# Generative AI Foundations

## Algorithms and Architectures

**Ashkan Jasour**

2024-2025

# Introduction

# Generative AI:

Artificial Intelligence systems that are designed to **generate new content**—such as text, images, music, code, or data—rather than just making predictions or classifications.

- **Generative AI** tries to model the data distribution so it can generate new samples from that learned distribution.
- **Traditional neural networks** aim to fit a function that maps inputs to correct outputs (e.g., image → label), often through supervised learning.

| Feature | Generative AI | Traditional Neural Networks |
|---|---|---|
| **Primary Goal** | Generate new content | Predict or classify existing input |
| **Output Type** | Novel data samples (text, image, etc.) | Labels, scores, or fixed outputs |
| **Examples** | GPT, DALL·E, VAE, GANs, Diffusion models | CNNs for classification, LSTMs for forecasting |
| **Training Objective** | Learn data distribution to generate new samples | Learn a function that maps inputs to correct outputs |
| **Architecture Examples** | Transformers, GANs, VAEs, Diffusion Models | CNNs, RNNs, LSTMs, MLPs |
| **Use Cases** | Writing, drawing, designing, simulating | Face recognition, speech recognition, forecasting |
| **Creativity** | Yes — generates new content | No — only analyzes/acts on input data |

➢ The first part provides different Generative AI **algorithms**, while the second part details the underlying neural network **architectures** used to construct these models.

- **Algorithms** refer to the underlying methods or modeling approaches that define how data is generated. They capture the statistical patterns of data and use them to synthesize new content.
  Examples include:
  - Autoregressive Models – generate outputs one step at a time (e.g., GPT for text)
  - Variational Autoencoders (VAEs) – learn latent representations and reconstruct data
  - Denoising Diffusion Models – generate data by gradually reversing noise
  - Generative Adversarial Networks (GANs) – use a generator vs. discriminator game
  - Normalizing Flows / ODE-based / SDE-based flows – model complex distributions using invertible transformations

  These algorithms define the type of generative process being modeled and how the probability distributions of data are learned or approximated.

- **Architectures** refer to the specific neural network structures or building blocks used to implement those algorithms. They define how the model is organized—its layers, connections, and flow of information. Common architectures include:
  - Transformers – widely used in text, image, and code generation tasks due to their powerful attention mechanism
  - CNNs (Convolutional Neural Networks) – excel at spatial data like images
  - U-Net – a convolutional architecture especially useful in image and video generation (used in diffusion models)
  - RNNs (Recurrent Neural Networks), LSTM (Long Short-Term Memory), and GRU (Gated Recurrent Unit)– process sequences, used in earlier generative models
  - Diffusion Transformers (DiT), Vision Transformers (ViTs), Attention-Based U-Net, VQ-VAE, Graph Neural Networks – more specialized architectures depending on data type

**Algorithms**: *what* the model is trying to do (e.g., generate step-by-step, denoise, sample from latent space)
**Architectures**: *how* the model is physically structured to do it (e.g., using attention layers, convolutional blocks, etc.)

# Topics:

## ➢ Generative AI - Algorithms

- Flow Models
- Ordinary Differential Equation (ODE)-based Flow Models
- Denoising Diffusion Models (DDMs)
- Stochastic Differential Equation(SDE)-based Denoising Diffusion Models
- Autoencoders and Variational Autoencoders (VAEs)
- Latent Space Diffusion Models
- Autoregressive Models
- Generative Adversarial Networks (GANs)

## ➢ Generative AI - Architectures

- Multilayer Perceptrons (MLPs)
- Training and Loss Functions Types
- Backpropagation Algorithm, Stochastic Gradient Descent (SGD), and Adam Optimizer
- Common Training Issues, Regularization in Deep Learning, and Scaling Laws for Deep Learning
- Convolutional Neural Networks (CNNs) and PixelCNN
- U-Net Denoising Model
- Recurrent Neural Networks (RNNs), LSTM (Long Short-Term Memory), and GRU (Gated Recurrent Unit)
- Transformers: Self-Attention, Multi-Head Attention, and Cross-Attention
- Diffusion Transformers (DiTs), Vision Transformers (ViTs), and Attention-Based U-Nets
- Multimodal Models
- Foundation Models

# Appendices:

- Key Differential Equations in Generative AI
- Fine-tuning Large Language Models
- Deep Reinforcement Learning - Key Concepts and Summary
  - PG, VPG, PPO, DDPG, TD3, SAC
- Reinforcement Learning from Human Feedback (RLHF) and Imitation Learning
- Adversarial Training, Robustness in Language Models, and Language Models Evaluation
- Python Libraries for Generative AI

