoders

- Autoencoders can have multiple hidden layers: stacked (deep) autoencoders
- Typically symmetrical with regards to the central coding layer.

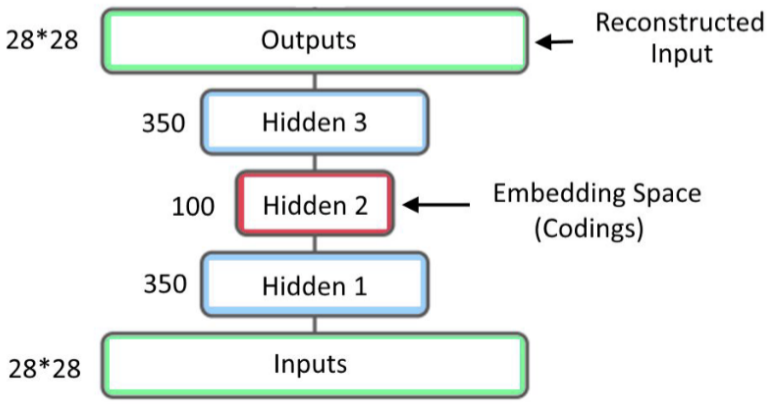


Figure 4.5: Stacked Autoencoders

Visualizing Reconstructions: a way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs.

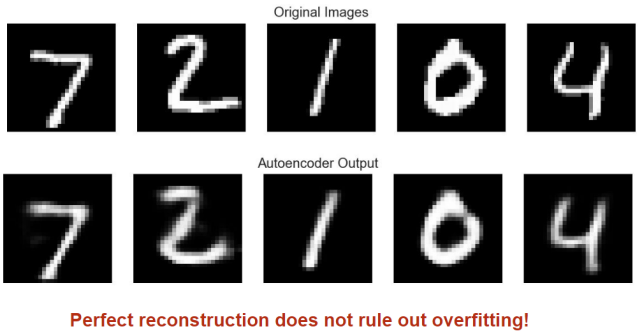


Figure 4.6: Visualizing autoencoder reconstructions

Idea

Autoencoders often have issues with overfitting. We can help mitigate this problem using denoising autoencoders.

Denoising Autoencoders

- Noise can be added to the input images of the autoencoder to force it to learn useful features (regularization)
- Autoencoder is trained to recover the original, noise-free inputs.
- Prevents it from trivially copying its inputs to its outputs, has to find patterns in the data.
- Two common ways:
  - Adding gaussian noise (salt and pepper effect)
  - Randomly masking inputs (dropout for pixels)

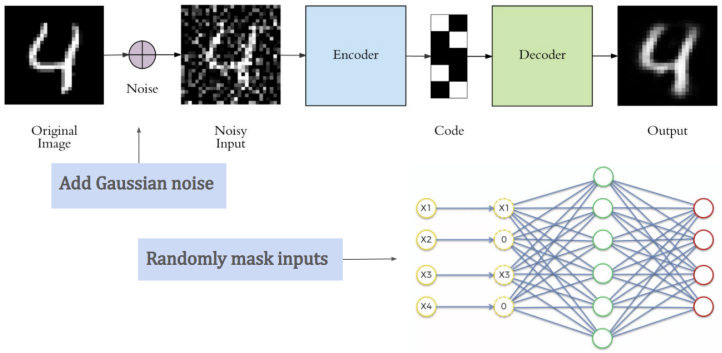


Figure 4.7: Ways to denoise autoencoders

```
# how much noise to add to images
nf = 0.4

# add random noise to the input images
noisy_img = img + nf * torch.randn(*img.shape)

# Clip the images to be between 0 and 1
noisy_img = np.clip(noisy_img, 0., 1.)

# compute predicted outputs using noisy_img
outputs = model(noisy_img)

# the target is the original img
loss = criterion(outputs, img)
```

Figure 4.8: PyTorch implementation of Gaussian noise

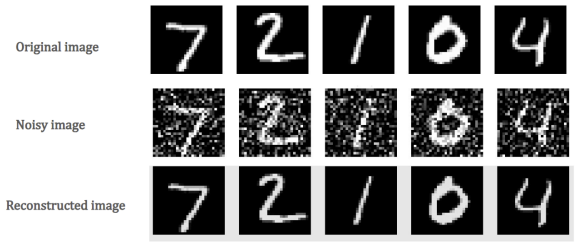


Figure 4.9: Visualization of adding Gaussian noise to autoencoders

Generating New Images

- Since we are drastically reducing the dimensionality of the image, there has to be some kind of structure in the codings (i.e. embedding space)
- That is, the network should be able to save space by mapping similar images to similar embeddings
- Let's see how we can exploit this to allow us to generate new types of images which can be used to create a more robust and vast dataset

New Images with Interpolation

1. First compute low-dimensional embeddings of two images.
2. Then interpolate between the two embeddings and decode those as well!
3. Interpolated codings result in new images that are somewhere in between the two starting images.

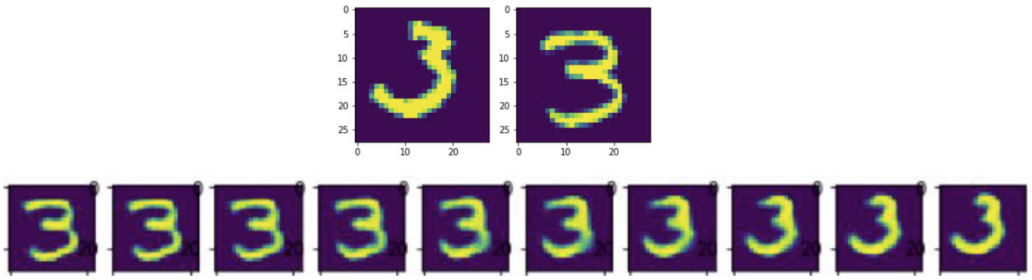


Figure 4.10: Combining embeddings using Interpolation

What if we randomly select a coding?: The latent space in autoencoders can become disjoint and non-continues (looks like the input but is actually nonsense). Very rarely, the random number might acrually generate a meaningul (actual) digit, but this is unlikely due to the high number of dimensions.

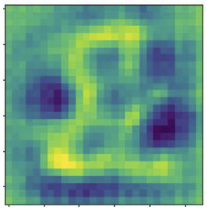


Figure 4.11: Randomly selecting a coding

Idea

Denoising autoencoders often have issues with the smoothness of the embedding space. We can help mitigate this problem using variational autoencoders.

4.3 Variational Autoencoders

Definition

Variational Autoencoders: probabilistic generative models that consist of an encoder and a decoder neural network. VAEs aim to learn a probabilistic mapping between the input data and a latent space, where data points are represented as probability distributions. Unlike traditional autoencoders, VAEs generate data points by sampling from these distributions, making them useful for generating new data samples and data generation tasks. VAEs are often used in tasks such as image generation, data synthesis, and data denoising.

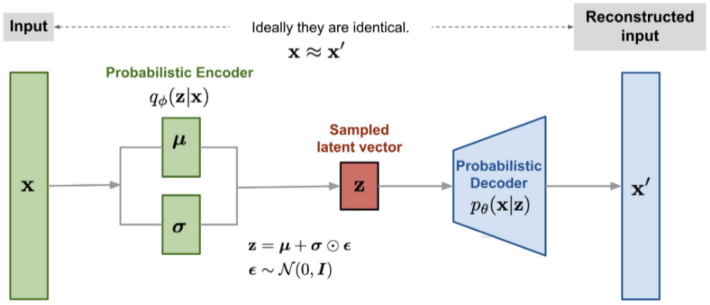


Figure 4.12: Variational autoencoder structure

They are quite different from the autoencoders we have discussed so far:

- **Probabilistic** $\rightarrow$ their outputs are partly determined by chance even after training (the same input will not always yield the same results every time)
- **Generative** $\rightarrow$ they can generate (an infinite number of) new instances that look like they were sampled from the training set, but are not the same as the training set.

They impose a distribution constraint on the latent space to have a smooth space.

Figure 4.13: Generated images that look like handwritten digits by training a variational autoencoder

Definition

Gaussian Distribution: also known as a normal distribution, is a probability distribution that is characterized by its bell-shaped curve. It is defined by two parameters: the mean (μ), which represents the center of the distribution, and the standard deviation (σ), which measures the spread or dispersion of the data. In a Gaussian distribution, data tends to cluster around the mean, with the majority of values close to the mean, and it follows a symmetrical pattern.

Encoder generates a normal distribution with mean μ and a standard deviation σ instead of a fixed embedding. An embedding is sampled from the distribution and decoder decodes the sample to reconstruct the input.

We want the encoder distribution $q_\phi(z|x) = N(\mu, \sigma)$ to be close prior $p(z) = N(0, I)$. We can use Kullback-Leibler (KL) divergence to measure the difference between two distributions P(X) and and Q(X):

$$D_{KL}(P||Q) = \sum_{x \in X} p(x) log(\frac{p(x)}{q(x)})$$

If we plug in the encoder distrubution and the prior KL-divergence of two multivariate Gaussians, we get:

$$D_{KL}(p|q) = \frac{1}{2}\sum_{i=1}^{N}[\mu_i^2 + \sigma_i^2 - (1 + log(\sigma_i^2))]$$

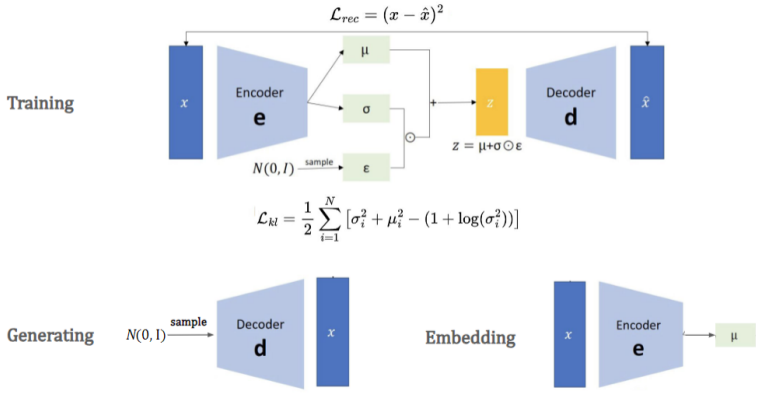


Figure 4.14: Variational autoencoders: training, generating, and embedding

Idea

Generating new images requires that we sample a latent vector from the unit Gaussian and pass it into the decoder. Variational autoencoders use probability distribution for embeddings such that the entirety of the embeddings subspace is valid input to the decoder. Basically, to ensure that the output from an encoder is not nonsense.

4.4 Convolutional Autoencoders

Idea

When it comes to dealing with images, convolution is much better than fully connected networks.

Definition

Convolutional Autoencoders: a type of artificial neural network designed for unsupervised learning that uses convolutional layers to encode and decode input data efficiently. It is primarily used for feature extraction and dimensionality reduction in tasks such as image reconstruction and denoising.

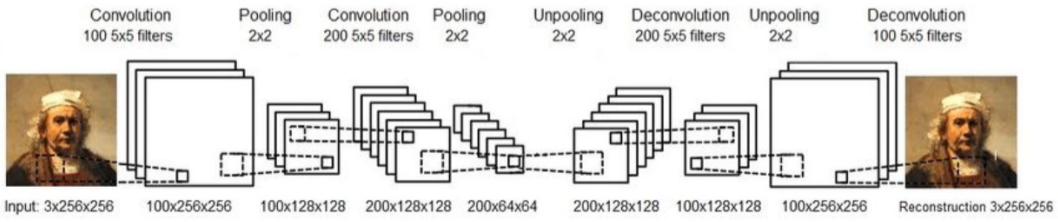


Figure 4.15: Convolutional autoencoder

Convolutional autoencoders take advantage of spatial information.

- **Encoder** → Learns visual embedding using convolutional layers
- **Decoder** → Up-samples the learned visual embedding to match the original size of the image.

Definition

Transposed Convolution: also known as ”deconvolutions” or ”up-sampling,” are operations in deep learning that expand the spatial dimensions of data, typically in convolutional neural networks, by using learnable filters to perform upsampling or interpolation, allowing the network to generate higher-resolution feature maps from lower-resolution input.

The opposite of the convolution is the **transposed convolution** (different from an inverse convolution). They work with filters, kernels, padding, strides just as the convolution layers. Instead of mapping KxK pixels to 1, they can map from 1 pixel to KxK pixels. The kernels are learned just like normal convolutional kernels.

Theorem

Computing the output size of the feature map (from transpose convolving an image with a kernel)

For each dimension of an input image with:

- Image dimension of size **i**
- Kernel of size **k**
- Padding of size **p**
- Output padding of size **op**
- Stride of size **s**

The size of output dimension is computed by:

$$o = (i-1)\cdot s + (k-1) - 2p + op + 1$$

To perform transposed convolution:

1. Take each pixel of your input image
2. Multiply each value of your kernel with the input pixel to get a weighted kernel
3. Insert it in the output to create an image

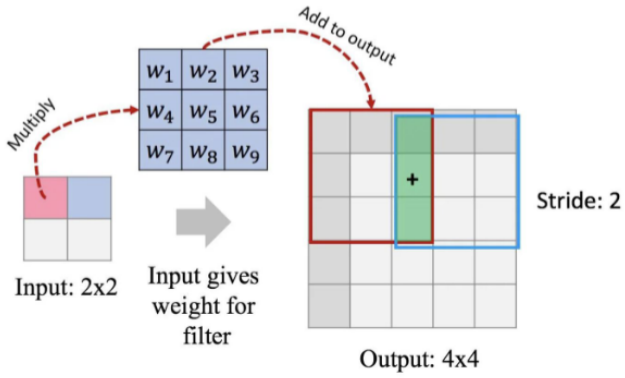


Figure 4.16: Transposed Convolution

4. Where the outputs overlap, sum them

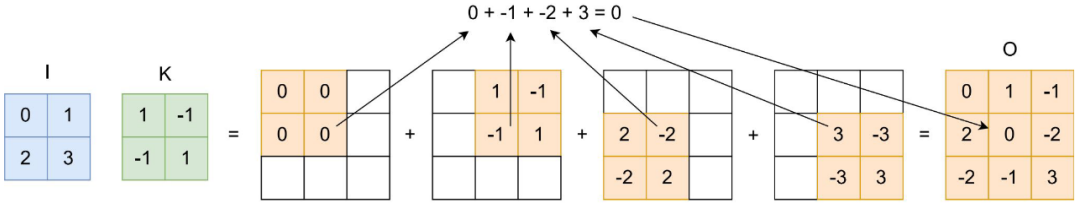


Figure 4.17: Transposed Convolution

Padding
The effect is the opposite of what happens with the convolution layers:

- Compute the output as normal
- Remove rows and columns around the perimeter

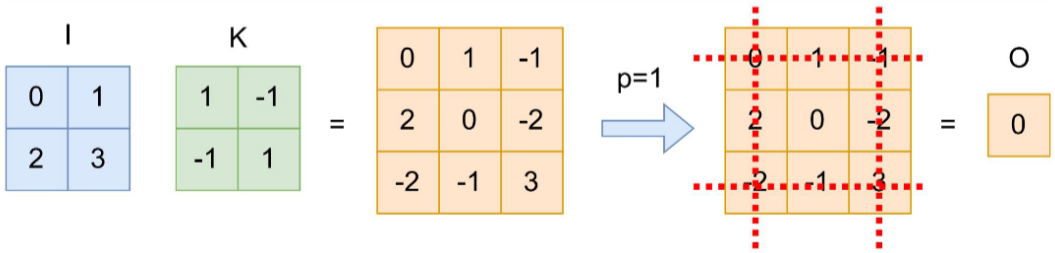


Figure 4.18: Padding in transposed convolution

Output Padding

- When stride > 1, `Conv2d` maps multiple input shapes to the same output shape.
- E.g. Inputs of size 7×7 and 8×8 both return an output of 3×3 for a kernel of size 3×3 with `stride`$= 2$.
- When applying the transpose convolution, it is ambiguous which output shape to return, 7×7 or 8×8 for `stride`$= 2$ transpose convolution.
- Output padding is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side.
- It is only used to find the output shape but does not actually add zero-padding to the output.

Strides

- The effect is also the opposite from what happens with the convolution layers
- Increasing the stride results in an increase in the upsampling effect.

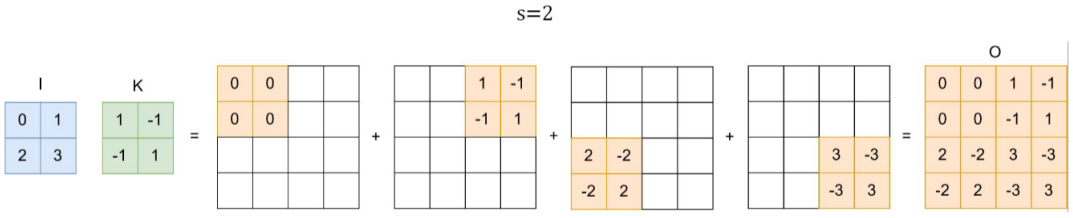


Figure 4.19: Strides in transposed convolution

A convolution transpose layer with the exact same specifications as the convolution layer would have the reverse effect on the shape.

```
conv = nn.Conv2d(in_channels=8,
                 out_channels=8,
                 kernel_size=5)

x = torch.randn(2, 8, 64, 64)
y = conv(x)
y.shape
```

```
torch.Size([2, 8, 60, 60])
```

```
convt = nn.ConvTranspose2d(in_channels=8,
                           out_channels=8,
                           kernel_size=5)

convt(y).shape # should be same as x.shape
```

```
torch.Size([2, 8, 64, 64])
```

Figure 4.20: PyTorch implementation of convolution and transposed convolution

We also have the option of including convolution transpose padding:

```
convt = nn.ConvTranspose2d(in_channels=16,
                           out_channels=8,
                           kernel_size=5,
                           padding=2)
x = torch.randn(32, 16, 64, 64)
y = convt(x)
y.shape
```

```
torch.Size([32, 8, 64, 64])
```

Figure 4.21: Transpose padding in PyTorch

We can add a stride to the convolution to increase our resolution!

```
convt = nn.ConvTranspose2d(in_channels=16,
                           out_channels=8,
                           kernel_size=5,
                           stride=2,
                           padding=2)

x = torch.randn(32, 16, 64, 64)
y = convt(x)
y.shape
```

```
torch.Size([32, 8, 127, 127])
```

Figure 4.22: Transpose stride in PyTorch

Output padding is another type of padding that adds an additional row and column to the output. It's easy to mix it up with padding.

```
convt = nn.ConvTranspose2d(in_channels=16,
                           out_channels=8,
                           kernel_size=5,
                           stride=2,
                           padding=2,
                          output_padding=1)

x = torch.randn(32, 16, 64, 64)
y = convt(x)
y.shape
```

```
torch.Size([32, 8, 128, 128])
```

Figure 4.23: Output padding in PyTorch

Idea

PyTorch Implementation of convolutional autoencoder:

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 7)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 7),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )

        def forward(self, x):
            x = self.encoder(x)
            x = self.decoder(x)
            return x

        def embed(self, x):
            return self.encoder(x)

        def decode(self, e):
        return self.decoder(e)
```

4.5 Pre-training with Autoencoders

Previously we discussed how transfer learning could use features obtained from ImageNet data to improve classification on other image tasks.

- Assumption that the ImageNet data is similar in the new task.
- If the new task is to detect new objects from similar images, then transfer learning makes sense.

Autoencoders can achieve similar results by pretraining on large set of unlabeled data, same type of data, just missing labels.

4.6 Self-Supervised Learning

What if we can cast unsupervised learning into supervised setting? **Define proxy supervised tasks such that:**

- The labels are generated automatically for free (utilizing advantage of no human input here)
- Solving the task, requires the model to “understand” the content

The challenge is devising the tasks such that they enforce the model to learn robust representations.

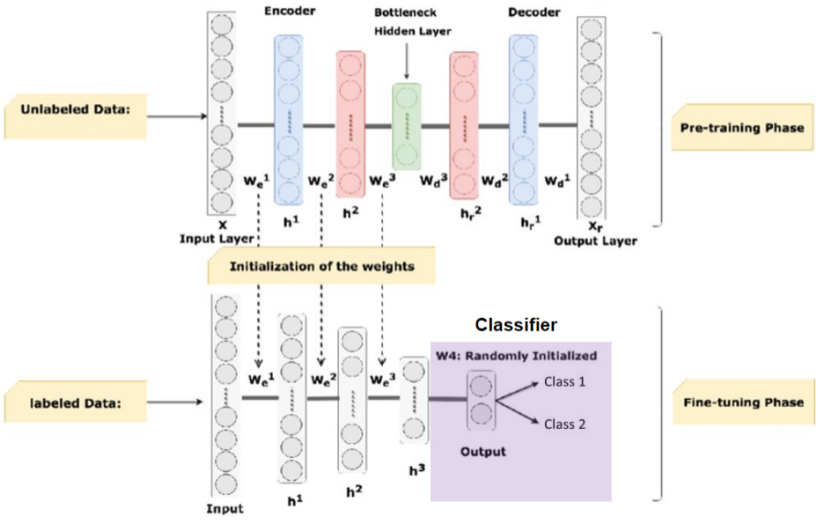


Figure 4.24: Pre-training Autoencoders

RotNet

Definition

RotNet: architecture designed for the task of image rotation recognition. It is trained to predict the rotation angle of an input image, typically in 90-degree increments (0°, 90°, 180°, or 270°). RotNet helps improve the robustness of machine learning models by enabling them to recognize objects in images regardless of their orientation, which can be useful in various computer vision applications.

Idea

Rotate images randomly by 0, 90, 180, or 270 degrees and make the model to predict the rotation angle.

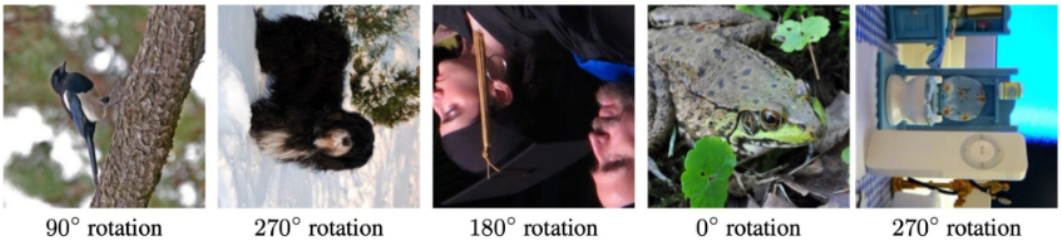


Figure 4.25: Rotating images and keeping rotations as ground truth labels assigned by program

If someone is not aware of the concepts of the objects depicted in the images, they cannot recognize the rotation that was applied to them.

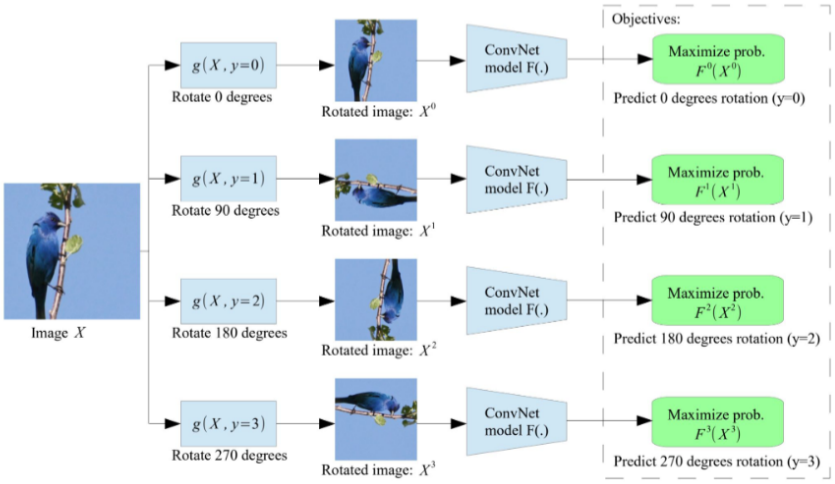


Figure 4.26: RotNet Multiclass Classification

The task is multi-class classification with 4 classes (therefore cross-entropy loss) with free labels being generated automatically. **The disadvantage is that the task is still human-generated and needs to be ”interesting” enough.** Otherwise, the network is going to cheat. Because of this, people moved to contrastive learning.

Contrastive Learning

Definition

Contrastive Learning: a self-supervised learning technique in machine learning and deep learning, where a neural network is trained to differentiate between pairs of data points, typically by maximizing the similarity (or minimizing the distance) between positive pairs (similar data points) and minimizing the similarity (or maximizing the distance) between negative pairs (dissimilar data points). This approach is often used for feature learning and representation learning, where it helps the network learn meaningful and discriminative representations of data without the need for labeled data.

Autoencoding Methods	Contrastive Learning
• Reconstruct input • Compute the loss in output space • Compress all the details	• Contrast pair of positive/negative samples • Compute the loss in embedding space • Compress relevant information • Requires lots of negative examples

Table 4.1: Comparison: Autoencoding vs. Contrastive Learning

Figure 4.27: Autoencoding vs. Contrastive Learning

SimCLR

Definition

SimCLR: short for ”Simultaneous Contrastive Learning,” is a self-supervised deep learning framework for learning powerful representations from unlabeled data. It is designed to encourage the model to pull together similar data points (positive pairs) while pushing apart dissimilar ones (negative pairs) in a high-dimensional feature space. SimCLR has been influential in achieving state-of-the-art results in various computer vision tasks by training on large datasets without the need for manual labeling.

- Augmentations of the same image are positive examples
- Augmentations of different images are negative examples
- **Same images are pushed together, different images are pushed away**

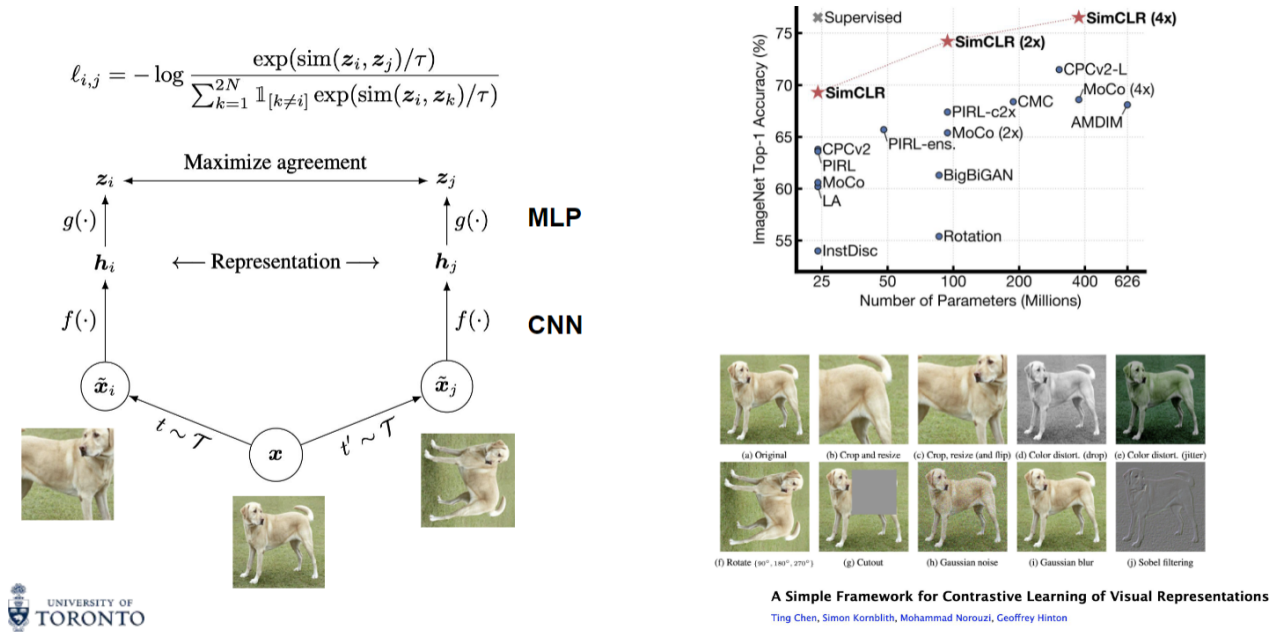


Figure 4.28: SimCLR

5 Recurrent Neural Networks (RNNs)

5.1 Motivation

Autoencoders can be used to learn an **embedding space**.

- **Encoder:** data $\rightarrow$ embedding
- **Decoder:** embedding $\rightarrow$ data

How can we learn **embedding of words**?

First, we need a way to convert words into numerical features, or in other words, **turn word features into vectors**.

One way to do this is to treat each word as a unique feature (**integer encoding**):

- E.g. cat = 0, dog = 1, or
- favourite color: red = 1, blue = 2, green = 3, etc.

Definition

Integer Encoding: technique in data preprocessing and natural language processing (NLP) that involves assigning a unique integer value to each distinct element or category in a dataset. This encoding simplifies the representation of categorical or nominal data, such as words in text or categories in a dataset, by replacing them with corresponding integer IDs.

The problem with this is that the **model might assume some relationship between classes solely based on the distance between indices**, while in reality, the numbers are assigned arbitrarily. When there is no specific order, integer encoding is not enough. Assuming an order may lead to **poor performance**.

A better way is to convert word features into numerical features with **one-hot encoding**.

- For example, at UofT, the categorical feature ”Term” can take on three possible values: Fall, Winter, Summer where:
  - Fall = [1, 0, 0]; Winter = [0, 1, 0]; Summer = [0, 0, 1]

Definition

One-hot Encoding: a data preprocessing technique used in machine learning and data analysis. It converts categorical data, such as discrete categories or labels, into a binary vector representation. Each category is represented as a binary vector where all elements are zero except for one, which corresponds to the category being ”hot” or ”on.”

- This is better than integer encoding in the way that there is no superficial relationship created between the words, however, there are two significant problems:

1. **Dimensions grow with the number of words**: eg. 10000 words means 10000 dimensional encoding!
2. **One-hot encoding assumes each word is completely independent**: cannot model relative similarities between words

5.2 Word Embeddings

How can we achieve word embedding?

- Words are different from images
- Characters are not like pixels in images
- The **meaning of a word** is not represented by the letters that make up the word
- Meaning comes from the sequence of characters and how they are used in conjunction with other words
- **Meaning comes from context**

Word Embedding models were created to address this problem.
The term Word embedding was coined in 2003 (Bengio et al.) Two commonly used models:

- **word2vec** model proposed in 2013 (Mikolov et al.)
- **GloVe** vectors released in 2014 (Pennington et al.)

Training Word Embeddings

- Encoder: word(??) $\rightarrow$ embedding
- Decoder: embedding $\rightarrow$???

Two things to consider:

1. How do we encode the word?
2. What is our target?

One-Hot Encoding of Words

- Each word has its own index
- If there are 10,000 words, there are 10,000 features
- "happy" $\rightarrow$ [0, 0, 0, 0, ... , 1, ... , 0, 0, 0, 0]
- One-hot embedding as input to the encoder
- Encoder: one-hot embedding $\rightarrow$ low dim embedding
- Decoder: low-dim embedding $\rightarrow$???

This is a possible solution to the encoding problem but what is our target?

Text as Sequences

> **Idea**
>
> The meaning of a word depends on its context, or other words that appear nearby.

There is evidence that children learn new words based on their surrounding words.

Architecture of word2vec

- Encoder: one-hot embedding $\rightarrow$ low-dim embedding
- Decoder: low-dim embedding $\rightarrow$ nearby words

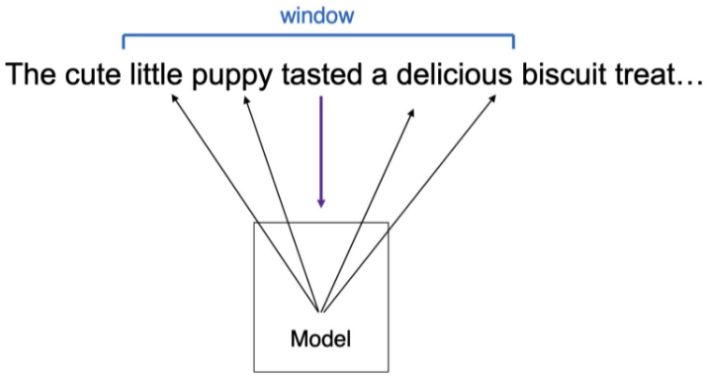


Figure 5.1: Architecture of word2vec model

Idea

The concept of a sliding window is similar to sliding a kernel across an image in CNN.

The middle word of the window is passed into the model as input and the model tries to predict the context around that word (surrounding words).

word2vec

Definition

word2vec: a family of architecture used to learn word embeddings.

- **Skip-Gram** → Predict context from target (center word as input, context, or surrounding words, as output)
- **CBOW (Continuous Bag of Words)** → Predict target from context (context as input, center word as output)

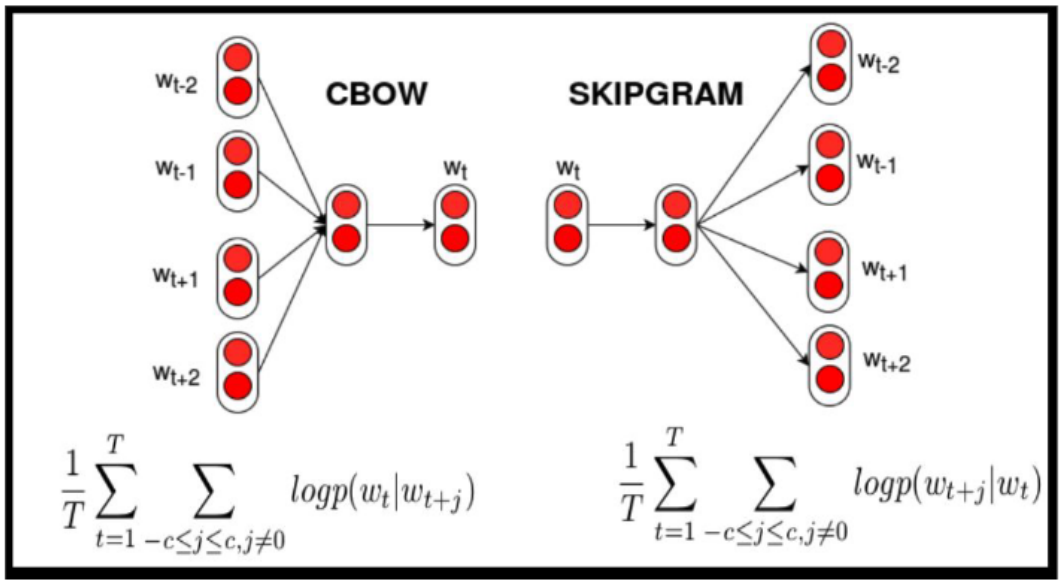


Figure 5.2: CBOW and Skipgram Architectures

Skip-Gram Model

- Predict context words from target words
- Components don't need to be consecutive in the text'
- Can be skipped over or randomly seleted from many documents

Definition

n-Gram: a contiguous sequence of n items from a given text.

Definition

k-Skip n-Gram: an n-gram that can involve a skip operation of size k or smaller.

Idea

Skip-Gram: Given a word predict its neighbouring words.

- The number of neighboring words are defined by the window size, which is a hyper-parameter.

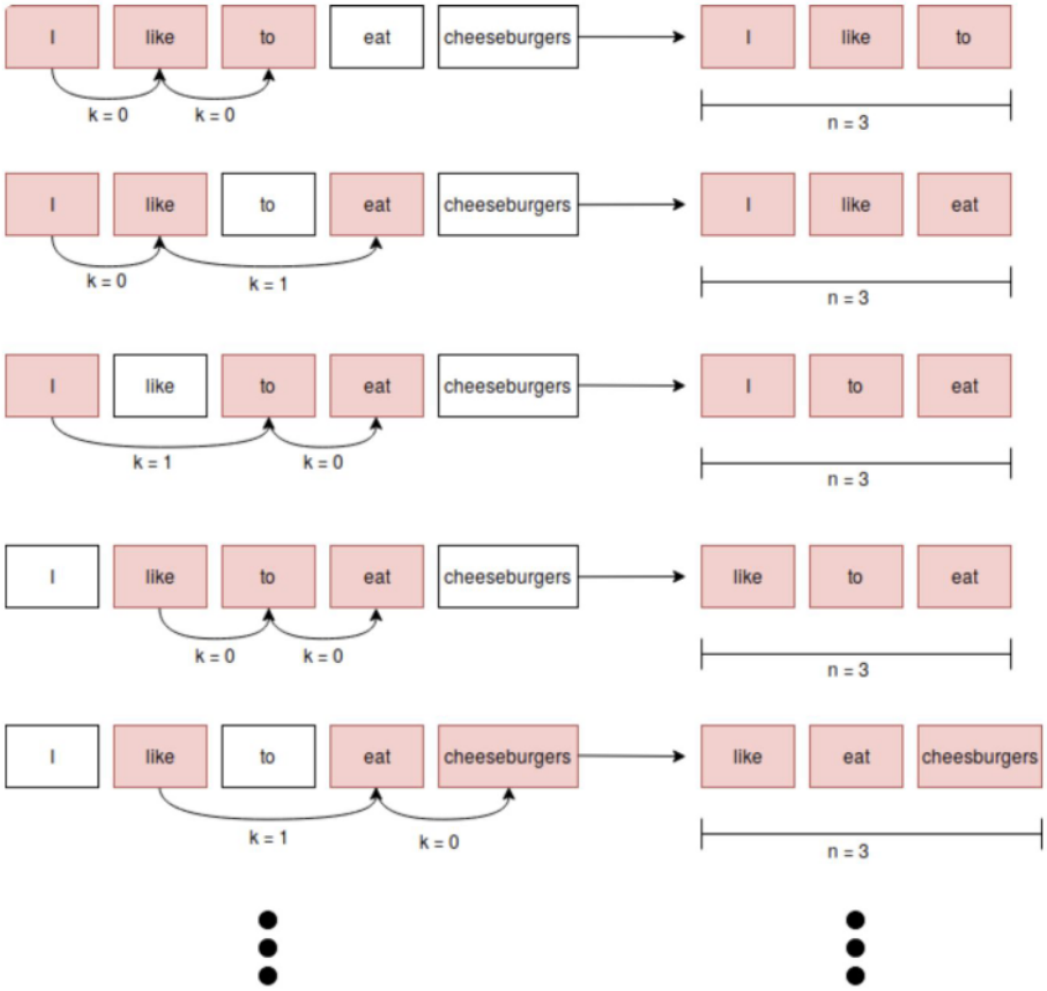


Figure 5.3: 1-Skip 3-Gram

Source Text	Training Samples generated from source text			
I will have orange juice and eggs for breakfast	(will, I)	(will, have)	(will, orange)	
I will have orange juice and eggs for breakfast	(have, I)	(have, will)	(have, orange)	(have, juice)
I will have orange juice and eggs for breakfast	(orange, will)	(orange, have)	(orange, juice)	(orange, and)
I will have orange juice and eggs for breakfast	(juice, have)	(juice, orange)	(juice, and)	(juice, eggs)
I will have orange juice and eggs for breakfast	(and, orange)	(and, juice)	(and, eggs)	(and, for)
I will have orange juice and eggs for breakfast	(eggs, juice)	(eggs, and)	(eggs, for)	(eggs, breakfast)
I will have orange juice and eggs for breakfast	(for, and)	(for, eggs)	(for, breakfast)	

Figure 5.4: Skip-Gram neighbouring words

- The output layer is only used for training
- After the model is trained, we only keep the weights from input to hidden layer
- Words that have similar context words will be mapped to similar embeddings

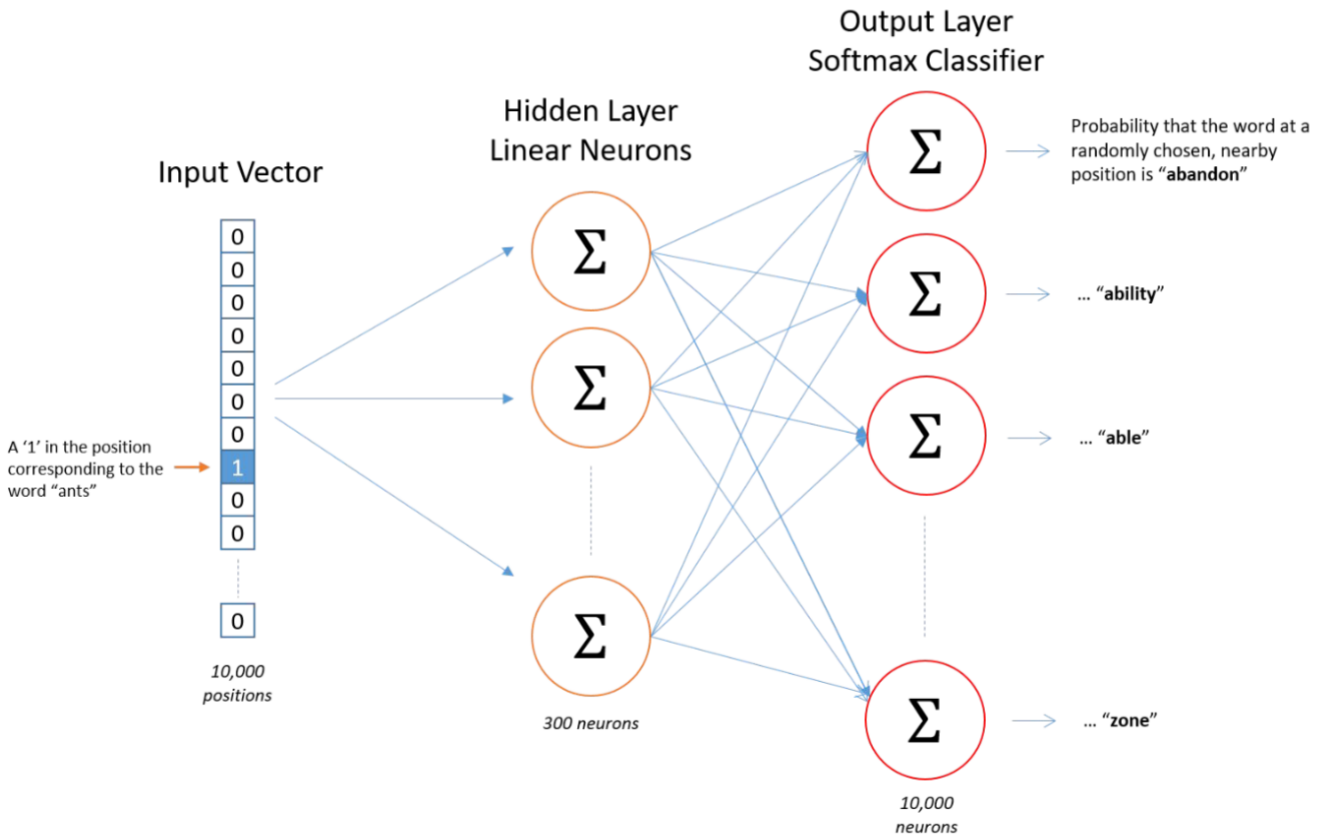


Figure 5.5: Skip-Gram Architecture

Continuous Bag of Words (CBOW) Model

- Predicts the center word from a fixed window size of context words
- Note that similar to SkipGram model, the input and output to the model are one-hot representation of pair of words

CBOW Vs Skip-Gram

Skip-Gram:

- Works well with small datasets
- Better semantic relationships (cat & dog)
- Better representation of less frequent words

CBOW:

- Trains faster than Skip-Gram as the task is simpler
- Better syntactic relationships (cat & cats)
- Better representation of more frequent words

Idea

The biggest limitation of word2vec is that the model is constrained to windows of text.

GloVe

Word2Vec does not have any explicit global information. GloVe enforces global information into the embeddings:

- Compute **co-occurrence frequency counts** for each word, represented as a matrix where element X ij denotes the number of times word i appears in the context of word j
  - Number of times when words are associated each other is counted
- **Optimization:** Inner product of word vectors should be a good predictor of co-occurrence frequency

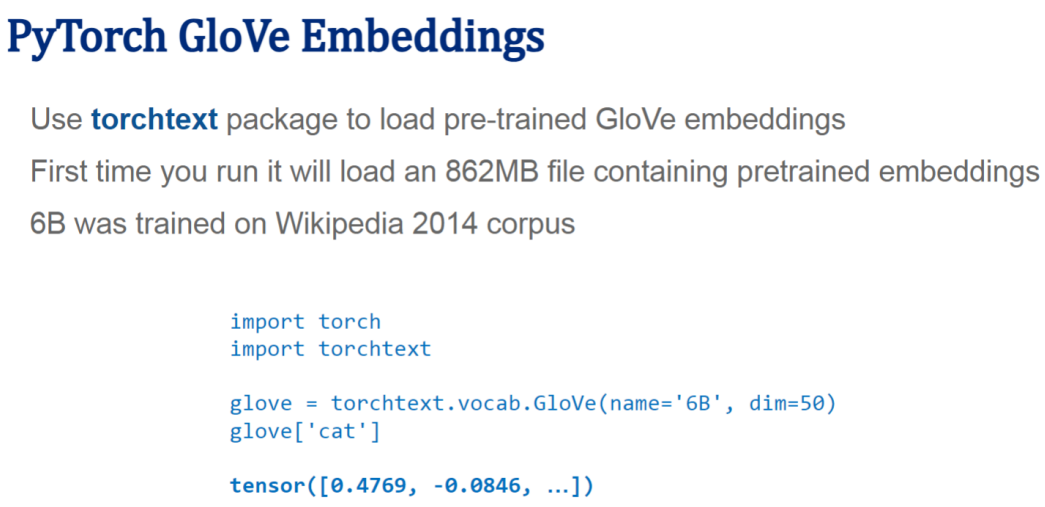


Figure 5.6: PyTorch GloVe Embeddings

Distance measures will allow us to measure the distance between embeddings.

5.3 Distance Measures

In order to talk about which words have similar embeddings, we need to introduce a measure of distance in the embedding space:

- **Euclidean Distance** → L2-norm of embeddings (sensitive to magnitude)

$$D(\vec{X},\vec{Y}) = ||\vec{X} - \vec{Y}|| = \sqrt{\sum_{i=0}^{d}(x_i - y_i)^2}$$

- **Cosine Similarity** → cosine of the angle between embeddings (invariant to magnitude)

$$Similarity(\vec{X},\vec{Y}) = cos(\theta) = \frac{\tilde{\mathbf{X}} \cdot \tilde{\mathbf{Y}}}{\|\tilde{\mathbf{X}}\| \cdot \|\tilde{\mathbf{Y}}\|} = \frac{\sum_{i=0}^{d} x_i \cdot y_i}{\sqrt{\sum_{i=0}^{d} x_i^2} \cdot \sqrt{\sum_{i=0}^{d} y_i^2}}$$

Euclidean Distance:

```
torch.norm(glove['cat']-glove['dog'])
```

Cosine Similarity:

```
torch.cosine_similarity(
      glove['cat'].unsqueeze(0), glove['dog'].unsqueeze(0))
```

Figure 5.7: Computing Distance in PyTorch

Word Analogies
One surprising thing about the embedding space is the extent of its structure.
We often see relationships like this in GloVe embeddings:

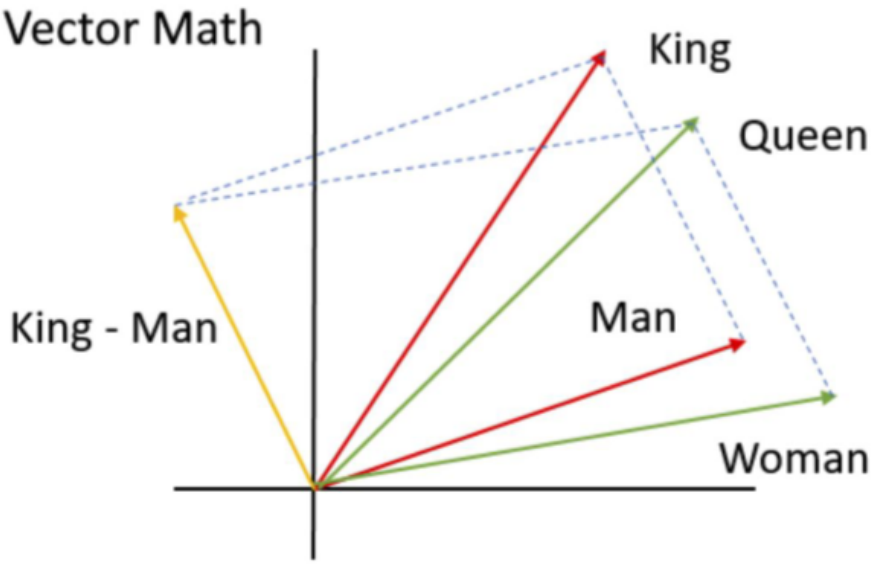


Figure 5.8: Showing bias of models

Idea

This word analogies show that machine learning models are not unbiased.

Warning

Machine learning models learn the biases present in the data it is trained on. The distance between embeddings is trained according to these biases.

5.4 Language Models

Definition

Language Models: neural network-based algorithms designed to understand and generate human language text. They use vast amounts of text data to learn the statistical patterns and relationships within language, enabling tasks such as text generation, translation, summarization, and sentiment analysis. These models have revolutionized natural language processing by providing powerful tools for various language-related tasks.

Learning probability distribution over sequences of words

- **Text Understanding**
  - Question Answering
  - Sentiment Analysis
- **Text Generation**
  - Sentence completion
  - Generating captions, sentences, stories. . .
  - Translation
- . . . and more!

How is working with text different (more challenging) from working with images?

- Grammar, spelling
- Many words to learn
- Choice of working with words vs characters
- Arbitrary length input / output
- Different languages

Sentiment Analysis

The goal of sentiment analysis is to identify the sentiment a text conveys. It can be applied to:

- Movie reviews
- Product feedback
- Emails
- Tweets

Dataset: Sentiment140

1,600,000 tweets collected by students doing a course project where sentiment determined by emoticon $\rightarrow$:) positive and :(negative. For each tweet in the training data, we will:

1. Split the tweet into words
2. Look up the GloVe embedding for each word, ignoring words that don't have embeddings
3. Add up the word embeddings to obtain an embedding for the entire tweet
4. The tweet embedding will be the input to a fully-connected neural network

Limitations: These two sentences will have the same embedding in our model:

- The food was adequate, but just not great
- The food was not just adequate, but great

Figure 5.9: Sentiment Analysis Fail

But they have drastically different meanings. Our model does not take into account the order of words; the summation operation causes the data to lose the order of the words. How can we do better?

Idea 1: Concatenate the word embeddings, then train a neural network that takes the concatenated embedding as input.

- While the order of the data is maintained, there are two main issues with this method:

1. Input to the fully-connected network will not be consistent
2. If we try to use padding techniques to problem 1, then we will have an extremely high number of parameters

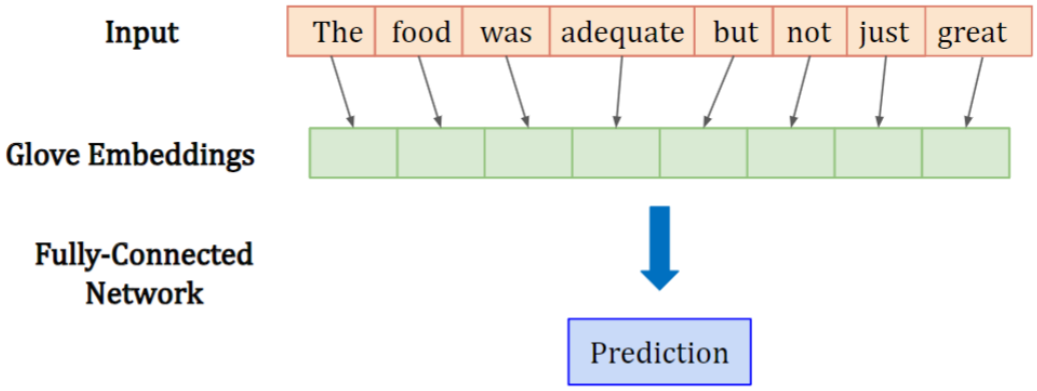


Figure 5.10: Sentiment140 Idea 1

Idea 2: Concatenate the word embeddings, then train a 1-dimensional convolutional neural network that takes the concatenated embedding as input.

- This addresses the sizing issue, however this presents a new issue that has to do with the nature of convolution.

- **Convolution is local**: this means that the model will miss the long range dependency that comes with sentences. Elements (words) that are away from each other in a string may still be highly dependent on each other for context.

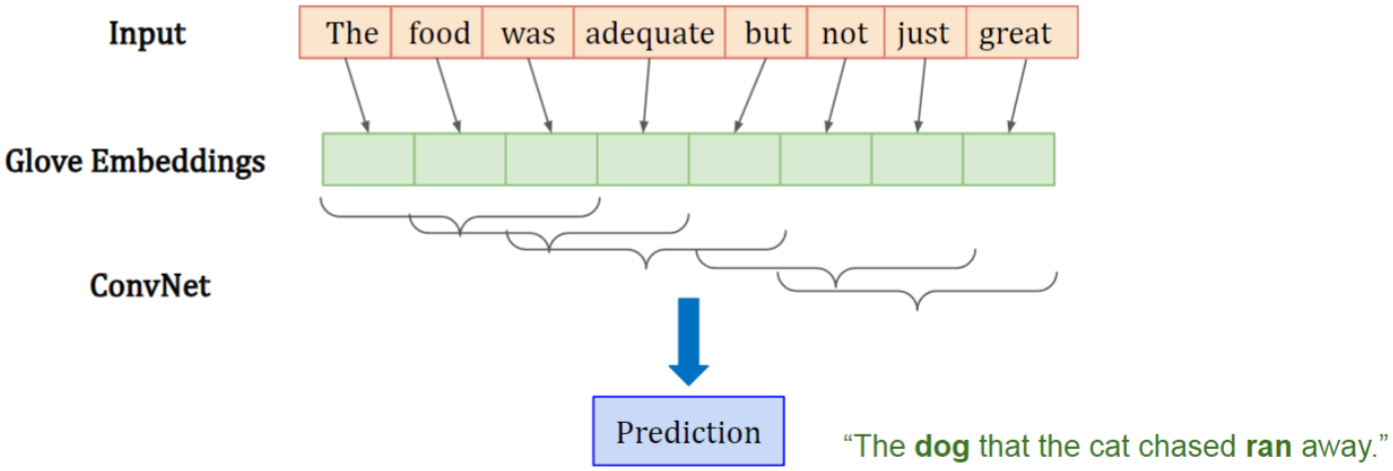


Figure 5.11: Sentiment140 Idea 2

We need recurrent neural networks to address these issues.

5.5 Recurrent Neural Networks

Idea 3: Use a recurrent neural network

- Can take in variable-sized sequential input
- Can remember things over time, or has some sort of **memory or state**

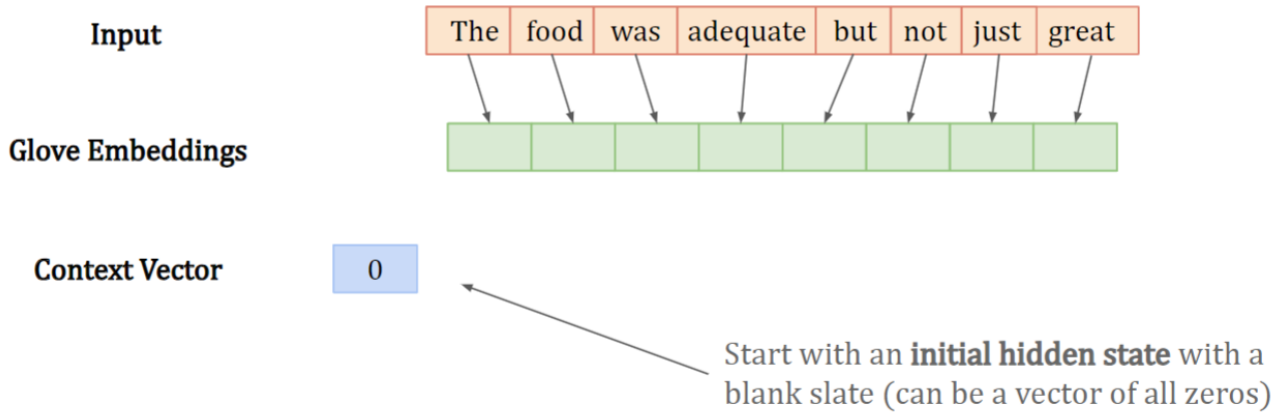


Figure 5.12: Sentiment140 Idea 3

Definition

Context Vector: a fixed-size representation that summarizes the information from the entire input sequence processed by the RNN. It captures the context or relevant information from past time steps and serves as an input to subsequent parts of the network, allowing the RNN to maintain memory and make predictions based on the entire sequence history.

Updating Hidden State: hidden state is updated based on previous hidden state and the input using the same neural network as before (weight sharing).

- The hidden state is updated until we run out of tokens.

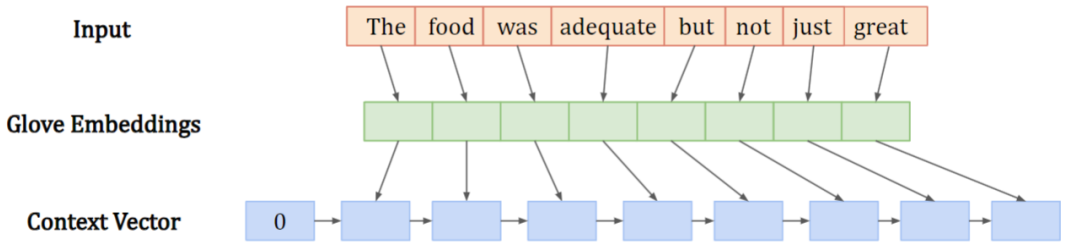


Figure 5.13: Updating hidden state

- The last hidden state is used as input to a prediction network (fully-connected network).

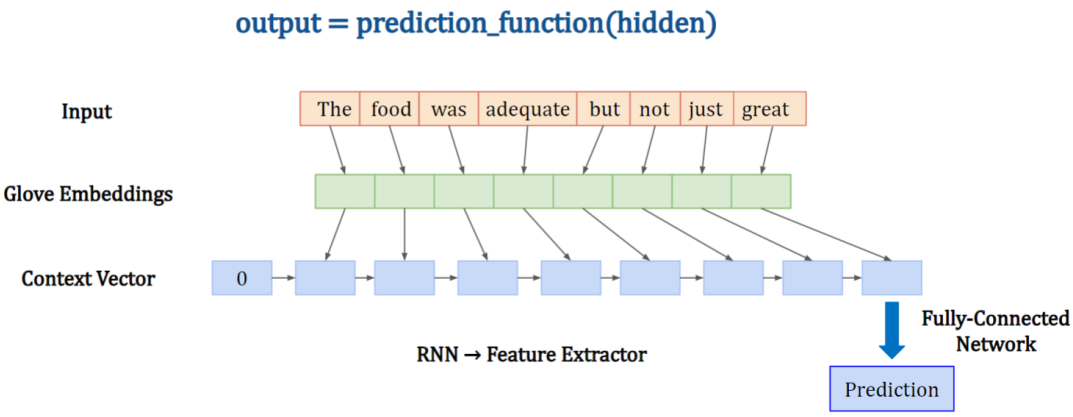
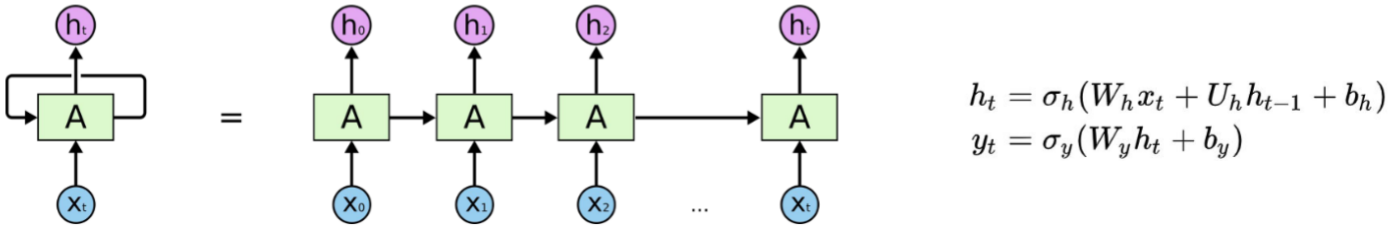


Figure 5.14: Last hidden state into classifier

- RNN layers have a recursive process that we have not seen in any other type of network so far.



$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

```
rnn_layer= nn.RNN(input_size=50,      # dimension of the input token
                  hidden_size=64,     # dimension of hidden state
                  batch_first=True)   # input format [batch, sequence, feature]
```

Figure 5.15: RNN layers visualized (unrolled)

Idea

RNN uses its past to determine its future. Many real world problems deal with inputs/outputs with varying sizes and require data from the past to accurately model future data.

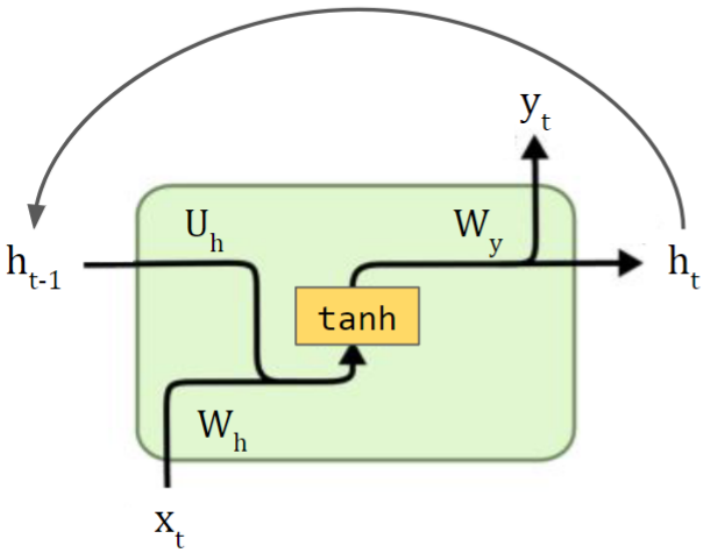
```python
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, __ = self.rnn(x, h0)
        # Pass the output of the last step to the classifier
        return self.fc(out[:,-1,:])

model = TweetRNN(50, 64, 2)
```

#layers

batch

Figure 5.16: PyTorch implementation of RNN (Training code is similar to previous architectures)

If we consider the concatenated input/hidden and output/hidden vectors as simply input/output, forward path in RNN is simply a fully-connected NN.



$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

Figure 5.17: Forward path of RNN as a fully-connected NN

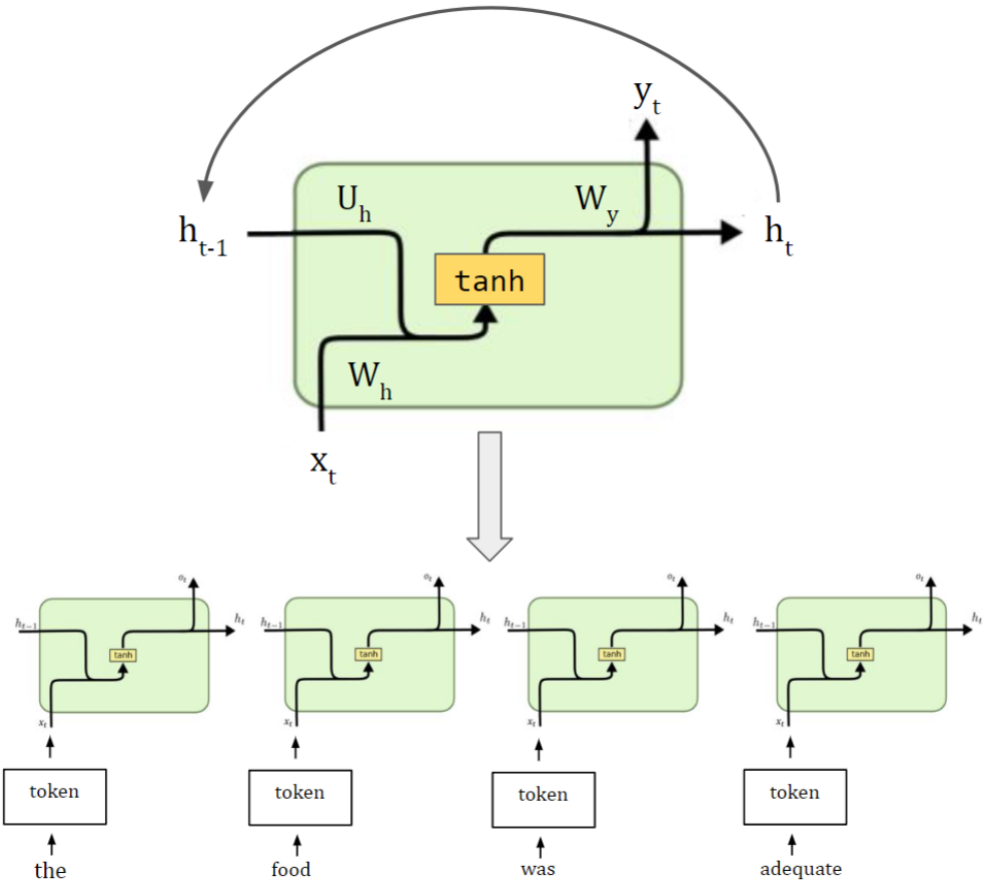


Figure 5.18: Unrolling an RNN

Sequence-Level Predictions

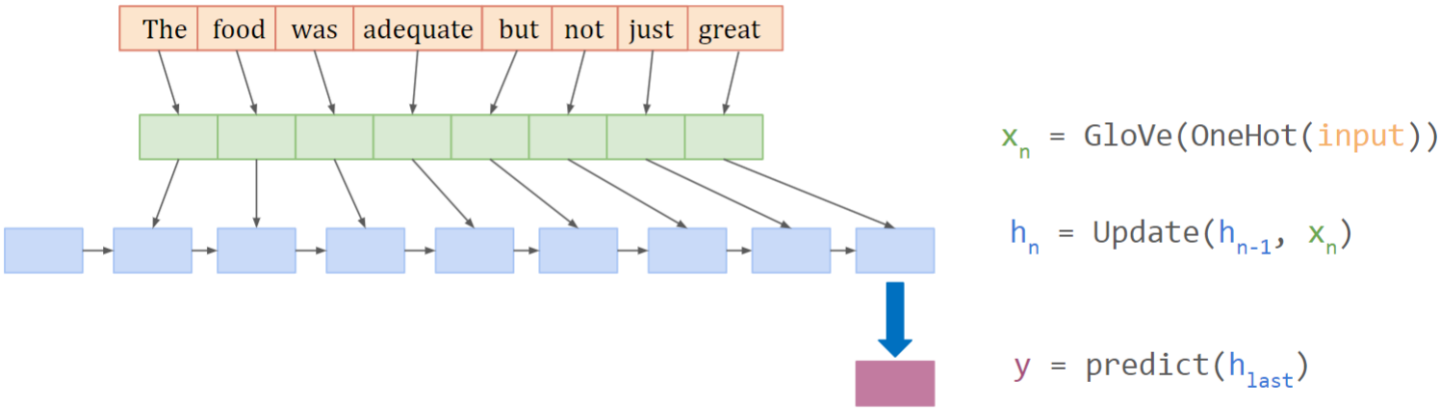


Figure 5.19: Sequence-Level Predictions

- These come into play when the goal is to make predictions or decisions based on the **entire sequence as a whole**.
- This is common in tasks like text summarization, machine summarization, or sequence-to-sequence tasks like language translation.
- In these cases, the model generates an output sequence that summarizes or transforms the input sequence, and the quality of the entire generated sequence is assessed rather than individual tokens.

Token-Level Predictions

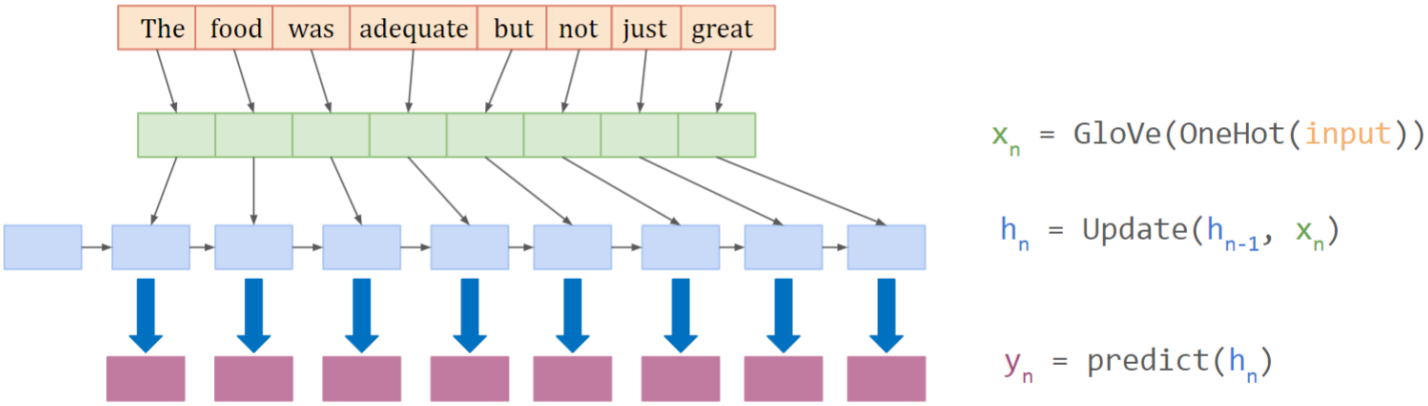


Figure 5.20: Token-Level Predictions

- These are employed when you need predictions or labels for **individual elements within a sequence**, such as words in a sentence or characters in a text.
- Token-level predictions are useful for tasks like part-of-speech tagging, named entity recognition, sentiment analysis, and machine translation, where you want to annotate or classify each element independently within the sequence

5.6 Limitations of Vanilla RNNs

What happens to RNNs unrolled onto a **long sequence**?

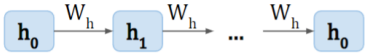
- RNNs can be very **deep** $\rightarrow$ Depth = Length of sequence

There are two related problems with vanilla RNNs:

- Not good at modelling **long-term dependencies**
- Hard to train due to **vanishing/exploding gradients**

Exploding/Vanishing Gradients

Suppose update function is a simple linear model. For simplicity, let's ignore inputs:



$$h_t = W_h h_{t-1}$$

We can write this for all time-steps as:

$$h_t = (W_h)^t h_0$$

Then we have:

- Exploding gradients $\quad h_t \rightarrow \infty \quad \text{if } |W_h| > 1$
- Vanishing gradient $\quad h_t \rightarrow 0 \quad \text{if } |W_h| < 1$

Figure 5.21: Exploding and Vanishing Gradients

Gradient clipping $\rightarrow$ Exploding Gradient

- If gradient is greater than a threshold, set the gradient to threshold

Skip-connection → Vanishing Gradient

- Ideally we want skip connections to all previous states → Too expensive
- We could preserve the hidden state/context over the long term

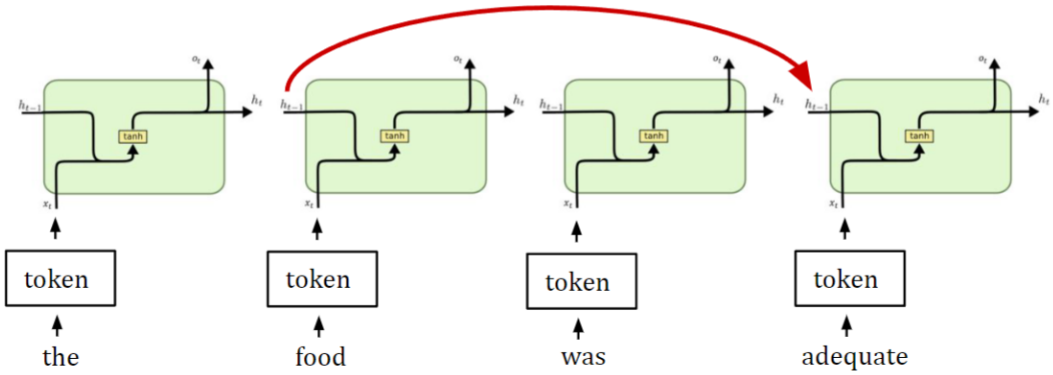


Figure 5.22: Skip-connections for RNN

Idea

Vanilla RNNs lack long-term memory.

5.7 LSTMs & GRUs

Gating Mechanism

Definition

Gating Mechanism: refers to a set of learnable components that regulate the flow of information through the network's hidden states. Gating mechanisms, such as those used in Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) cells, allow the network to selectively update, forget, or pass along information from previous time steps, enabling better modeling of long-term dependencies and mitigating issues like vanishing gradients.

- We can approximate skip-connections to all previous states by learning to **weight previous states differently** instead (soft skip-connections)
- We can use **gates** that learn to update the context **selectively**
- Gating mechanism controls how much informations flows through.
- Suppose X is a vector, then we can control how much of X to pass to next step by:
  1. Sigmoid or Tanh: $f(x) = X \cdot \sigma(X)$
  2. A neural network: $f(x) = X \cdot NN(X)$

Long Short-Term Memory (LSTM)

Definition

Long Short-Term Memory: a type of recurrent neural network (RNN) architecture designed to model sequential data by effectively capturing and preserving long-term dependencies. LSTMs use a specialized cell with gating mechanisms to control the flow of information, allowing them to store, update, and retrieve information over extended sequences, making them well-suited for tasks involving sequences, such as natural language processing and time series analysis.

- LSTMs consist of a **long-term memory** (cell state) and a **short-term memory** (context or hidden state)

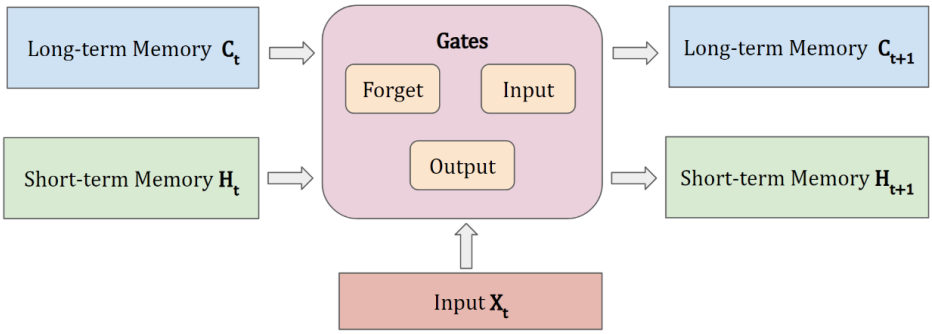
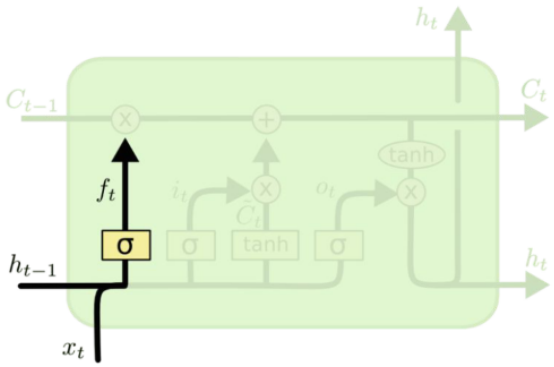


Figure 5.23: LSTM

- They use three gates to update the memories:

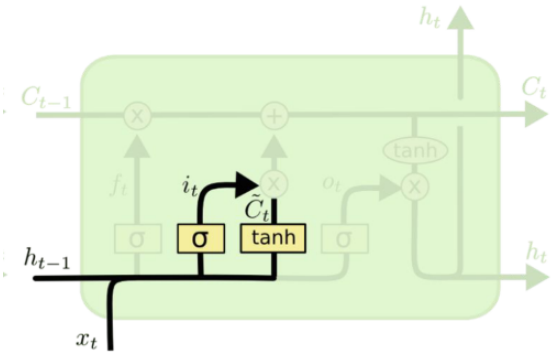
1. **Forget Gate** (long-term memory) → How much of the past memory should we forget?



$$f_t = \sigma\left(W_f . [h_{t-1}, x_t] + b_f\right)$$

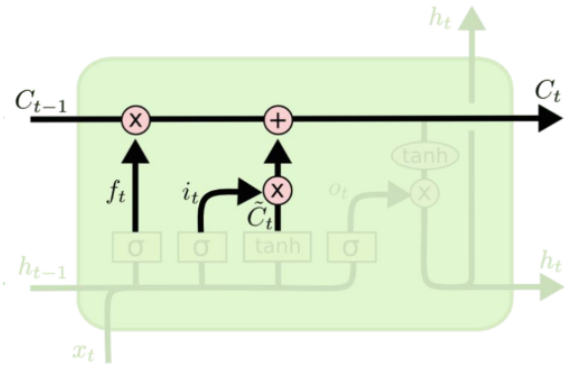
Figure 5.24: Forget Gate

2. **Input Gate** (long-term memory)→ How much the current input should contribute to the memory?
   (a) The updated long-term memory is the amount of past that is remembered (decided by forget gate) combined with the memory that was just created (decided by input gate)



$$i_t = \sigma\left(W_i . [h_{t-1}, x_t] + b_i\right)$$
$$\tilde{C}_t = \tanh\left(W_C . [h_{t-1}, x_t] + b_C\right)$$

Figure 5.25: Input Gate



$$C_t = f_t . C_{t-1} + i_t . \tilde{C}_t$$

Figure 5.26: Updated long-term memory

3. **Output Gate** (short-term memory) → How much of the updated long-term memory should construct the short-term memory?

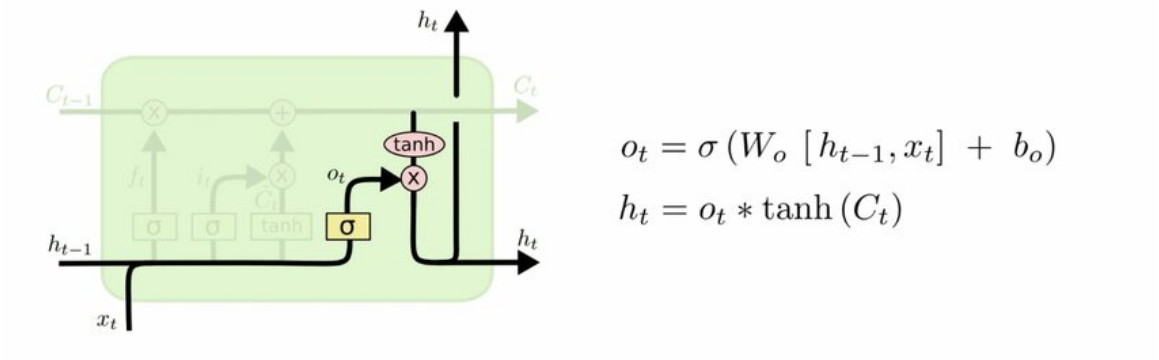


Figure 5.27: Output Gate

Gated Recurrent Unit (GRU)

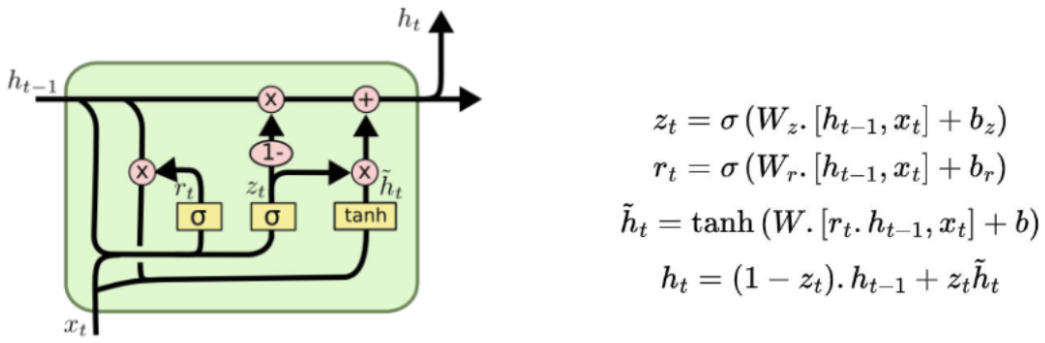


Figure 5.28: GRU

Definition

Gated Recurrent Unit: a type of recurrent neural network (RNN) architecture that is designed for modeling sequential data. It simplifies the architecture of Long Short-Term Memory (LSTM) networks by using fewer gates but still effectively captures and manages information over time. GRUs have gate mechanisms that control the flow of information, helping them mitigate the vanishing gradient problem and enabling the modeling of long-term dependencies in sequential data, making them suitable for various tasks such as natural language processing and time series analysis.

- GRUs are more efficient than LSTMs while having similar performance
- They combine **forget and input gates** into an **update gate**
- They also merge cell state and hidden state

LSTM/GRU vs RNN

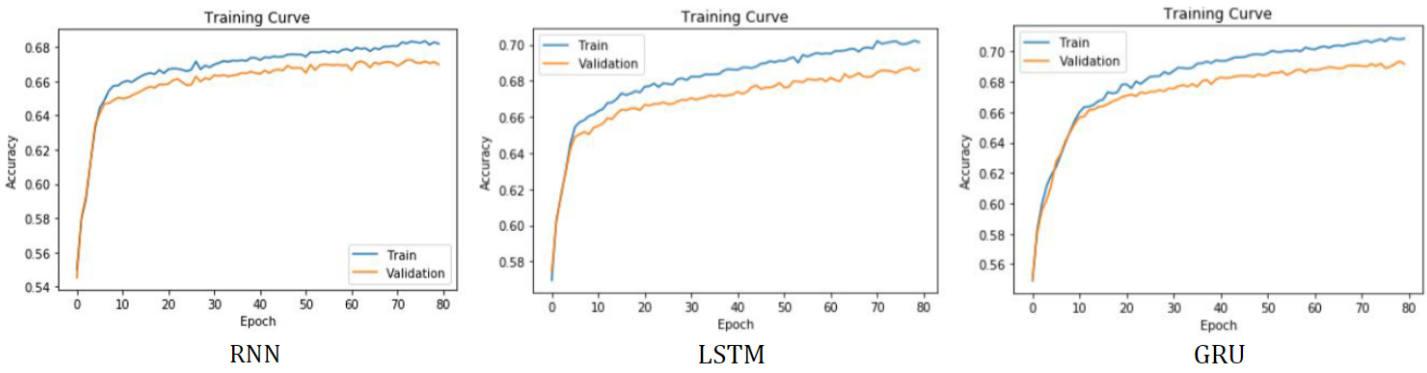


Figure 5.29: LSTM/GRU vs RNN

- LSTMs/GRUs can be trained on longer sequences and are much better at learning long-term relationships
- They are easier to train and achieve better performance than vanilla RNNs
- LSTM/GRU haven't finished converging here, with more training they do better