# Generative AI - Algorithms

# Generative AI Algorithms - Overview

- **Flow Models**
  - Likelihood-based Model: Explicitly models the probability density of the data
  - Transforms noise (latent space) into data by learning neural-based invertible, differentiable function (flow)
  - It leverages the **Change of Variables Formula** to transform probability densities when mapping between data and latent space
  - Loss function: KL divergence (maximize the likelihood of the data under the learned distribution where the learned probability density is represented in terms of the learned function and the probability density of the latent noise distribution)
- **ODE-based Flow Models**
  - Continuous-time version of the flow model
  - It uses the neural ODE to construct the mapping between the latent noise and the data
  - It leverages the **Continuity Equation** that governs the time-evolution of a probability distribution through an ODE
  - **Flow Matching**: This method learns the velocity field that describes how data evolves over time
  - Loss function: Measures the squared difference between the learned velocity field and sampled estimates of the true analytical probability flow field over time. For Gaussian vector field cases, this reduces to learning the data subtracted by noise.
- **Denoising Diffusion Models**
  - Transforms noise into data iteratively by learning to predict and remove the noise
  - It leverages discrete-time **Langevin dynamics** to iteratively refine noisy samples using the predicted noise, gradually denoising them to generate high-quality samples. Loss function: Measures the squared difference between the actual noise and the predicted noise.
- **SDE-based Denoising Diffusion Models**
  - Stochastic version of the ODE-based Flow model
  - It uses the neural SDE to construct the mapping between the latent noise and the data
  - It leverages the **Fokker-Planck PDE** that governs the time-evolution of a probability distribution through an SDE
  - **Score Matching**: This method learns the score function, which is the gradient of the log-probability density. It points in the direction where the data distribution increases the most, guiding probability flow in generative models
  - Loss function: Measures the squared difference between the learned score function and sampled estimates of the true analytical score function. For Gaussian vector field cases, this reduces to learning the noise that perturbed the data.

## ➢ Generative AI Algorithms - Overview

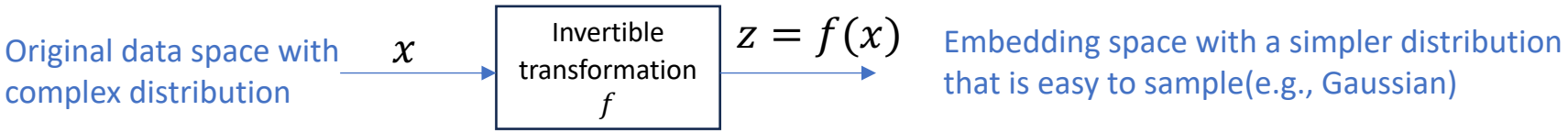
- **Variational Autoencoders (VAEs)**
  - Likelihood-based Model
  - Latent Space Representation: Maps data to a lower-dimensional latent space using a learned encoder and reconstructs it using a learned decoder.
  - The encoder receives a data sample and outputs the mean and variance of the approximate posterior distribution (modeled as a Gaussian) over the latent space, given the observed data. The decoder takes a sampled latent vector from this distribution and reconstructs the data.
  - Uses Variational Inference: Instead of directly optimizing the likelihood, it optimizes the **Evidence Lower Bound (ELBO)**, which provides a tractable approximation of the log-likelihood. This results in two main loss components: Reconstruction Loss and Regularization Loss.
  - Loss function: Reconstruction Loss which ensures that the reconstructed data closely matches the input by maximizing the log-likelihood of the data given the latent representation. Regularization Loss where uses KL Divergence to minimize the difference between the learned latent distribution and the known prior distribution (e.g., Normal), regularizing the latent space.
- **Autoregressive Models**
  - Likelihood-based Model
  - No Latent Variables: Unlike VAEs and Flow Models, autoregressive models directly model data without requiring a separate latent space.
  - Sequential Data Generation: Generates data one step at a time, where each new element depends on previously generated elements.
  - Uses the **Chain Rule of Probability** that ensures that each output is conditioned on past values
  - It learns the conditional probability distribution of the next token given the sequence of previously generated elements
  - Loss function: Cross-entropy loss between the predicted and true next-token distributions, optimizing for maximum likelihood in sequential data generation.
- **Generative Adversarial Networks (GANs)**
  - Implicit Generative Model: GANs are implicit generative models that learn to synthesize data resembling real-world samples without explicitly modeling likelihoods.
  - Adversarial Training: Consists of two neural networks—a **Generator** (G) and a **Discriminator** (D)—that compete against each other in a **min-max game**.
  - Generator takes random noise as input and produces synthetic samples while Discriminator tries to distinguish real data from fake (generated) data.
  - Loss Function: Uses binary cross-entropy loss, where the Discriminator is trained to distinguish real from fake samples, and the Generator is trained to fool the Discriminator by maximizing the probability of fake samples being classified as real.