```python
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        c0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, __ = self.rnn(x, (h0, c0))
        # Pass the output of the last step to the classifier
        return self.fc(out[:,-1,:])
```

Figure 5.30: PyTorch implementation of vanilla RNN

```python
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.GRU(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, __ = self.rnn(x, h0)
        # Pass the output of the last step to the classifier
        return self.fc(out[:,-1,:])
```

Figure 5.31: PyTorch implementation of GRU RNN

```python
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        c0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, __ = self.rnn(x, (h0, c0))
        # Pass the output of the last step to the classifier
        return self.fc(out[:,-1,:])
```

Figure 5.32: PyTorch Implementation of LSTM RNN

5.8 Deep & Bidirectional RNNs

Bidirectional RNNs

Definition

Bidirectional RNN: a type of neural network architecture that processes input sequences in both forward and reverse directions simultaneously. This allows them to capture information from past and future time steps, making them particularly useful for tasks where context from both directions is essential, such as natural language understanding and speech recognition.

- A typical state in an RNN (RNN, GRU, or LSTM) relies on the past and the present.
- In tasks such as machine translation, where a prediction depends on the past, present, and future, we can exploit the future to improve performance.

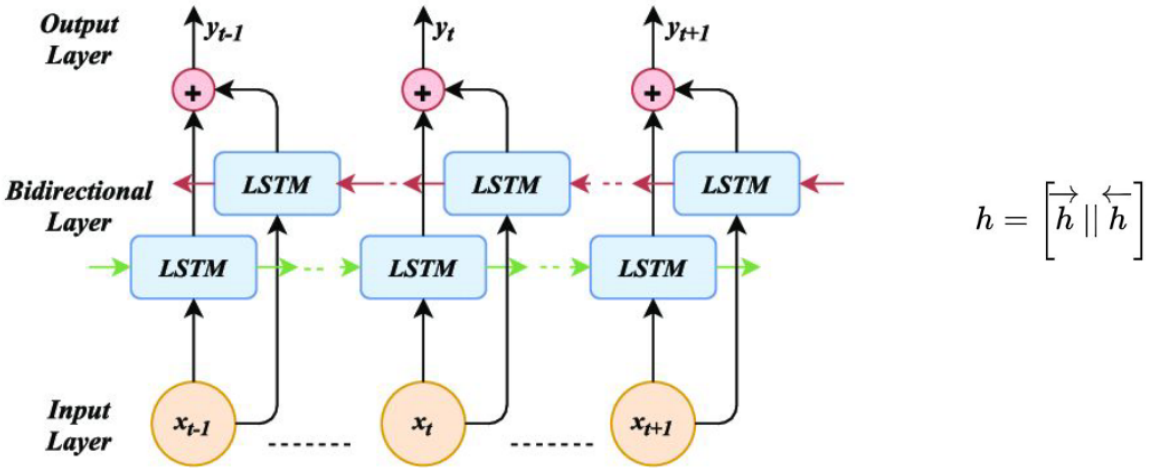


Figure 5.33: Bidirectional RNN

Deep RNNs

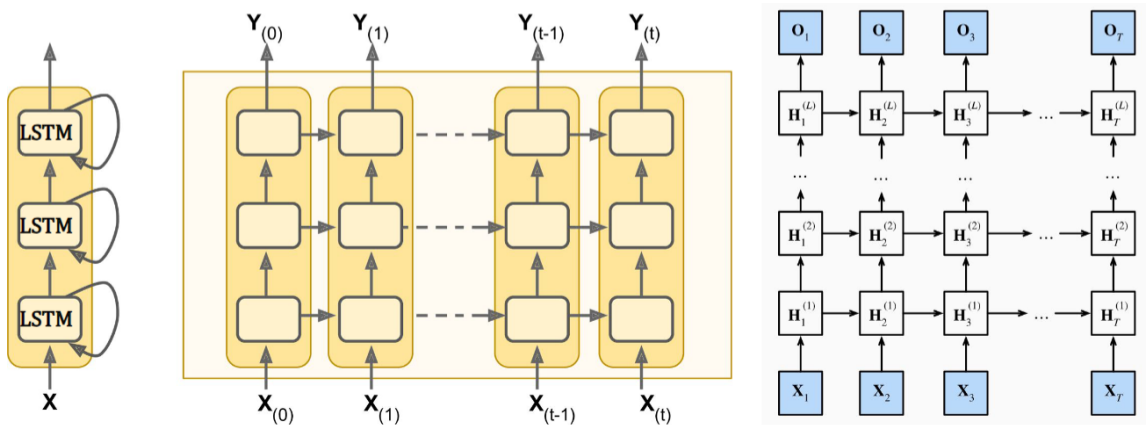


Figure 5.34: Deep RNN

Definition

Deep RNN: a type of recurrent neural network architecture that consists of multiple recurrent layers stacked on top of each other. These additional layers enable the network to capture more complex hierarchical features and representations from sequential data, making them suitable for tasks that require a deep understanding of temporal dependencies and context.

- We can also stack RNN layers to learn more abstract representations.
- Representations in first layers are better for syntactic tasks (low level) while representations in last layers perform better on semantic tasks (high level).

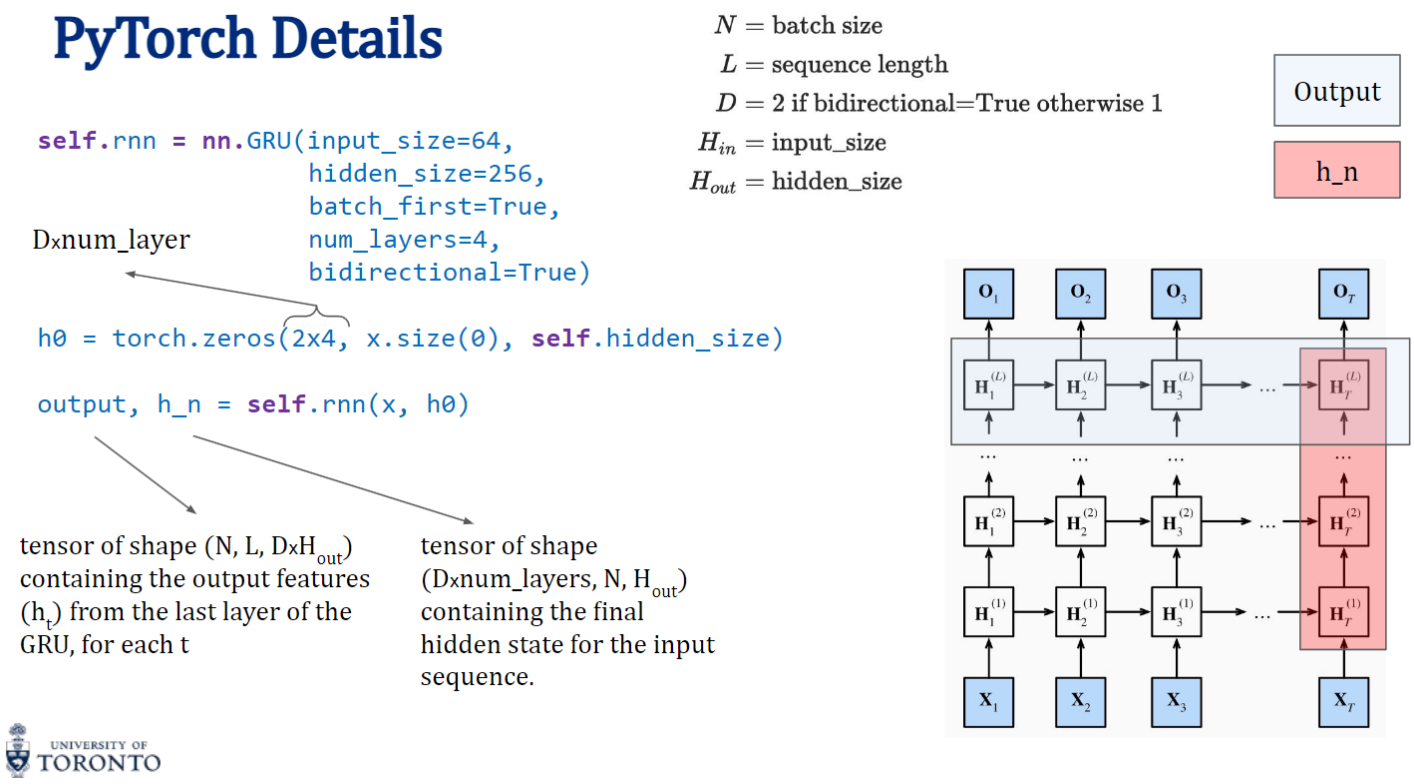


Figure 5.35: RNN configurations in PyTorch

5.9 Sequence-to-Sequence Models

Sequence as input and sequence as output.

RNN Model Types

- We have focused on many-to-one and many-to-many (same length)
- Many other types of tasks require different slightly different RNN setups → Translation, image captioning, etc.

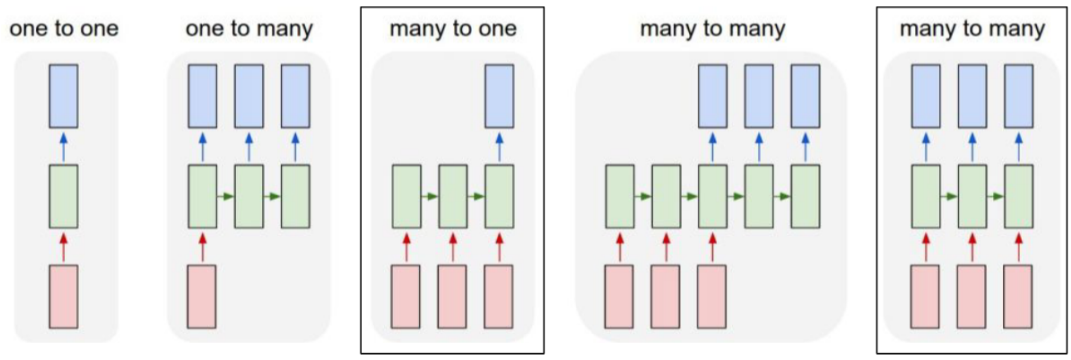


Figure 5.36: RNN Model Types

Hidden State Differences

> **Idea**
>
> Sequence to sequence models are actually generative. We will think of them like autoencoders.

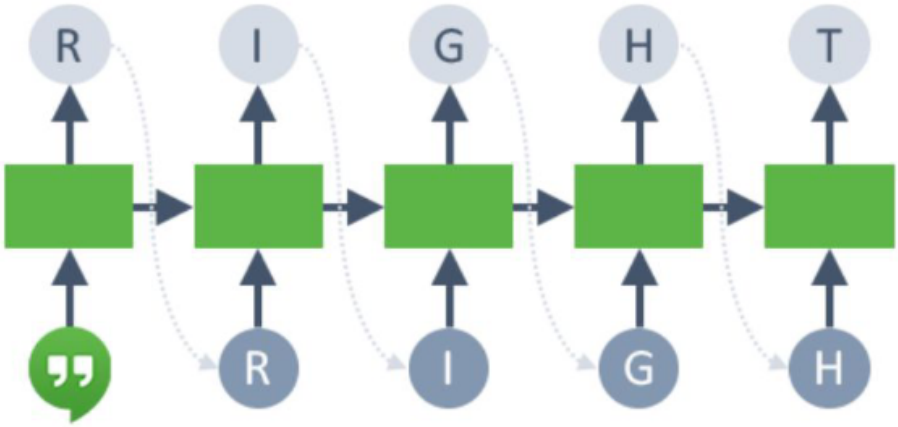


Figure 5.37: Generating ”RIGHT”

RNNs for prediction (**Encoder**)

- Process tokens one at a time
- Hidden state represents **all the tokens read thus far**

RNNs for generating sequences (**Decoder**)

- Generate tokens one at a time
- Hidden state is a representation of **all the tokens to be generated**

Sequence-to-Sequence RNNs

Learning to generate new sequences requires addressing some problems:

1. How do we generate variable-length sequences $\rightarrow$ how do we know when to **stop/finish a generated sequence!**
2. Training-time behavior must be changed (**teacher-forcing**)
    - Recall added ”randomness” in variational autoencoders
3. Inference-time behavior also changes (**sampling and temperature scaling**)

During Training

How do we know when to stop/finish a generated sequence? Let's use dedicated control symbols to define the Beginning of Sequence <**BOS**> and End of Sequence <**EOS**>.

<BOS> R I G H T <EOS>

Once the RNN generates <EOS>, we will know it is done generating!

How to define the **ground-truth and loss**? RNN is trained to generate one particular sequence in the training set:

- We feed the RNN with <BOS> and compare its prediction with R (**Cross-Entropy**)
- We then feed it with R and compare the prediction with I
- . . .

- Finally, we feed it with T and compare the prediction with <EOS>

Teacher Forcing

In each step, we compute the loss by comparing ground-truth and predicted tokens. In order to make training more **efficient**, we force the RNN to stay close to the ground-truth sequence. We do this by passing the **ground-truth label as the next input** instead of current prediction. This trick is known as **Teacher Forcing**.

During Inference

Unlike in a classification problem, always selecting the token with the **highest probability** won't work well:

- When using a generative model, we want **diversity**, not deterministic behavior.
- In practice, this greedy approach results in lots of **grammatical errors**.

To address this, we **sample from the predicted distributions**. We will address 3 sampling strategies:

- **Greedy search**

> **Definition**
>
> **Greedy Search:** selects the token with highest probability as the generated token.
>
> $$maxp(t_1, t_2, ...t_n) = maxp(t_1) \cdot p(t_2) \cdot ... \cdot p(t_n)$$

- **Beam search**

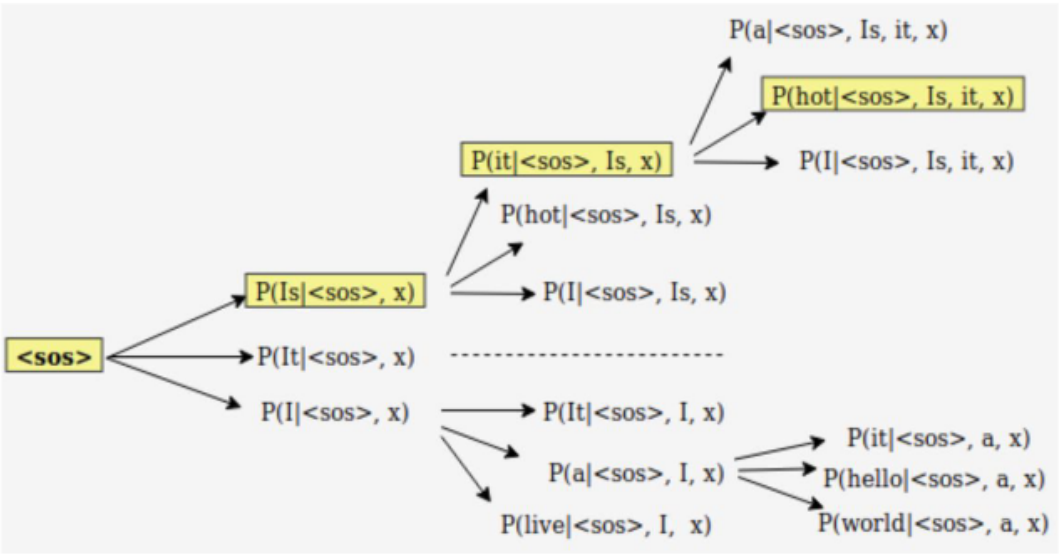


Figure 5.38: Beam Search

> **Definition**
>
> **Beam Search:** looks for a sequence of tokens with the highest probability within a window.
>
> $$maxp(t_1, t_2, ...t_n) = maxp(t_1) \cdot p(t_2|t_1) \cdot ... \cdot p(t_n|t_{n-1}, ..., t_2, t_1)$$

- **Softmax Temperature Scaling**

> **Definition**
>
> **Softmax Temperature Scaling:** helps with the problem of over-confidence in neural networks by by scaling the input logits to the softmax with a temperature.

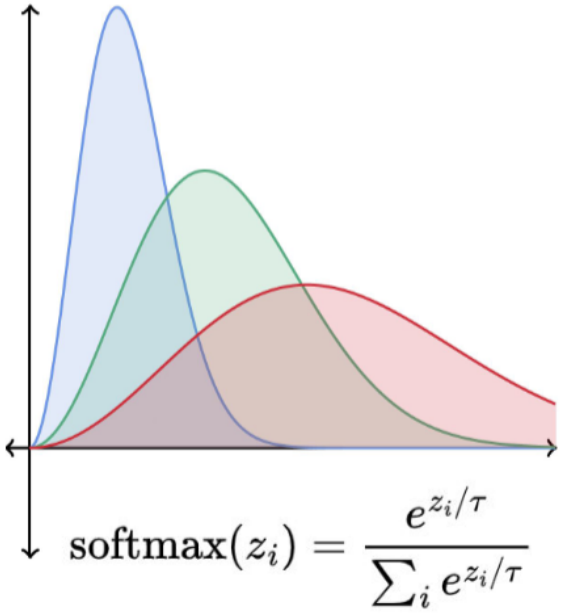


Figure 5.39: Softmax Temperature Scaling

- **Low Temperature** (larger logits, more confident): Higher quality samples, less variety
- **High Temperature** (smaller logits, less confident): Lower quality samples, more variety

```
print(sample_sequence(model, temperature=0.8))
print(sample_sequence(model, temperature=1.0))
print(sample_sequence(model, temperature=1.5))
print(sample_sequence(model, temperature=2.0))
print(sample_sequence(model, temperature=5.0))
```

```
God Bless the people of Venezuela!
God Bless the people of Venezuela!
os Bless the peopleeof Venezuela!
God Bdess Bpeooeop e ofzuenezeela!
auodlflBdptfh oe!on!!lhlpfeVeGhfpup!f
```

Figure 5.40: Various temperature scaling configurations

```
class TextGenerator(nn.Module):
    def __init__(self, vocab_size, hidden_size, n_layers=1):
        super(TextGenerator, self).__init__()
        # Identity matrix for generating 1-hot vectors
        self.ident = torch.eye(vocab_size)
        # Recurrent neural network
        self.rnn = nn.GRU(vocab_size, hidden_size, batch_first=True)
        # A FC layer outputting a distribution over the next token
        self.decoder = nn.Linear(hidden_size, vocab_size)

    def forward(self, inp, hidden=None):
        # Generate 1-hot vectors of input
        inp = self.ident[inp]
        # Get the next output and hidden state
        output, hidden = self.rnn(inp, hidden)
        # Predict distribution over next tokens
        output = self.decoder(output)
        return output, hidden
```

Figure 5.41: Text Generator in PyTorch

```
def train(model, data, batch_size=1, num_epochs=1, lr=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    data_iter = torchtext.legacy.data.BucketIterator(data,
                                                     batch_size=batch_size,
                                                     sort_key= lambda x: len(x.text),
                                                     sort_within_batch=True)
    for __ in range(num_epochs):
        Avg_loss = 0
        for (tweet, lengths), label in data_iter:
            target = tweet[:, 1:]
            inp = tweet[:, :-1]
            optimizer.zero_grad()
            output, __ = model(inp)
            loss = criterion(output.reshape(-1, vocab_size), target.reshape(-1))
            loss.backward()
            optimizer.step()
```

Figure 5.42: Training the text generator

```
def sample(sample, max_len=100, temperature=0.8):
    generated_sequence = ''
    inp = torch.Tensor([vocab_stoi['<BOS>']]).long()
    hidden = None
    for p in range(max_len):
        output, hidden = model(inp.unsqueeze(0), hidden)

        # Sample from the model as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = int(torch.multinomial(output_dist, 1)[0])

        # Add predicted character to string and use as next input
        predicted_char = vocab_itos[top_i]
        if predicted_char == '<EOS>':
            break
        generated_sequence += predicted_char
        inp = torch.Tensor([top_i]).long()
    return generated_sequence
```

Figure 5.43: Sampling the text generator

Idea

Language models and sequence-to-sequence models are self-supervised.

6 Generative Adversarial Networks (GANs)

Idea

One of the primary goals of GANs is to generate data that is indistinguishable from real data. This can be valuable in various applications, such as generating realistic images, videos, text, or other types of data. For example, GANs have been used to create lifelike images of nonexistent people, animals, or scenes.

6.1 Generative Models

Definition

Generative Model: a type of model that learns to model the probability distribution of the input data. It can generate new data points that resemble the training data, making it capable of creating synthetic data. Examples include Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs).

Definition

Discriminative Model: a model that learns to model the decision boundary between different classes or categories in the data. It's primarily used for classification tasks, aiming to distinguish between different classes based on input features. Examples include Logistic Regression, Support Vector Machines (SVMs), and Convolutional Neural Networks (CNNs) for image classification.

Generative Model vs. Discriminative Model

Suppose we have two tasks on a dataset of tweets:

1. Identify if a tweet is real or fake
   - This task is **supervised** and requires a **discriminative model.**
   - The model learns to approximate $p(y|x)$
2. Generate a new tweet
   - This task is **unsupervised** and requires a **generative model.**
   - The model learns to approximate $p(x)$

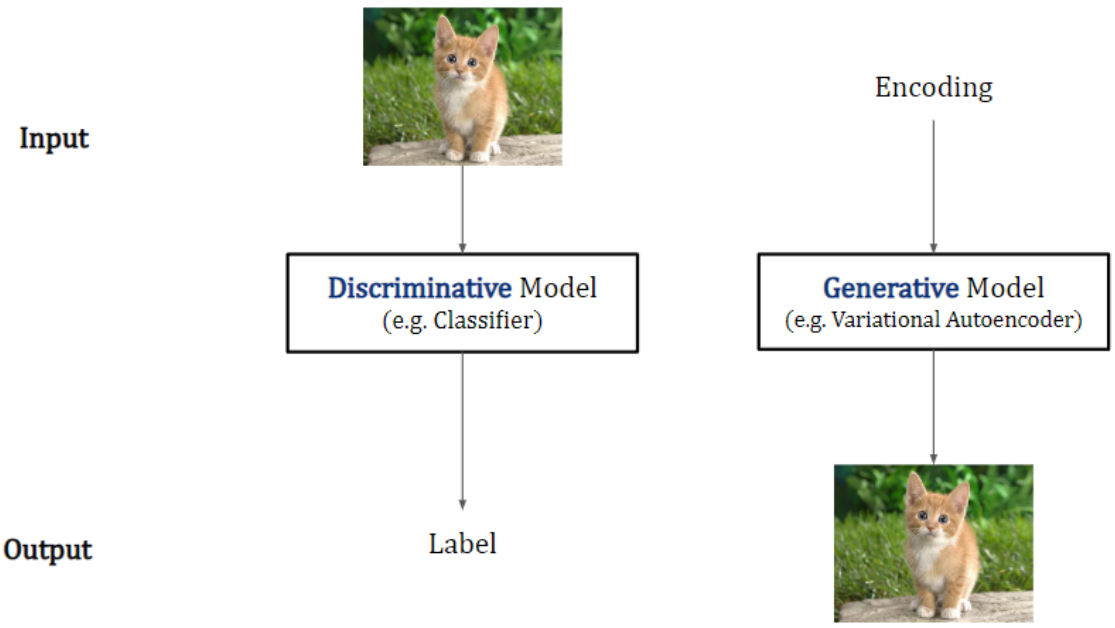


Figure 6.1: Generative Model vs. Discriminative Model

Generative learning is an **Unsupervised Learning task**:

- There is a loss function $\rightarrow$ an auxiliary task that we know the answer to
- There is no ground truth with respect to the actual task that we want to accomplish.
- We are learning the **structure & distribution of data**, rather than labels for data!

Generative Models

A generative model is used to generate new data, using some input encoding:

- **Unconditional Generative Models**
  - Only get random noise as input
  - No control over what category they generate
- **Conditional Generative Models**
  - One-hot encoding of the target category + random noise, or
  - An embedding generated by another model (e.g., from CNN)
  - User have a high-level control over what the model will generate

There are different families of deep generative models:

- Autoregressive Models
- Variational AutoEncoders (VAEs)
- Generative Adversarial Networks (GANs)
- Flow-Based Generative Models (not covered in course)
- Diffusion Models (not covered in course)

Problem with Autoencoders

Figure 6.2: Blurry result from a vanilla Autoencoder

Vanilla autoencoders generate blurry images with blurry backgrounds. To minimize the MSE loss, autoencoders predict the average pixel (**the loss function causes the blur**). Can we use a **better loss function**?

6.2 Generative Adversarial Networks

Idea

GANs: Idea → Train two models

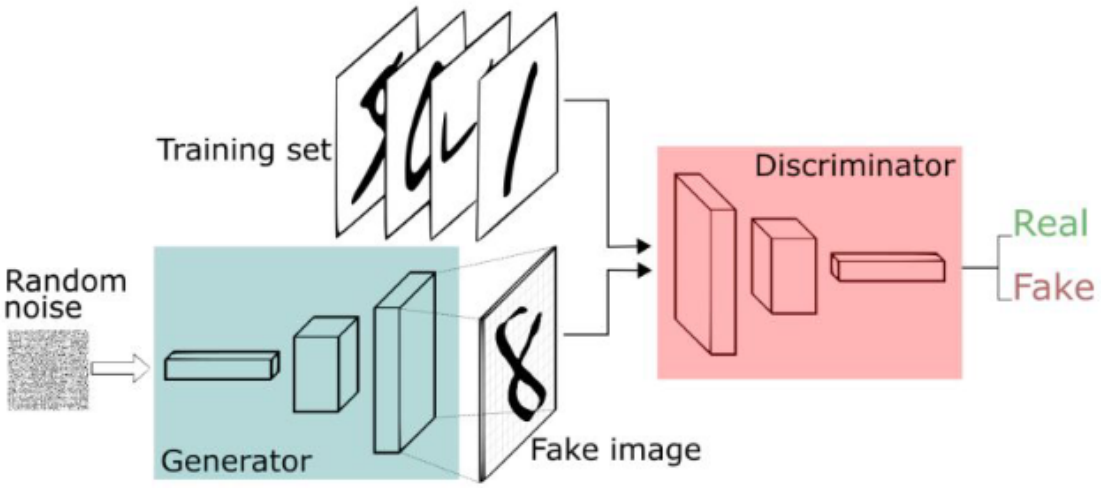


Figure 6.3: Architecture of a Generative Adversarial Network

- **Generator model:** try to fool the discriminator by generating real-looking images
- **Discriminator model:** try to distinguish between real and fake images

The loss function of the generator is defined by the discriminator!

- **Generator network**
  - Input → A noise vector
  - Output → A generated image
- **Discriminator network**
  - Input → An image
  - Output → A binary label (real vs fake)

Loss Function for MinMax Game

Idea

In a GAN, the discriminator and generator are playing a minmax game. The discriminator is trying to do the best job it can. The generator is set to make the discriminator as wrong as possible.

- The **discriminator** learns weights to **maximize the probability** that it labels a **real image as real** and a **generated image as fake**
- Since there are **two possiblities** as output for the discriminator (true & false), the optimal loss function is **Binary Cross-Entropy**
- On the other hand, the **generator** learns weights to **maximize the probability** that the **discriminator labels a generated (fake) image as real**
- The loss function of the generator is the **discriminator** itself
  - This is because if generator wins, discriminator loses and if generator loses, discriminator wins; there are only two possible scenarios

Training

- **Alternate** between training the discriminator and training the generator

Idea

GANs are notoriously difficult to train. One difficulty is that a training curve is no longer as helpful as it was for a supervised learning problem! The generator and discriminator losses tend to bounce up and down, since both the generator and discriminator are changing over time. Tuning hyperparameters is also much more difficult because we don't have the training curve to guide us.

for number of training iterations **do**
  for k steps **do**
    - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 6.4: Training a GAN

6.3 PyTorch Implementation

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
          nn.Linear(28*28, 300),
          nn.LeakyReLU(0.2),
          nn.Linear(300, 100),
          nn.LeakyReLU(0.2),
          nn.Linear(100, 1))

    def forward(self, x):
        x = x.view(x.size(0), -1)
        out = self.model(x)
        return out.view(x.size(0))
```

Figure 6.5: PyTorch Implementation of a Discriminator

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
          nn.Linear(100, 300),
          nn.LeakyReLU(0.2),
          nn.Linear(300, 28*28),
          nn.Sigmoid())

    def forward(self, x):
        out = self.model(x).view(x.size(0), 1, 28, 28)
        return out.view(x.size(0))
```

Figure 6.6: PyTorch Implementation of a Generator

```
def train_discrimintor(discriminator, generator, images):
    batch_size = images.size(0)
    noise = torch.randn(batch_size, 100)
    fake_images = generator(noise)
    inputs = torch.cat([images, fake_images])
    labels = torch.cat([torch.zeros(batch_size),    # Real
                        torch.ones(batch_size)])    # Fake
    outputs = discriminator(inputs)
    loss = criterion(outputs, labels)
    return outputs, loss
```

Figure 6.7: PyTorch Implementation of Training the Discriminator

```python
def train_generator(discriminator, generator, batch_size):
    batch_size = images.size(0)
    noise = torch.randn(batch_size, 100)
    fake_images = generator(noise)
    outputs = discriminator(fake_images)
    # Only looks at fake outputs
    # gets rewarded if we fool the discriminator!
    labels = torch.zeros(batch_size)
    loss = criterion(outputs, labels)
    return fake_images, loss
```

Figure 6.8: PyTorch Implementation of Training the Generator

6.4 Problems of Training GANs

Vanishing Gradients

- If the discriminator is too good, then the generator will not learn (or the other way around)
- Remember that we are using the discriminator as a loss function for the generator
- If the discriminator is too good, small changes in the generator weights **won't change the discriminator output**
  - No proper feedback for the generator to improve
- If small changes in generator weights make no difference, then we can't incrementally improve the generator **(no gradients!)**

Mode Collapse

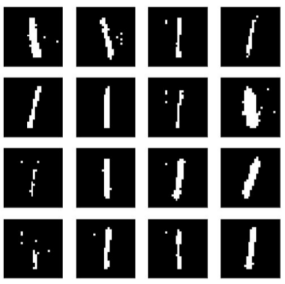


Figure 6.9: Mode Collapse

- We want the generator to generate variety of outputs (e.g., all digits within MNIST).
- If generator starts producing the **same output (or a small set of outputs),** the best strategy for the discriminator is to reject that output.
- However, if the discriminator is trapped in **local optimum**, it cannot adapt to generator, and the generator can fool it by only generating one type of data (e.g. only digit 1)
  - Basically, the detector become very good at detecting some specific categories but not others.
  - The generator exploits this weakness

> **Idea**
>
> To prevent mode collapse, newer variations of GANs provides the discriminator with a small set of either real or fake data, rather than one at a time. A discriminator would therefore be able to use the variety of the generated data as a feature to determine whether the entire small set of data is real or fake

Failing to Converge

- Due to the MinMax optimization process, training Vanilla GANs is **very difficult**.
- It is difficult to numerically see whether there is progress → Plotting the "training curve" doesn't help much!
- Takes a long time to train (a long time before we see progress)
- To train GANs faster, we'll use:
  - LeakyReLU Activations instead of ReLU
  - Batch Normalization
  - Regularizing discriminator weights & adding noise to discriminator inputs

6.5 Applications of GANs

- **Face Generation (This Person Does Not Exist)**

Figure 6.10: Examples of GAN-generated faces

- **Grayscale to Color**

Figure 6.11: Grayscale to color examples

Idea

Grayscale to color involves a conditional generator.

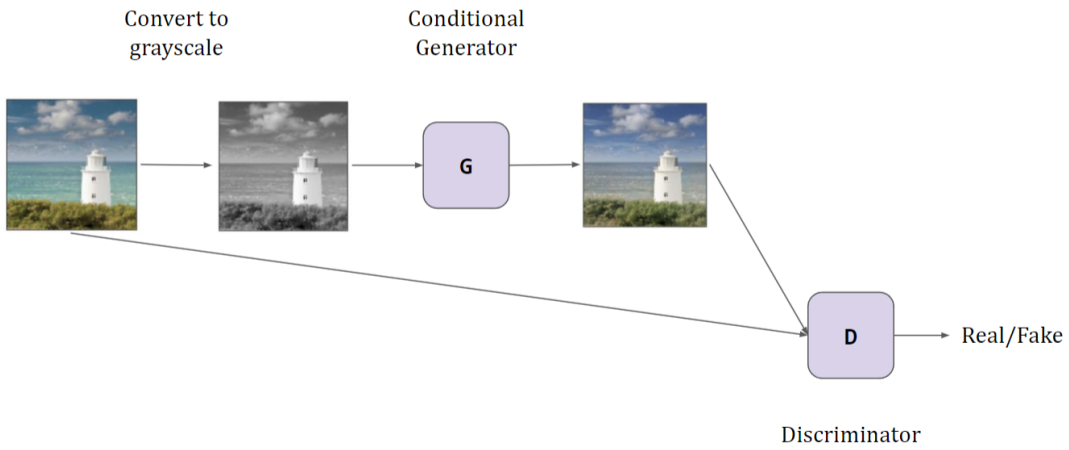


Figure 6.12: Conditional Generator

Example

How could we have a GAN trained on MNIST output only specific digits?
Pass the specific class as input to the generator and discriminator so that both will know what type of class is being generated, and they can compete on that specific class

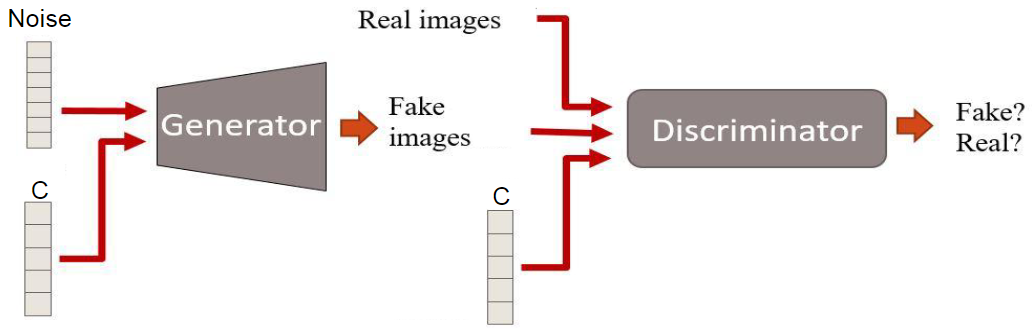


Figure 6.13: Passing in a specific class

- **Style Transfer**

Cycle GAN: Cycle loss is reconstruction loss between input to cyclegan and output of cyclegan to ensure consistency (consists of two gans)

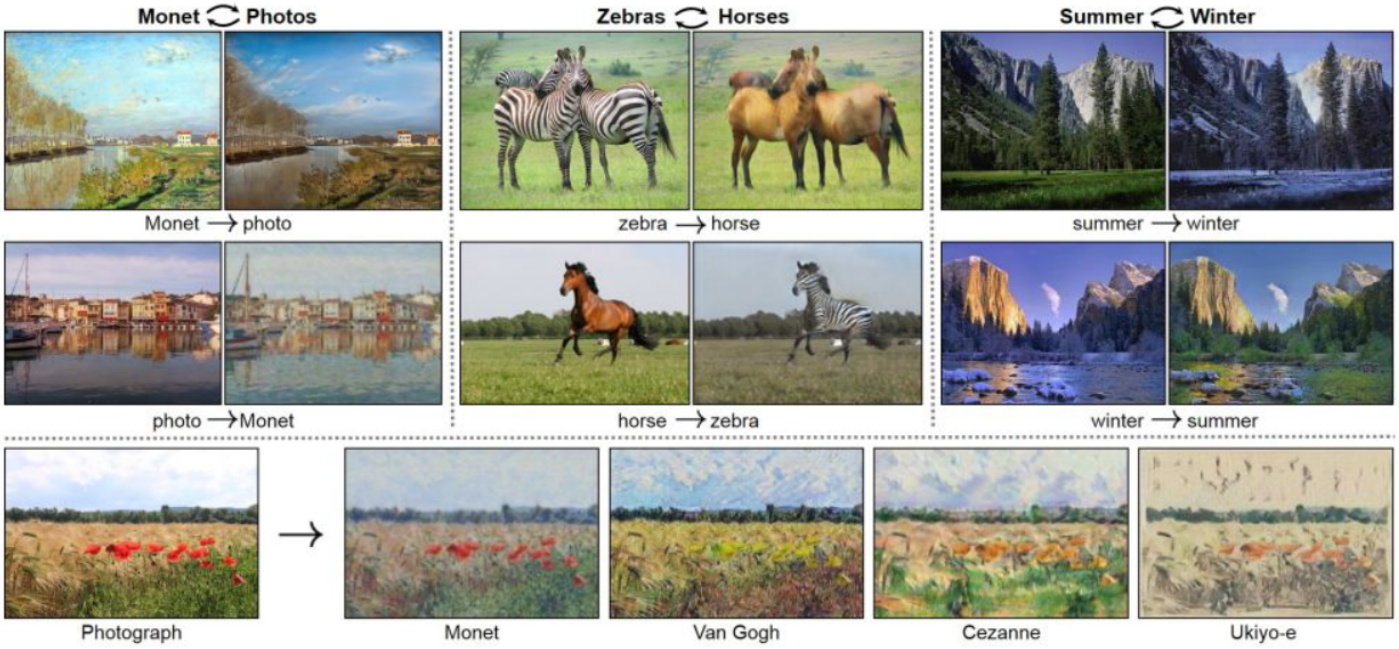


Figure 6.14: Style Transfer

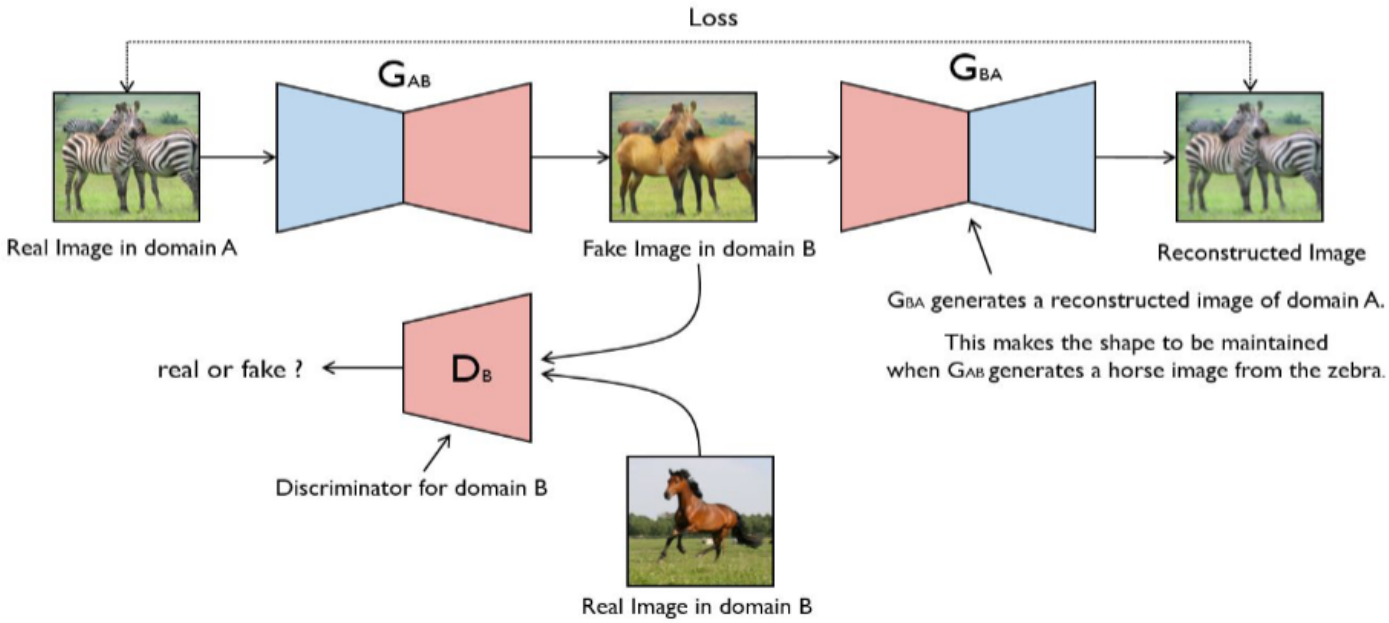


Figure 6.15: CycleGan

6.6 Adversarial Attacks

Definition

Adversarial Attack: a deliberate manipulation of input data, typically to machine learning models, with the goal of causing the model to make incorrect predictions or classifications. These manipulations are often small, imperceptible changes designed to exploit vulnerabilities in the model's decision boundaries and undermine its accuracy and reliability. Adversarial attacks can pose a significant security and reliability challenge for machine learning systems.

- **Goal**: Choose a small perturbation ϵ on an image x so that a neural network f misclasses x + ϵ
- **Approach**: Use the same optimization process to choose ϵ to minimize the probability that

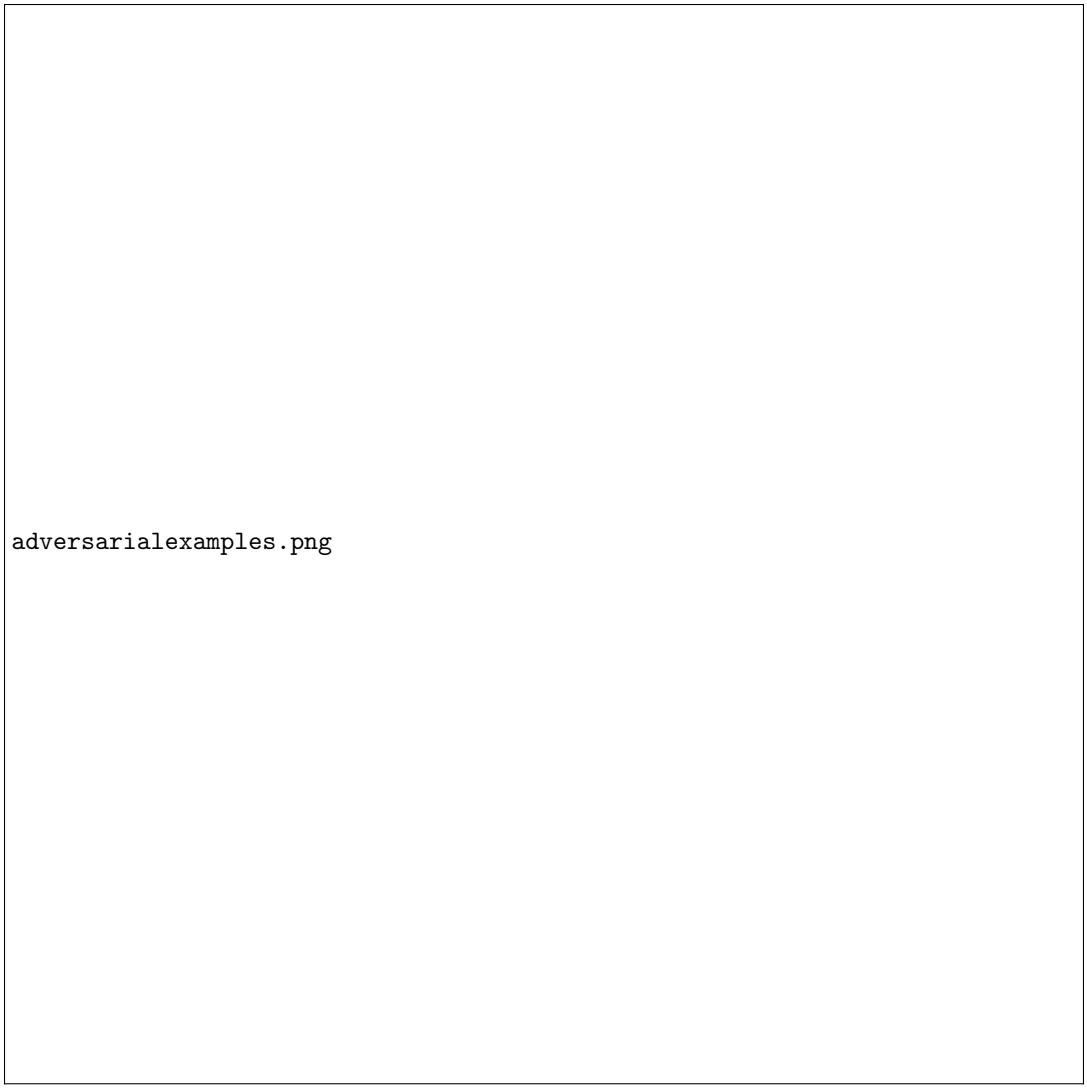


Figure 6.16: Adversarial Examples

$$f(x + \epsilon) = \text{correct class}$$

- We are treating ϵ as the **parameters**

Targeted vs. Non-Targeted Attack

- **Non-targeted attack**
  - Minimize the probability that:
    $$f(x + \epsilon) = \text{correct class}$$
  - The attacker's goal is to cause any form of misclassification or disrupt the model's predictions without specifying a particular target class
  - The objective is to **degrade the model's performance** without a specific end result in mind
  - Example: manipulate image of cat so that it's classified as anything other than a cat
  - Often easier to carry out because they do not require the attacker to have knowledge of the target class. They may use simpler techniques and do not necessarily need to find the smallest perturbation
- **Targeted attack**
  - Maximize the probability that:
    $$f(x + \epsilon) = \text{target class}$$

  - Attacker aims to cause a particular misclassification or a specific response from the machine learning model
  - The attacker has a **clear target class** or outcome in mind
  - Example: manipulate image of cat so that it's classified as a dog
  - Requires a deep understanding of the model's decision boundaries and often involve optimization techniques to find the smallest possible perturbations that achieve the desired target outcome

> **Idea**
>
> Targeted and Non-Targeted Attacks differ in the attacker's intent.

White-Box vs. Black-Box Attacks

- **White-box attacks**
  - Assumes that the model is known
  - We need to know the architectures and weights of f to optimize ϵ
- **Black-box attacks**
  - Don't know the architectures and weights of f to optimize ϵ
  - Substitute model mimicking target model with known, differentiable function
  - Adversarial attacks often transfer across models! (because they aren't tailored to a model with specific parameters)

> **Idea**
>
> White and Black-Box Attacks differ in the attacker's knowledge of the model.

Defence Against Adversarial Attack
It is a very active area of research, and we still don't know how to handle them. Failed Defenses:

- Adding noise at test time
- Averaging many models
- Weight decay
- Adding noise at training time
- Adding adversarial noise at training time
- Dropout

7 Transformers

7.1 Motivation

- **RNNs** are used to model sequences
- **Vanilla RNNs** cannot capture long dependencies due to exploding/vanishing gradients
- **LSTMs and GRUs** are commonly preferred over vanilla RNNs

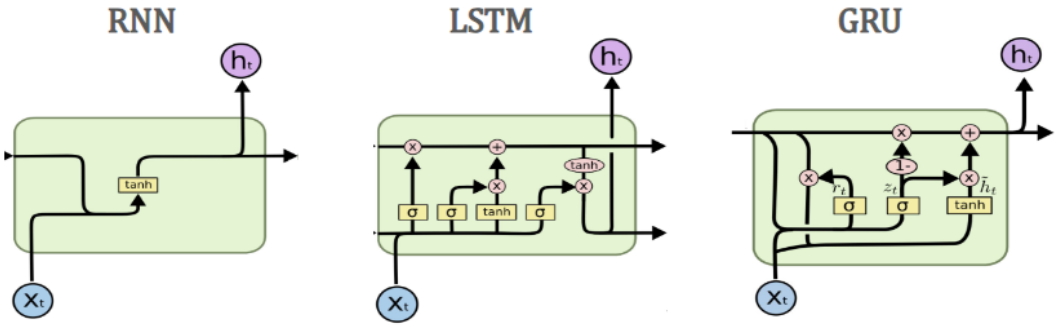


Figure 7.1: Types of RNNs for modeling sequences

- Regardless of their type, RNNs are **sequential in nature**
  - This means that they are **inefficient** as we can't take full advantage of vectorization or GPUs (which can **parallelize** computations)

7.2 Attention Mechanism

When humans read or look:

- They focus (**attend**) on some regions within the input → **high resolution**
- They pay less attention to surrounding and perceive it less → **low resolution**

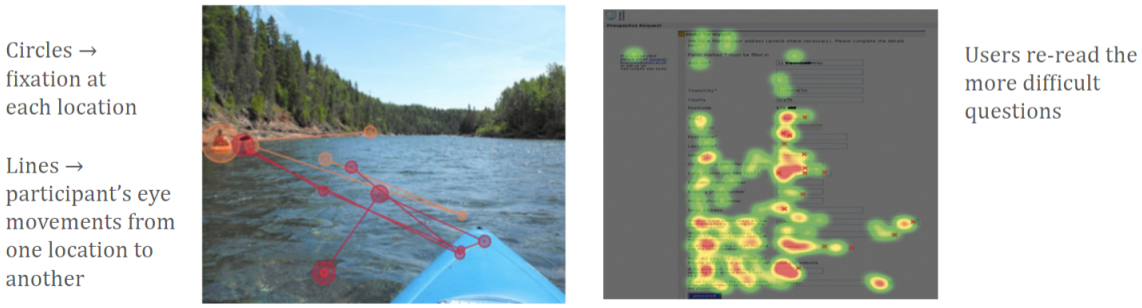


Figure 7.2: Visual Attention

How does Attention work?

- The network learns an attention score
- Attention score indicates the importance of different parts of the data
- We then use the attention score to aggregate the data

Simple Attention Mechanism

- Let's design an attention-based pooling for **classifying tweets**

Figure 7.3: Importance of words in a sentence

- We want to use it to pool the word embeddings of a tweet into a **tweet embedding**
- A possible way is to **sum or average** the word embeddings
  - The problem with this is assumes all the words have the same importance (or warrant the same attention)
- A better way is to use a fully-connected network that takes in word embeddings and generates a single score for each embedding
- We can then normalize these scores across all words within the tweet using a softmax
- We then multiply each embedding with its normalized score and sum them up:

$$c_i = \sum_j \alpha_{ij} h_j$$

- This network is trained end-to-end with the classifier

Attention

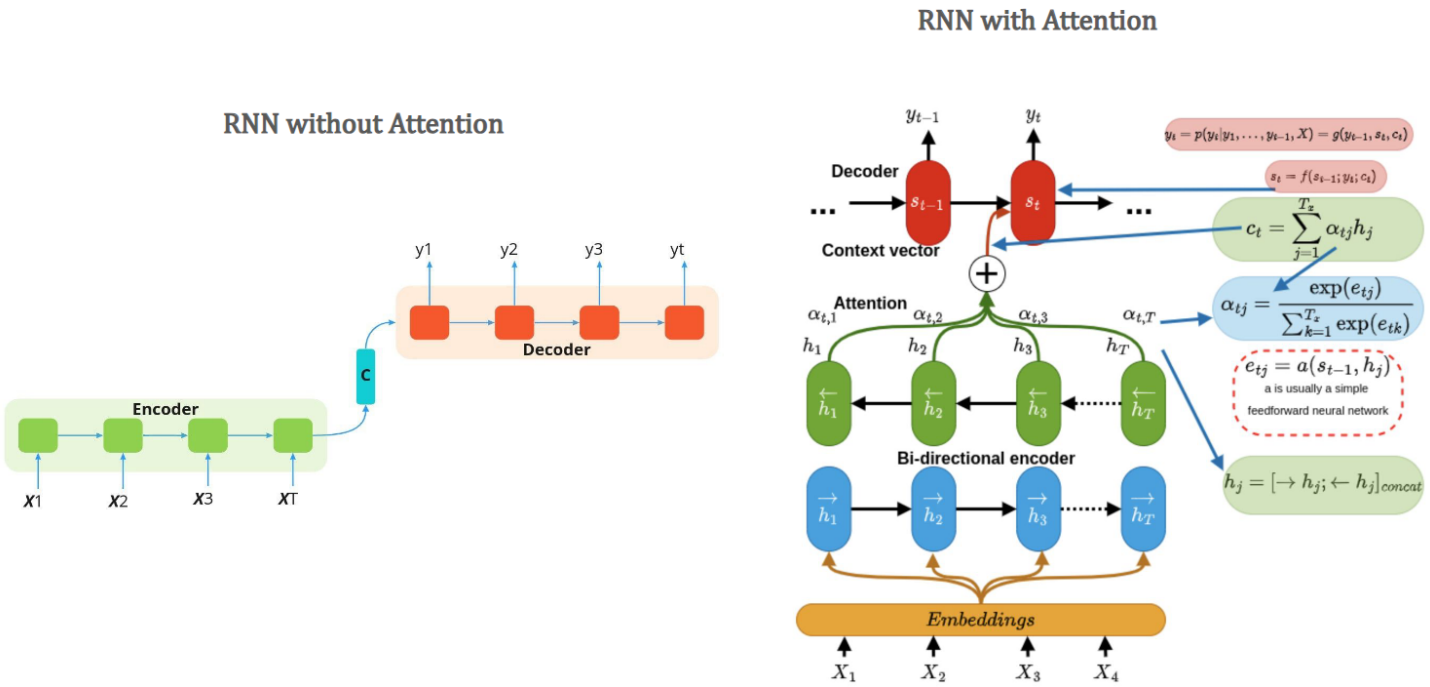


Figure 7.4: RNN with attention vs RNN without attention

Definition

Cross-Attention: considers interactions between elements from two different sequences. Example: one sequence in language 1 and another in language 2 (for a translator).

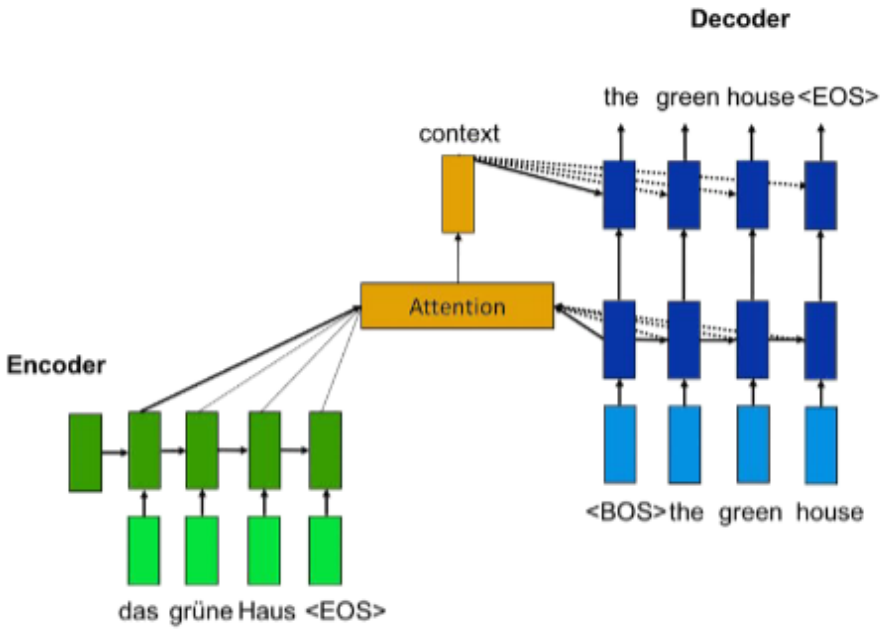


Figure 7.5: Cross-Attention

Definition

Self-Attention (Inta-Attention): compute attention of input with respect to itself to itself: for a given token of the input, comput attention weight for all other tokens in the sequence

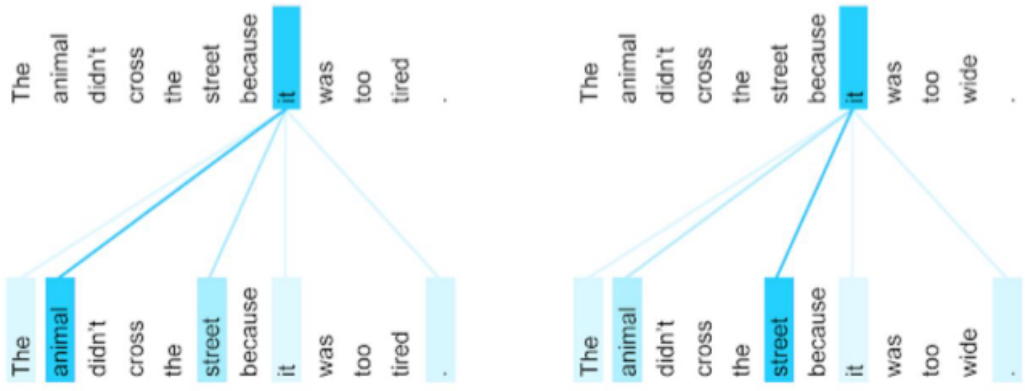


Figure 7.6: Self-Attention (Inta-Attention)

Computing Attention Score

- Suppose we have two embeddings: $a, b \in R^d$
- We can use different methods to compute the attention score between them:
  - Dot product score: $score(a, b) = a^T \cdot b$

- Cosine similarity score: $score(a, b) = \frac{a^T \cdot b}{\|a\| \cdot \|b\|}$
- Bilinear score: $score(a, b) = a^T W b$
- MLP score: $score(a, b) = \text{Sigmoid}(W[a; b])$
- ...

7.3 Transformers

Definition

Transformer: a type of deep learning architecture introduced in the field of natural language processing. They are designed to handle sequential data efficiently by relying on a self-attention mechanism, allowing them to capture long-range dependencies in input sequences. Transformers have become a fundamental building block for various tasks, such as machine translation, text generation, and more, due to their parallelizable and scalable nature.

Definition

Values: represent the information that is being retrieved or queried based on the relationships between the keys and queries. In other words, values hold the content or features associated with each element in the input sequence. During self-attention, the values are weighted and combined according to the attention scores computed between queries and keys.

Definition

Queries: used to inquire about the relationships between different elements in the input sequence. They represent a specific element's information and are compared to keys to determine how relevant each key is to the current query. Queries are typically generated from the input data and used to compute attention scores.

Definition

Keys: used to provide context and establish relationships between different elements in the input sequence. They are also generated from the input data and are compared to queries to calculate attention scores. Keys help determine how much attention should be given to each value when processing the current query.

Attention in Transformers

Transformers are a class of deep models that are based on **self-attention** where attention is modeled as a neural dictionary:

- It retrieves a **value** v_i for a **query** q based on a **key** k_i
- Values, queries, and keys are d-dimensional embeddings
- However, rather than retrieving a single value for a query, it uses a **soft retrieval**
- It retrieves all the values but then computes their importance with respect to the query based on the similarity between the query and their keys

$$\text{attention}(q, \mathbf{k}, \mathbf{v}) = \sum_i \text{similarity}(q, k_i) \times v_i$$

- Suppose X is an input sequence consisting of n tokens where each token $t \in R^i$.
- To compute the queries, keys, and values from X, we use three linear layers:

$$\begin{aligned} Q &= XW_Q, \quad X \in R^{n \times i}, \quad W_Q \in R^{i \times k}, \quad Q \in R^{n \times k} \\ K &= XW_K, \quad X \in R^{n \times i}, \quad W_K \in R^{i \times k}, \quad K \in R^{n \times k} \\ V &= XW_V, \quad X \in R^{n \times i}, \quad W_V \in R^{i \times v}, \quad V \in R^{n \times v} \end{aligned}$$

Note that queries, keys, and values are computed on the same input sequence.

- Self-attention in Transformers is defined as follows:

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

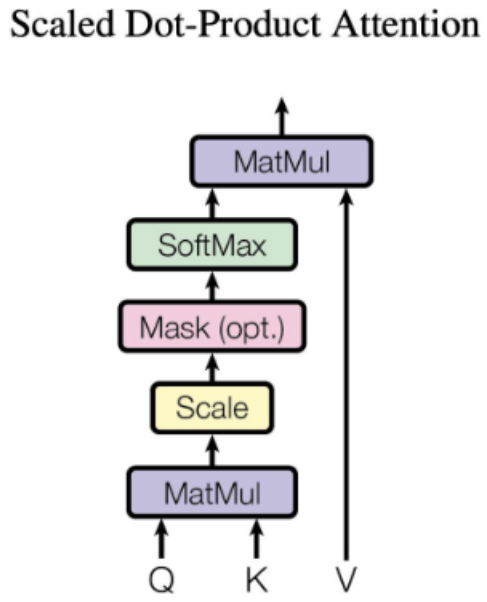


Figure 7.7: Scaled Dot-Product Attention

Idea

Word2Vec and GloVe models have fixed embeddings (static) for each word, which is not good because natural language is contextual. Attention in transformers allows context of the sentence to define the meaning and the embedding of a word.

Multi-Head Attention

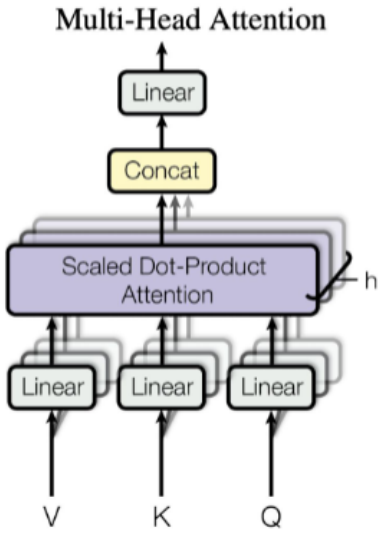


Figure 7.8: Multi-Head Attention

To improve the performance:

1. Divide the representation space to h sub-spaces
2. Run parallel linear layers and attentions
3. Concatenate them back to form the original space

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

Transformer Encoders

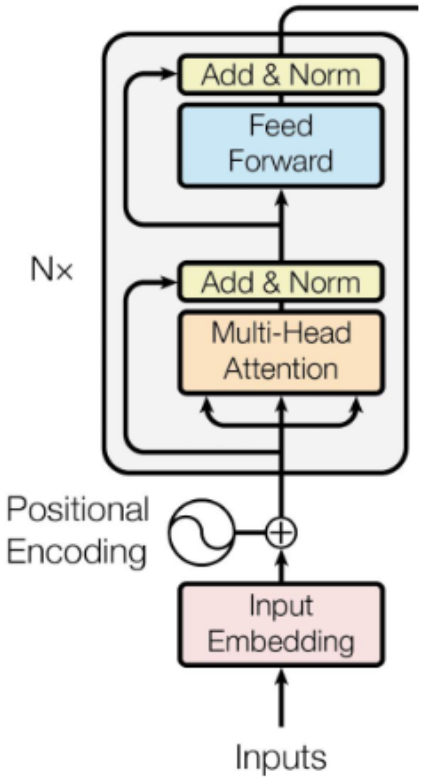


Figure 7.9: Transformer Encoder

Each encoder layer consists of:

1. A multi-head self-attention sub-layer
2. A fully-connected sub-layer
3. A residual connection around each of the two sub-layers followed by layer normalization

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$
$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$
$$LayerNorm(x + Sublayer(x))$$

Positional Encoders

- The model does **not have recurrent or convolutional layers** so it doesn't into account the order of sequence
- We use **positional encoding** to make use of the order which allows the model to easily learn to attend by relative positions

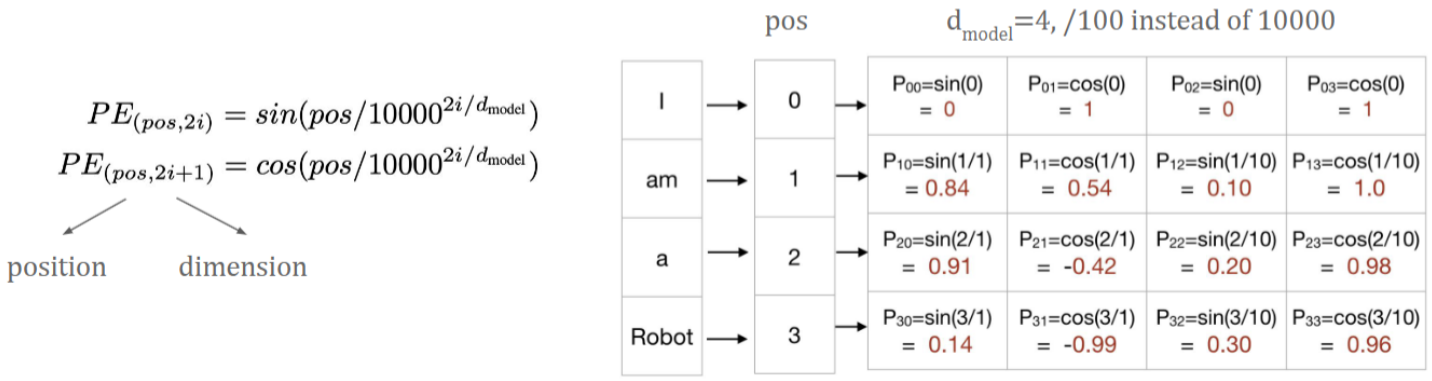


Figure 7.10: Positional Encoder

RNNs v.s. Transformers

RNN	Transformer
• Struggling with long range dependencies • Gradient vanishing and explosion • Large number of training steps • Recurrence prevents parallel computation	• Facilitate long range dependencies • Less likely to have gradient vanishing and explosion problem • Fewer training steps • No recurrence, facilitates parallel computation

7.4 PyTorch Implementation

```
class TransformerEncoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(TransformerEncoder, self).__init__()
        self.linear_q = nn.Linear(input_size, hidden_size)
        self.linear_k = nn.Linear(input_size, hidden_size)
        self.linear_v = nn.Linear(input_size, hidden_size)
        self.linear_x = nn.Linear(input_size, hidden_size)
        self.attention = nn.MultiheadAttention(hidden_size, num_heads=4, batch_first=True)
        self.fc = nn.Sequential(
                        nn.Linear(hidden_size, hidden_size),
                        nn.ReLU(),
                        nn.Linear(hidden_size, hidden_size))
        self.norm = nn.LayerNorm(hidden_size)

    def forward(self, x):
        q, k, v = self.linear_q(x), self.linear_k(x), self.linear_v(x)
        x = self.norm(self.linear_x(x) + self.attention(q, k, v))
        x = self.norm(x + self.fc(x))
        return x
```

Figure 7.11: PyTorch Implementation of a Transformer Encoder

```
class TweetTransformer(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetTransformer, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.encoder = TransformerEncoder(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x, pos):
        # Add GloVe vectors to positional encoding
        x = self.emb(x) + pos
        x = self.encoder(x)
        # Add embeddings from transformer encoding to get tweet embedding
        x = torch.sum(x, -1)
        # Classify
        return self.fc(x)
```

Figure 7.12: PyTorch Implementation of a Transformer Classifier

7.5 Language Modeling

We can use a self-supervised objective such as predicting the next work to to learn embeddings over tokens:

Word2Vec/GloVe:

- Learn static embeddings
- One embeding for all senses

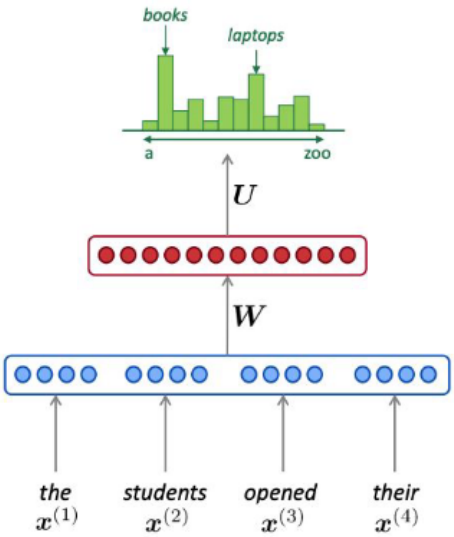


Figure 7.13: Static Embeddings

RNNs/Transformers:

- Learn contexual embeddings
- Embeddings of a same word changes according to the sentence it appears in

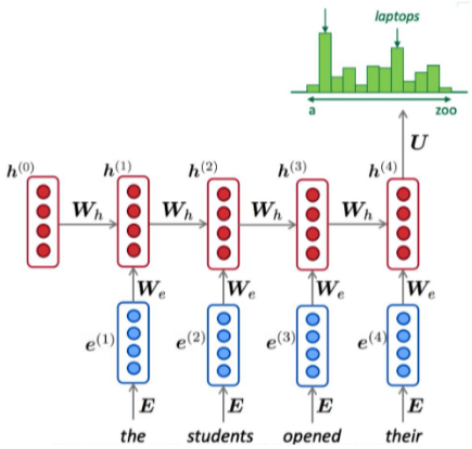


Figure 7.14: Contextual Embeddings

Large Language Models with Transformers

Tranformer models differ in terms of their prediction tasks and training objectives

- XLNet
- ELMo (Word embedding model)
- GPT1, GPT2, GPT3 (Generative models)
- BERT-small, BERT-medium, BERT-large, ...
- RoBERTa
- Big Bird (sparse attention)
- And more...

BERT

Bidirectional Encoder Representations from Transformers (BERT)

- Achieved SOTA results on various NLP tasks
- A Transformer model trained with two self-supervised tasks
- Shows great transfer learning capabilities
- Being used in Google search engine to represent user queries and documents

Input Embeddings

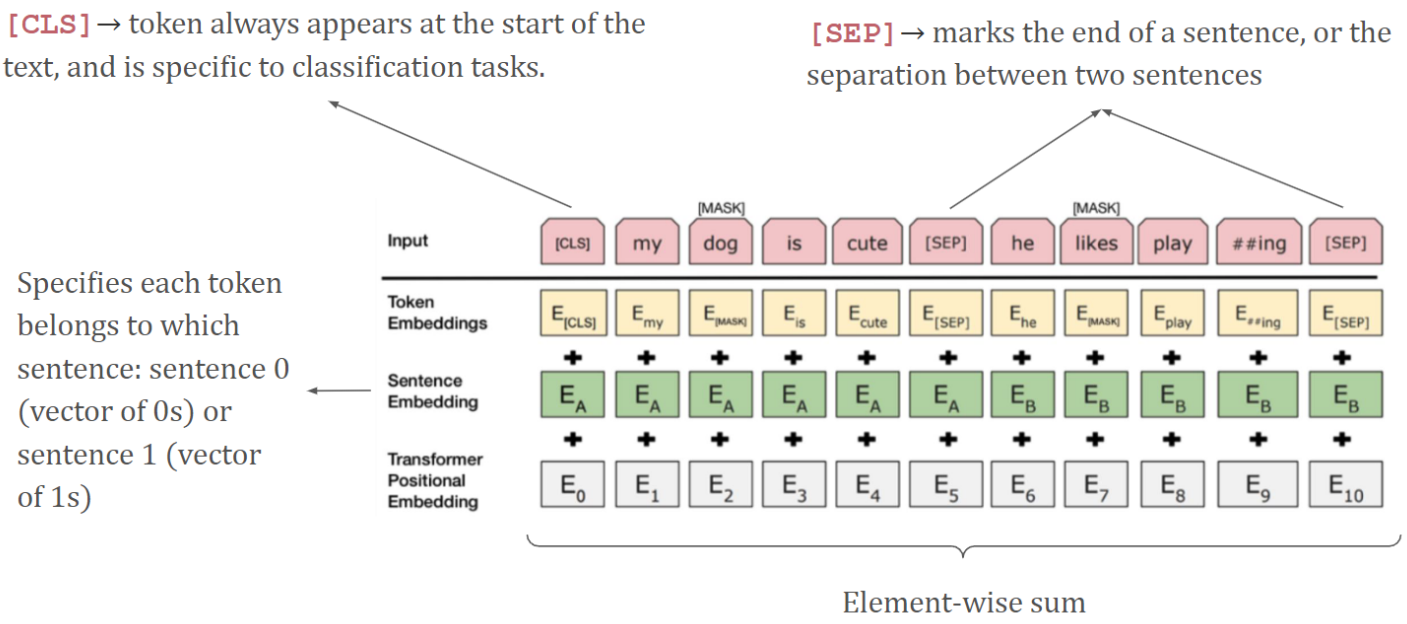


Figure 7.15: Input Embeddings

Task 1: Masked Word Prediction

- Replace **15% of words**, at random, with [MASK] token (think dropout)
- Using the context of non-masked words, **predict original value** of [MASK] token
- Loss is computed on just the masked word (contrast with next word prediction)

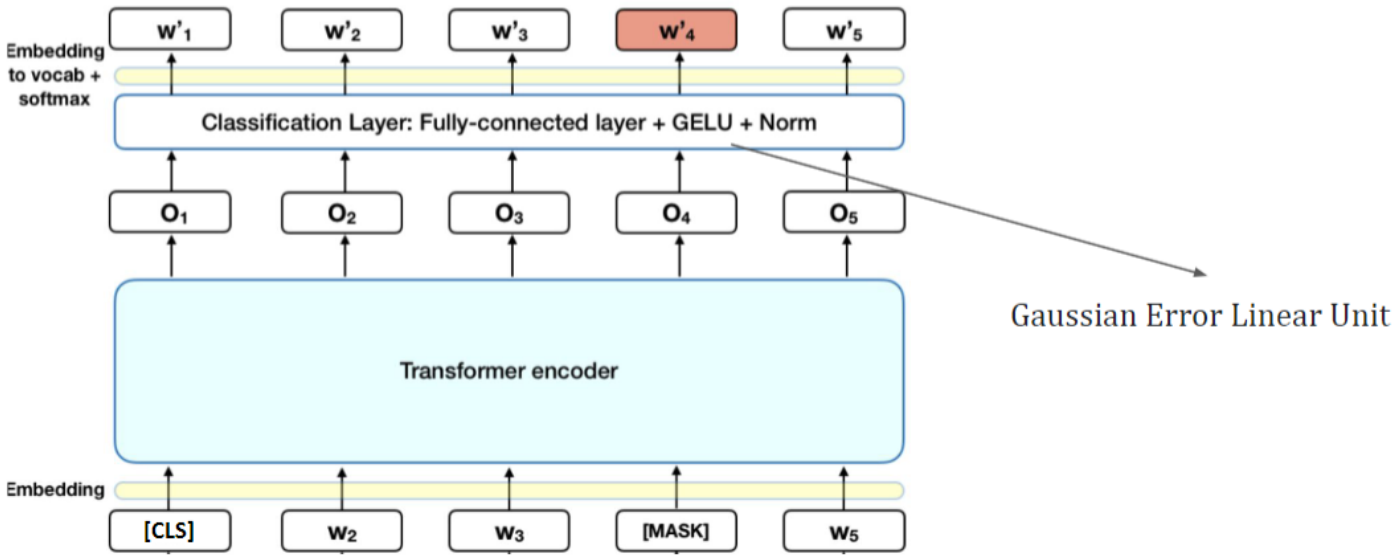


Figure 7.16: Masked Word Prediction

Task 2: Next Sentence Prediction

- Given two sentences, predict if they **appear together**.
- Create 50% positive and 50% negative pairs of sentences (512 tokens)

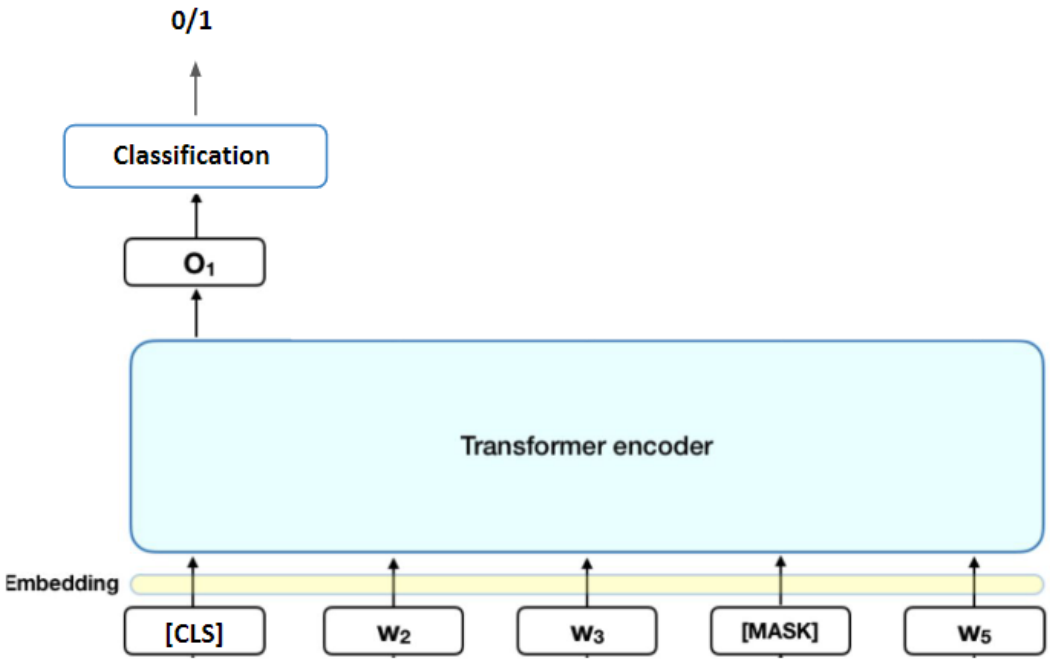


Figure 7.17: Next Sentence Prediction

Transfer Learning

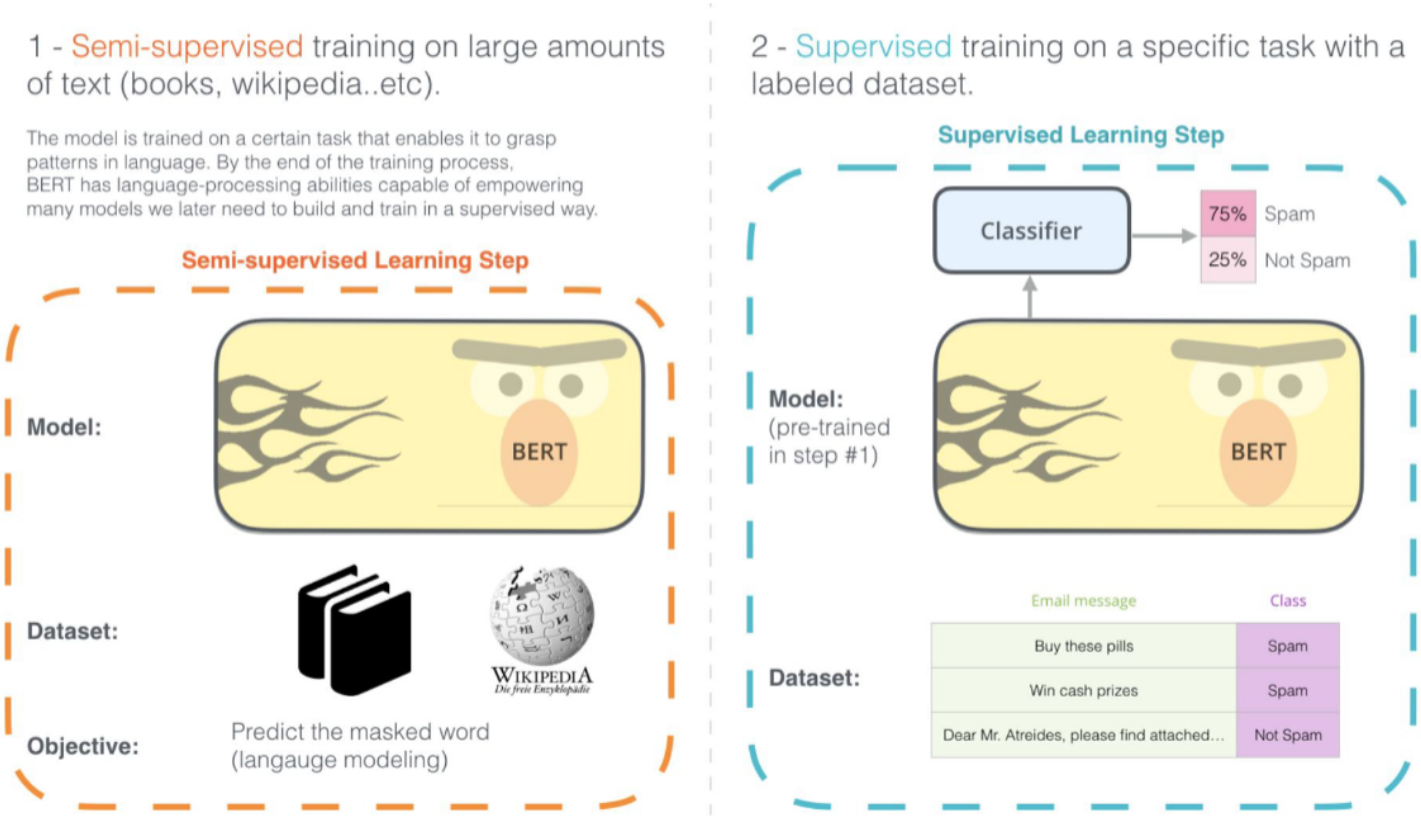


Figure 7.18: Transfer Learning

7.6 Computer Vision

ViT
Vision Transformers (ViT) are starting to dominate computer vision.
Compared to CNNs, they achieve **higher accuracies on large datasets** due to their:

- Higher modeling capacity
- Lower inductive biases
- Global receptive fields

CNNs are still on-par or better than ViTs on ImageNet in terms of model complexity or size versus accuracy.

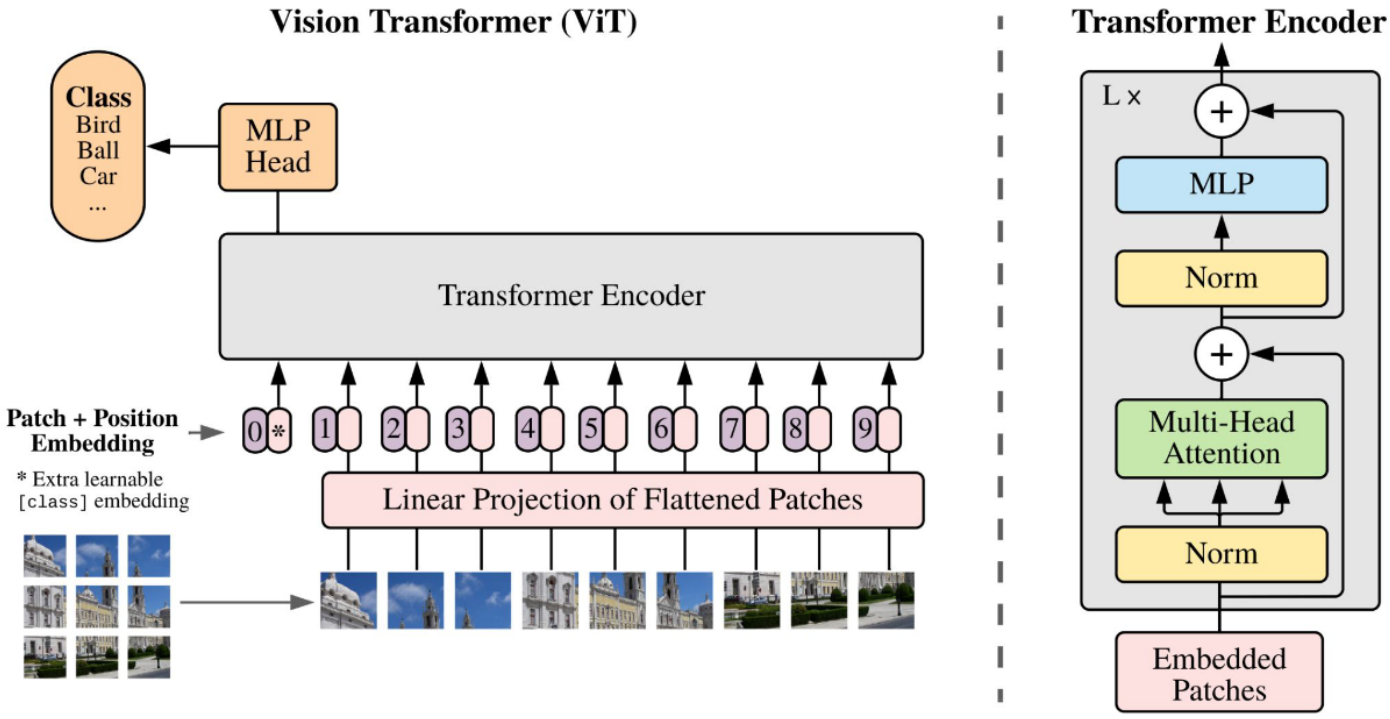


Figure 7.19: ViT

8 Graph Neural Networks (GNNs)

8.1 Motivation

- CNNs work well on images
- RNNs work well on sequences (one-dimensional)
- Both work on Euclidean spaces in R^n

Definition

Non-Euclidean Space: also known as curved or non-flat spaces, are mathematical spaces that do not follow the rules of Euclidean geometry. They exhibit different geometric properties compared to Euclidean space (R^n), which is flat and follows Euclid's axioms.

- We can use GNNs to model predictions for non-Euclidean spaces (usually graphs).

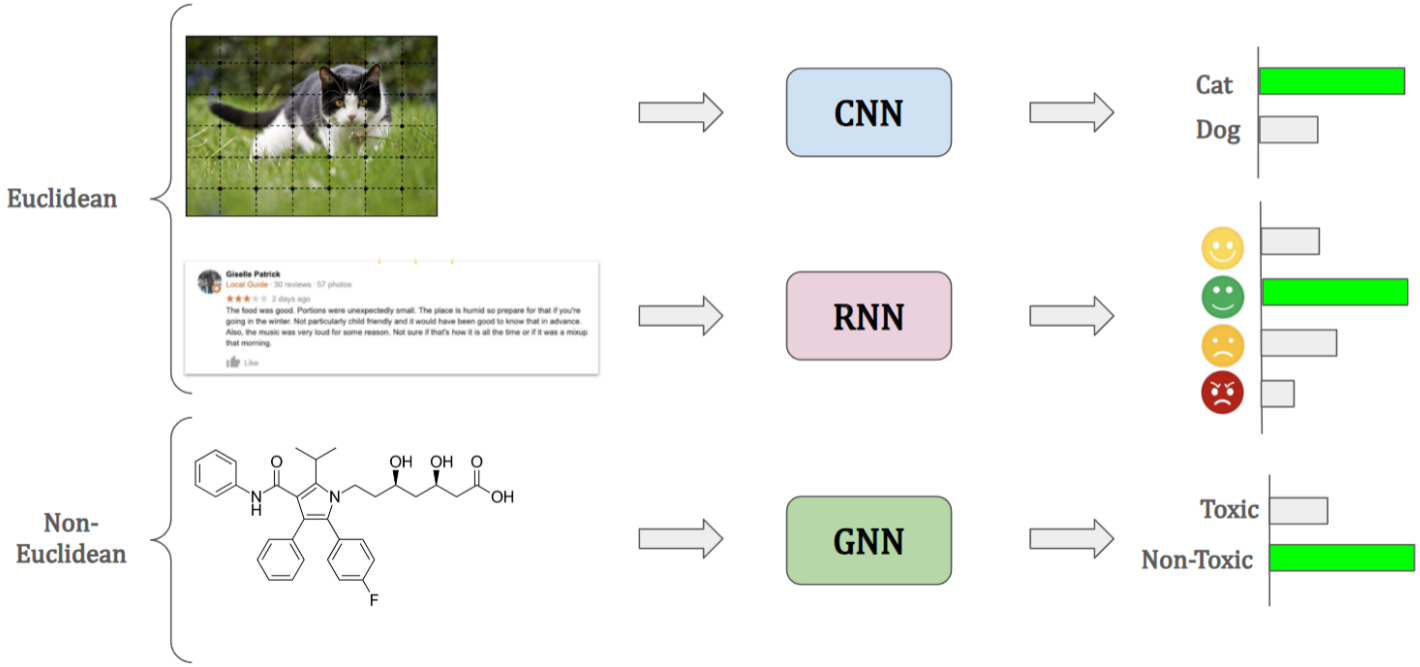


Figure 8.1: Why GNNs?

Definition

Graph: a mathematical structure consisting of nodes (vertices) and edges (connections) that represent relationships or connections between the nodes.

Figure 8.2: Examples of Graphs

8.2 Deep Sets

Sets
What happens if we **omit the Positional Encoding** from Transformers?

- The input will be treated as a set, and the learned representation won't change if you randomly shuffle the input tokens.
- This property is known as **order-invariance** or **permutation-invariance**.
- But when is this property useful?

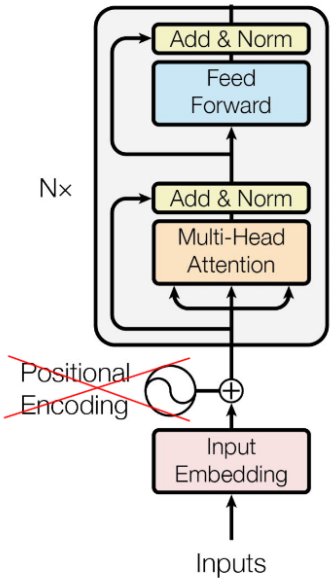


Figure 8.3: Omitting positional encoding from transformers

Data modalities that we have seen so far have **underlying order**. If we shuffle the order of:

- Pixels within an image
- Words within a sentence
- Frames within a video

- Signals within audio

...they will lose their meaning.

What if we want to train a model that predicts the likelihood of a set of items to be purchased by customers in a store?

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} x_3 \\ x_5 \\ x_1 \\ x_2 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_5 \\ x_3 \\ x_4 \\ x_2 \\ x_1 \end{bmatrix} = \ldots = \begin{bmatrix} x_4 \\ x_1 \\ x_2 \\ x_5 \\ x_3 \end{bmatrix}$$

$$f\left(\{\mathbf{x}_5, \mathbf{x}_1, \mathbf{x}_4, \mathbf{x}_2, \mathbf{x}_3\}\right) = \mathbf{y} = f\left(\{\mathbf{x}_2, \mathbf{x}_5, \mathbf{x}_4, \mathbf{x}_1, \mathbf{x}_3\}\right)$$

Figure 8.4: Model must be invariant to the order of the items

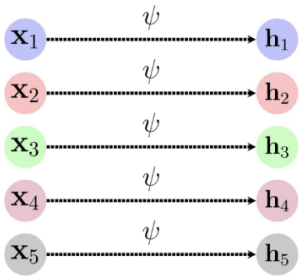
Idea

We want the model to ignore the order in which items are presented because the order is not essential and can confuse the model, similar to how shuffling training data helps prevent the model from learning irrelevant patterns.

Deep Sets

1. Learn embeddings for each item $\rightarrow$ Use a **shared neural network** ψ (e.g., MLP, Transformer, etc.) to project each item to a shared space.
2. Learn embeddings for the set $\rightarrow$ Use an **order-invariant aggregation function** such as sum, mean, max, etc. to aggregate the embeddings into a single embedding.
3. Use another **neural network** ϕ (e.g., MLP) to project the embedding to the final space.

$$\mathbf{h}_i = \psi(\mathbf{x}_i)$$



$$f(\mathbf{X}) = \phi\left(\sum_{i \in \mathbf{X}} \psi(\mathbf{x}_i)\right)$$

Classifier Encoder

Figure 8.5: Deep Sets

8.3 Graphs

Let's reiterate on Transformers without positional encoding:

- The transformer learns an NxN attention matrix which represent pairwise importance scores
- This means Transformer creates a fully-connected graph over the input and learns the edge weights

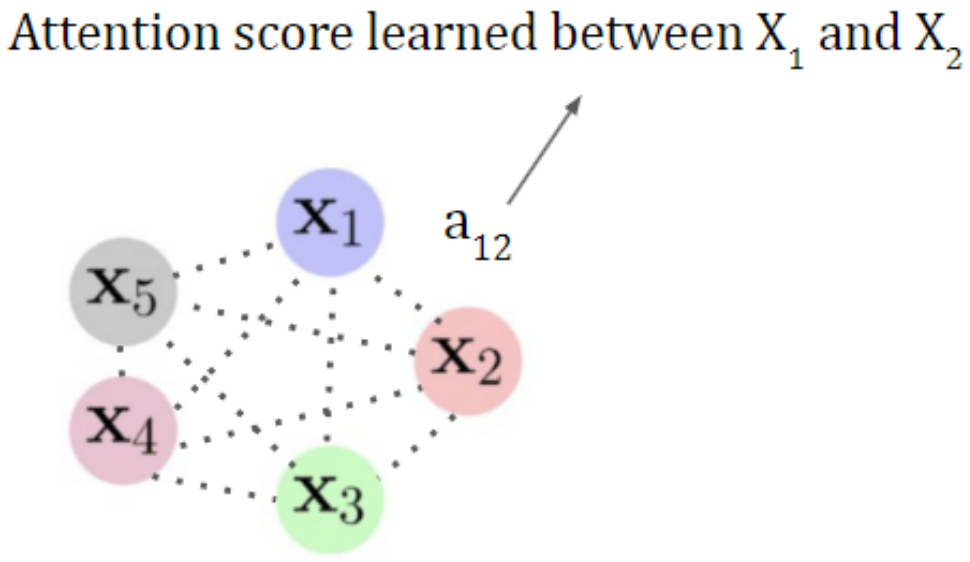


Figure 8.6: Learning edge weights (attention scores) in a fully-connected graph

Definition

Graph: G=(V, E, X) is a data-structure that encodes **pair-wise interactions or relations** among **concepts and objects**:

- V is set of nodes representing concepts or objects
- $E \subseteq V \times V$ is a set of edges connecting nodes and representing relations or interactions among them
- X encodes the node features of each node
- We can represent the edges in an **adjacency matrix A**:
- **Degree** of a node is number of edges connecting to that node

- Graphs are **order-invariant**: the naming or ordering of nodes may be shuffled and the graph won't change
- Functions on graphs must also be order-invariant

8.4 Graph Neural Networks

Definition

Graph Neural Networks: a class of deep learning models specifically designed to work with structured data represented as graphs. The key idea behind GNNs is to extend neural network architectures to handle data with a graph structure, such as social networks, recommendation systems, molecular chemistry, and more.

- GNNs are neural networks that function on graphs

Adjencymat.png

Figure 8.7: Adjacency matrix (1 = connection; 0 = no connection)

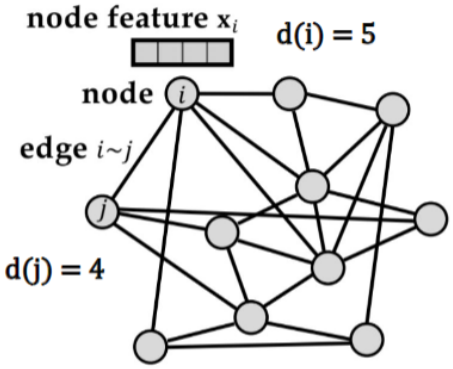


Figure 8.8: Graph

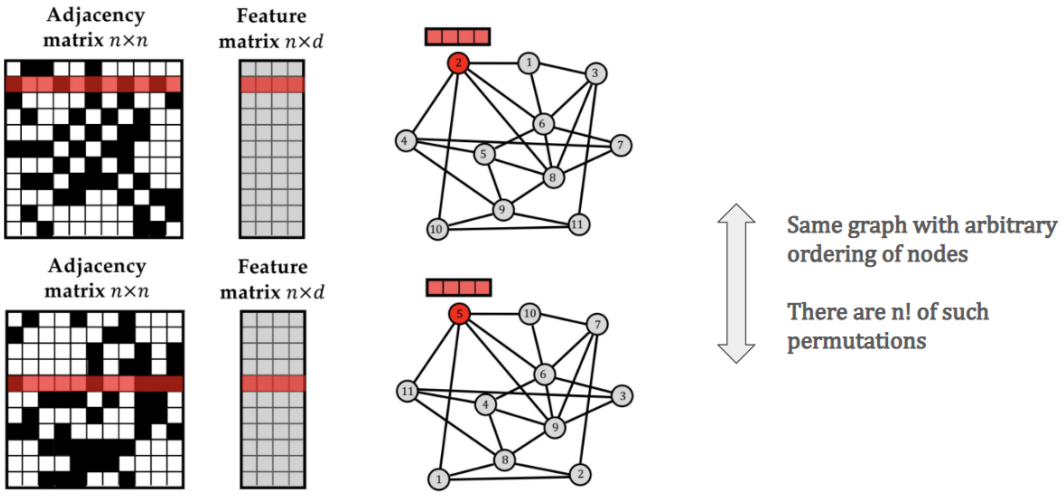


Figure 8.9: Indices and naming are arbitrary

- They are mostly based on **Message-Passing** $\rightarrow$ communicate with neighbors to update embeddings
- **Node classification**: we can use them to predict node classes

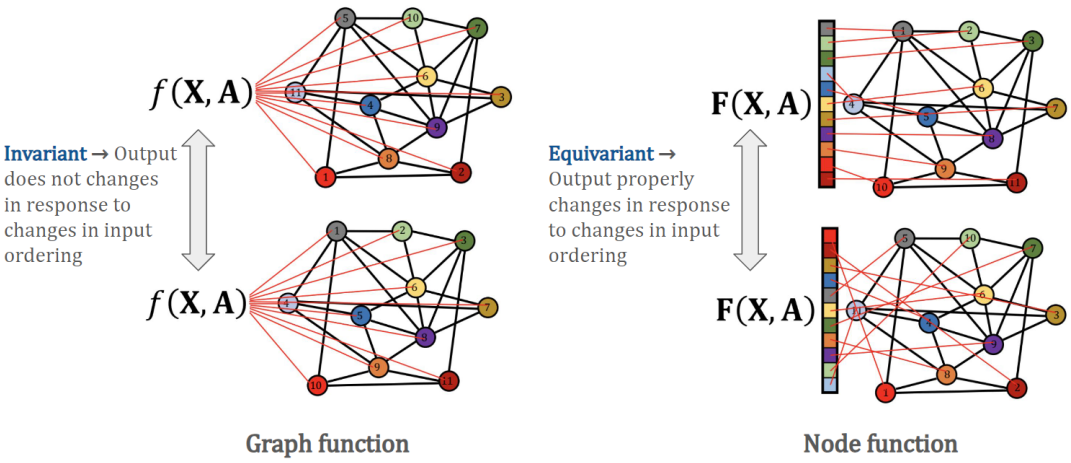


Figure 8.10: Invariance and Equivariance

  - Eg. Predicting atom types in molecule structure
- **Graph classification**: We can use them to predict graph classes
  - Eg. Predicting if a molecule structure is toxic
- **Link prediction**: We can use them to predict links between nodes
  - Eg. Predicting if there should be a bond between two atoms

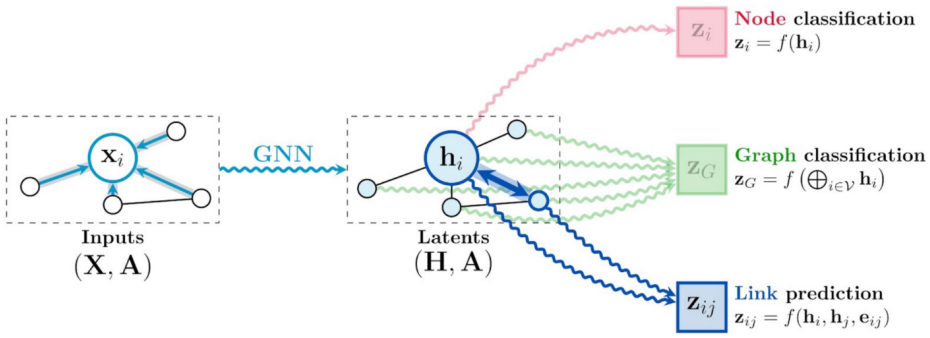


Figure 8.11: Message passing and application examples

Message Passing
For each node in graph:

1. **Aggregate** embeddings of its neighbor nodes
2. **Combine** the aggregated embedding with the node embedding
3. **Update** the node embedding

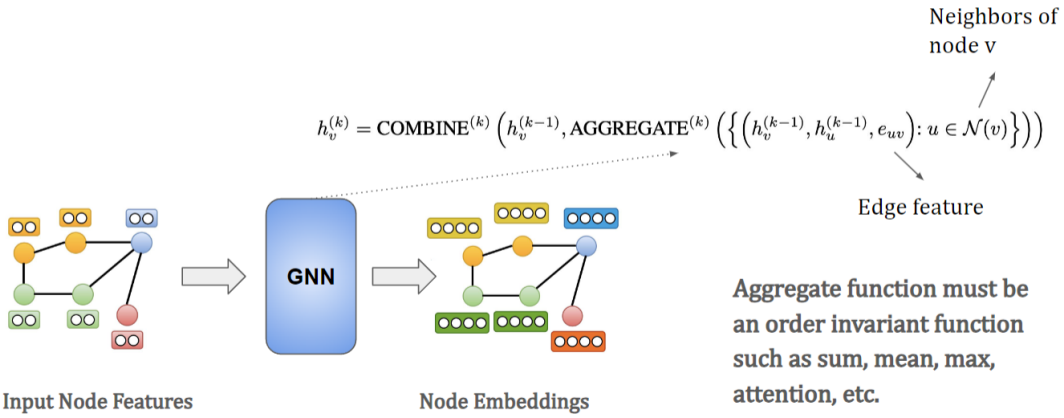


Figure 8.12: Message Passing

Read-out (graph pooling) function

- Aggregates all node embeddings into a graph embedding
- Must be an order invariant function such as sum, mean, max, attention, etc.

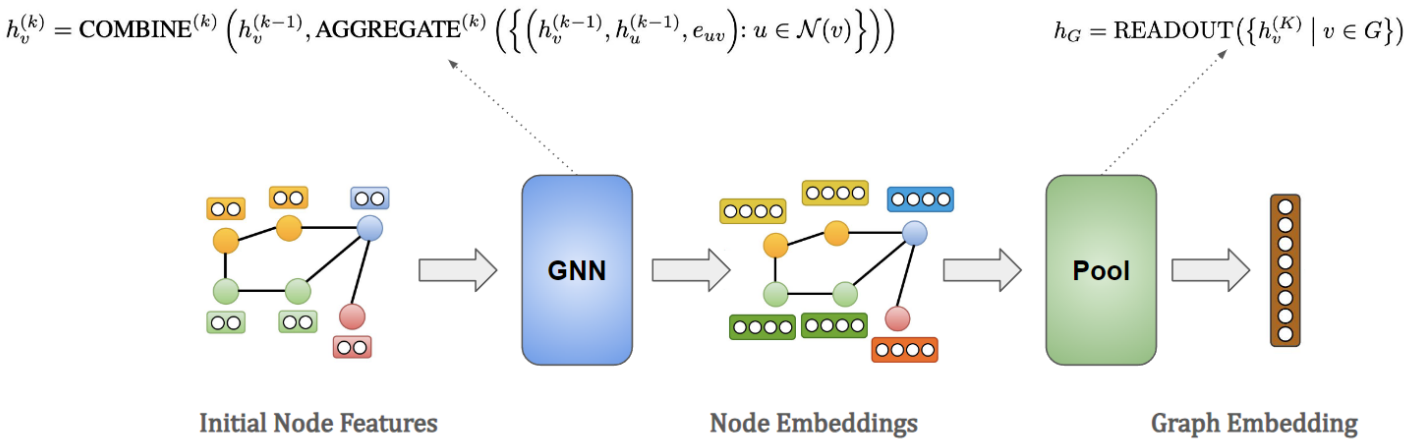


Figure 8.13: Read-out function

GNN Models

- Various ways of using aggregation functions lead to the creation of different Graph Neural Network (GNN) models.

8.5 Graph Convolutional Networks (GCNs)

- A layer of a GNN is basically a nonlinear function over node features and adjacency matrix:

$$H = f(A, X)$$

- The simplest model we can define is:

$$H = \sigma(AXW)$$

  - Where:
    - σ is the non-linearity function
    - W is the weight matrix

$$h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, \text{AGGREGATE}^{(k)}\left(\left\{\left(h_v^{(k-1)}, h_u^{(k-1)}, e_{uv}\right): u \in \mathcal{N}(v)\right\}\right)\right)$$

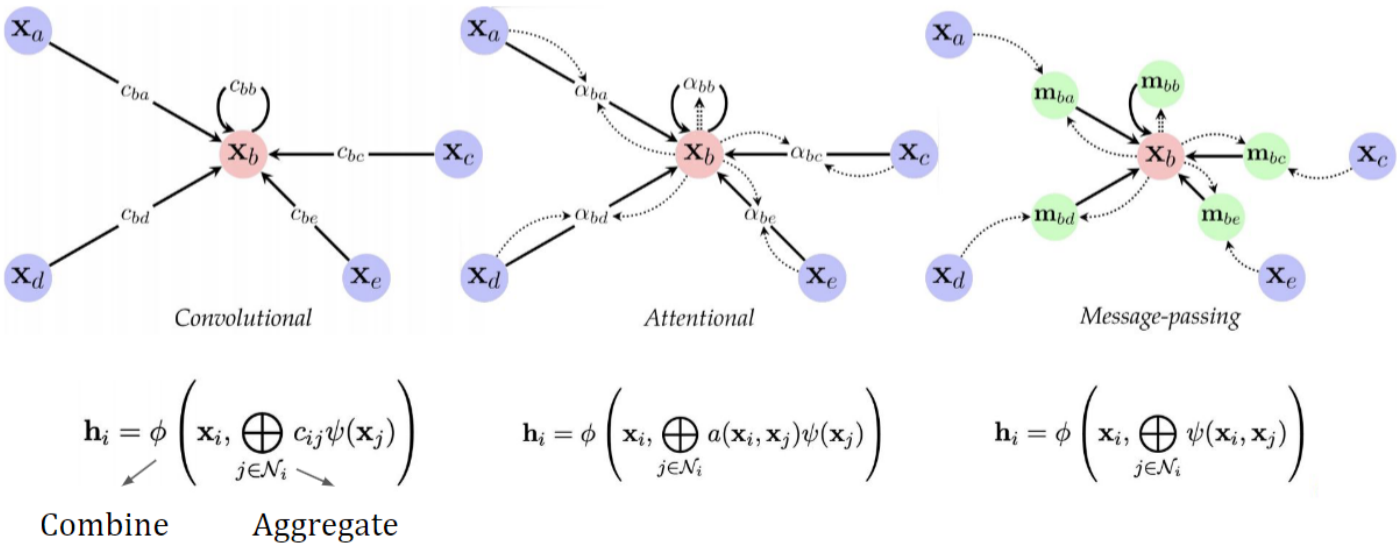


Figure 8.14: 1. **Convolutional**: Multiply importance of each neighbour by scalar and summing;
2. **Attentional**: Use attention mechanism to do weighted summation;
3. **Message-Passing**: Pass all embeddings between target node and neighbouring nodes computed by neural network, then aggregate

- This is already a strong model, but has two limitations:

1. Multiplication with A means that, for every node, we sum up all the feature vectors of all neighbouring nodes but not the node itself
   (a) Fix: Add self-loops (add the identity matrix to A)

$$A = A + I$$

2. A is not normalized and therefore the multiplication with A will completely change the scale of the feature vectors
   (a) Fix: Symmetrically normalize A using diagonal degree matrix D such that all rows sum to one

$$D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$$

- With these fixes we define a GCN layer as follows:

$$H = \sigma(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}XW)$$

Going Deeper

- A GCN layer updates the node embeddings based on the features of the immediate neighbors (recall multiplication with A)
- We can influence the embeddings from further neighborhood by stacking GCN layers
- This is analogous to increasing the receptive field in CNNS

8.6 Graph Attention Networks (GAT)

Idea

Instead of using node degree, learn an attention score between two nodes, i.e., learn the contribution weight of neighbor nodes

$$H^{(0)} = X$$
$$H^{(1)} = \sigma\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}H^{(0)}W^{(1)}\right)$$
$$H^{(2)} = \sigma\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}H^{(1)}W^{(2)}\right)$$
$$\cdots$$
$$H^{(l)} = \sigma\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}H^{(l-1)}W^{(l)}\right)$$

Node embeddings in layer 2 are computed based on contributions from 1-hop and 2-hop neighbors

Figure 8.15: Stacking GCNs

1. Use a shared neural network to compute an attention score between two nodes:

$$e_{ij} = NN(h_i, h_j)$$

2. Normalize the attention scores:

$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{exp(e_{ij})}{\sum_{k \in N_i} exp(e_{ik})}$$

3. Update the node embeddings based on the attention score:

$$h_i = \sigma(\sum_{j \in N_i} \alpha_{ij} W h_j)$$

8.7 PyTorch Implementation

Example

Suppose we want to classify the following molecule using a GCN to a toxic/non-toxic where node features represent the atom type (Carbon, Hydrogen, Nitrogen).

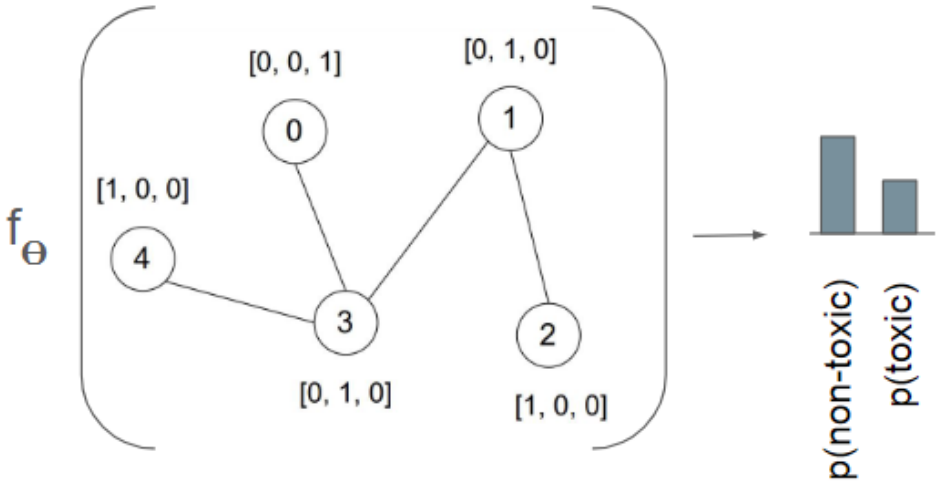


Figure 8.16: Toxicity classifier

Dense Implementation

- Most graphs are sparse

- Dense implementation uses N^2 space for adjacency
- We can represent this graph with only 4 edges where dense implementation represents it with 25
- Implementing sparse operations are supported by PyTorch but are not very straightforward
- We can use **PyTorch Geometric (PYG)**

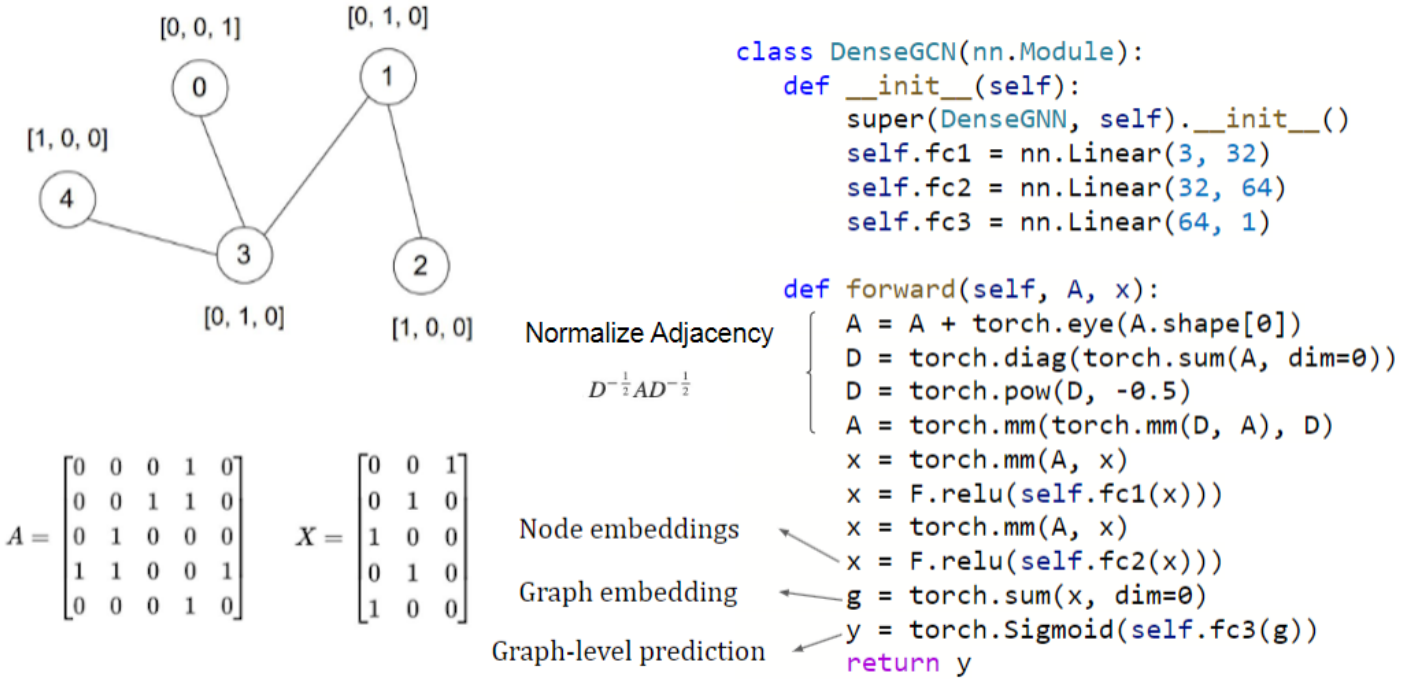
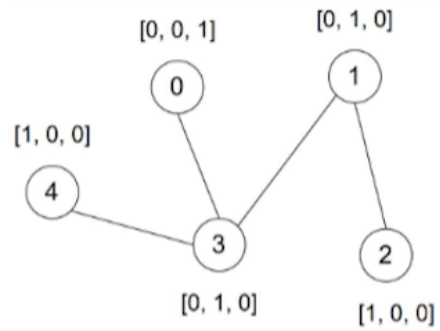


Figure 8.17: Dense Implementation

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1, 1, 2, 3, 3, 3, 4],
                           [3, 2, 3, 1, 0, 1, 4, 3]],
                          dtype=torch.long)
x = torch.tensor([[0, 0, 1],
                  [0, 1, 0],
                  [1, 0, 0],
                  [0, 1, 0],
                  [1, 0, 0]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index, y=[1.0])
```

Figure 8.18: Sparse implementation with PYG

Sparse Implementation



$$A = \begin{bmatrix} 0 & 1 & 1 & 2 & 3 & 3 & 3 & 4 \\ 3 & 2 & 3 & 1 & 0 & 1 & 4 & 3 \end{bmatrix}$$

$$X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

We can replace it with GATConv.

```python
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv

class SparseGCN(nn.Module):
    def __init__(self):
        super(SparseGCN, self).__init__()
        self.gcn1 = nn.GCNConv(3, 32)
        self.gcn2 = nn.GCNConv(32, 64)
        self.fc = nn.Linear(64, 1)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.relu(self.gcn1(x, edge_index))
        x = F.relu(self.gcn2(x, edge_index))
        g = torch.sum(x, dim=0)
        y = torch.Sigmoid(self.fc(g))
        return y
```

Figure 8.19: Sparse Implementation

Sparse implementation is better but dense implementation is more intuitive to implement.

Dense vs. Sparse: Computational efficiency

- Dense GNNs assume dense connectivity and involve **interactions between all nodes.**
- Sparse GNNs take advantage of the sparsity of the graph and **only consider interactions between connected nodes**, which is more efficient for sparsely connected graphs.

Dense vs. Sparse: DataLoader & Dataset

- Dense implementation: batching is done by creating a **diagonal matrix** of adjacency matrices
  - Large scale graph datasets will almost always run out of memory

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix}, \qquad \mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_n \end{bmatrix}$$

Figure 8.20: Batching in dense implementation

- Sparse implementation: uses an **index vector** which maps each node to its respective graph in the batch
  - Very simple and efficient

$$\text{batch} = [0 \quad \cdots \quad 0 \quad 1 \quad \cdots \quad n-2 \quad n-1 \quad \cdots \quad n-1]^\top$$

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 2 \\ 2 \end{bmatrix}$$

Nodes 0-2 → Graph 0

Nodes 3-6 → Graph 1

Nodes 7-8 → Graph 2

Figure 8.21: Batching in sparse implementation

```python
from torch_geometric.datasets import TUDataset
from torch_geometric.loader import DataLoader

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES')
loader = DataLoader(dataset, batch_size=32, shuffle=True)
for data in loader:
    print(data)
    break

>>> DataBatch(batch=[1082], edge_index=[2, 4066], x=[1082, 21], y=[32])
```

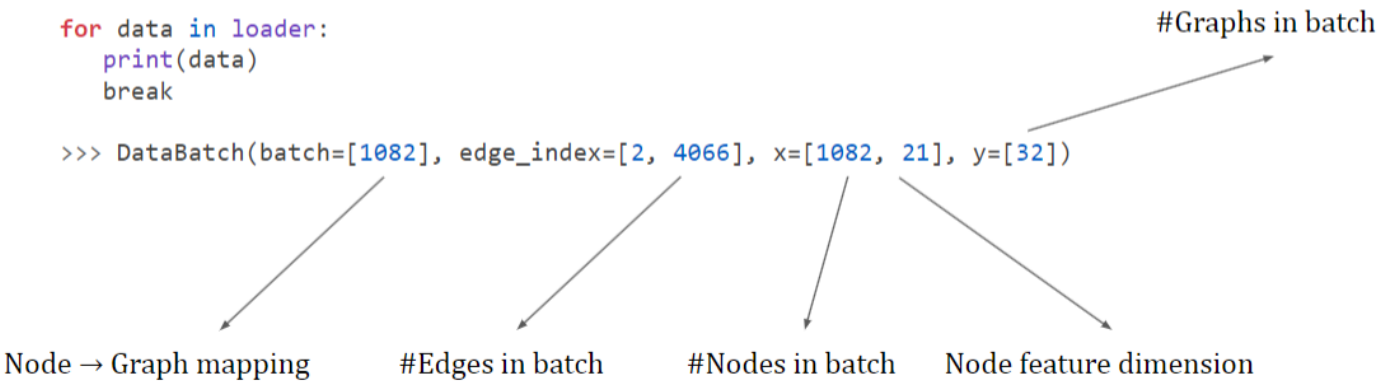


Figure 8.22: Dataloader & Dataset