# Flow Models


- Papamakarios, G., Nalisnick, E., Rezende, D. J., Mohamed, S., & Lakshminarayanan, B,"Normalizing flows for probabilistic modeling and inference", Journal of Machine Learning Research, 22(57), 1-64, 2021. https://jmlr.org/papers/volume22/19-1028/19-1028.pdf
- Pieter Abbeel , CS294-158-SP24 Deep Unsupervised Learning Spring, UC Berkeley, 2024, https://sites.google.com/view/berkeley-cs294-158-sp24/home
- Foster, David. Generative deep learning. " O'Reilly Media, Inc.", 2022.
- RealNVP: "Density Estimation using Real NVP", Laurent Dinh, Jascha Sohl-Dickstein, Samy Bengio, ICLR, 2017, https://arxiv.org/abs/1605.08803
- Glow: "Glow: Generative Flow with Invertible 1x1 Convolutions", Diederik P. Kingma, Prafulla Dhariwal, NeurIPS, 2018, https://arxiv.org/abs/1807.03039
- Flow++: "Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design", Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, Pieter Abbeel, ICM, 2019, https://arxiv.org/abs/1902.00275
- FFJORD: "FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models", Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, David Duvenaud, ICLR, 2019, https://arxiv.org/abs/1810.01367

# Flow Models:

- **Goal:** Learn an Invertible transformation from data space (with complex distribution) to embedding space (with simple distribution, e.g., Gaussian)




- Hence, given the transformation $f$, we can sample embedding space (e.g., sample from Gaussian) and then transform it into a data $x = f^{-1}(z)$
- We will learn the transformation using a neural network $f_\theta$ with parameters $\theta$ such that $x = {f_\theta}^{-1}(z) \sim p_\theta(x)$ achieves the actual distrbution of the data $p^*(x)$; Hence samples of $z$ transformed by ${f_\theta}^{-1}(z)$ will sample data distribution.
  **Note:** $x$ and $z$ have the same dimensions.

- **Training:** learn parameters $\theta$ to minimize the distance between $p_\theta(x)$ (distribution of data parametrized with $\theta$) and actual distribution of the data $p^*(x)$ (KL divergence)

$$\min_\theta \text{KL Divergence} = \max_\theta \text{Liklihood of Samples}$$

fixed

$$\text{KL divergence: } \min_\theta D(p^* || p_\theta) = \int_{-\infty}^{+\infty} p^*(x) \log \frac{p^*(x)}{p_\theta(x)} dx = \mathrm{E}_{x\sim p^*}\left[\log \frac{p^*(x)}{p_\theta(x)}\right] = \mathrm{E}_{p^*}[\log p^*(x)] - \mathrm{E}_{p^*}[\log p_\theta(x)] = -\mathrm{E}_{x\sim p^*}[\log p_\theta(x)]$$

$$\approx -\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(x_i) = \max_\theta \frac{1}{N}\sum_{i=1}^{N} \log p_\theta(x_i) \quad \text{:maximum likelihood}$$

data samples

- **Sampling:** first sample $z$ then compute $x = {f_\theta}^{-1}(z)$

# Flow Models:

➢ **Data Parametric Distribution $p_\theta(x)$** :To construct $p_\theta(x)$ in terms of neural network $f_\theta$, we will use Change of Variables Formula.

- Change of Variables Formula $f: \mathbb{R} \to \mathbb{R}$: $p_\theta(x) = p_z(f_\theta(x)) \left| \underbrace{\frac{\partial f_\theta(x)}{\partial x}}_{\text{Jacobian}} \right|$ ( obtained by $z = f_\theta(x)$ and conservation of probability $\int_X p_x(x)dx = \int_Z p_z(z)dz$ )

- Change of Many Variables Formula $f: \mathbb{R}^n \to \mathbb{R}^n$: $p_\theta(x) = p_z(f_\theta(x)) \left| \underbrace{\det \frac{\partial f_\theta(x)}{\partial x}}_{\text{Jacobian determinant}} \right|$

➢ **Training:**

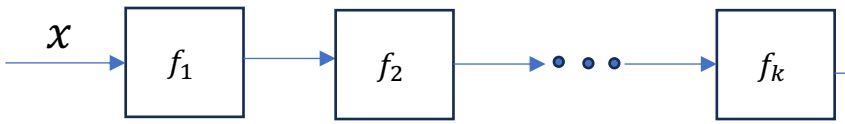
$$\max_\theta \frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x_i) = \max_\theta \frac{1}{N} \sum_{i=1}^{N} \log p_z(f_\theta(x_i)) + \log \left| \frac{\partial f_\theta(x_i)}{\partial x} \right|$$

- Assuming we have an expression for $p_z$, this can be optimized with Stochastic Gradient Descent.

$$\max_\theta \frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x_i) = \max_\theta \frac{1}{N} \sum_{i=1}^{N} \log p_z(f_\theta(x_i)) + \log \left| det \frac{\partial f_\theta(x_i)}{\partial x} \right|$$

- The Jacobian determinant must be computationally tractable to calculate and differentiate.

➢ **Composition of Flows:** to increase expressiveness




$$z = f_k \circ \ldots \circ f_1\ (x)$$

$$x = f_1^{-1} \circ \ldots \circ f_k^{-1}\ (z)$$

$$\log p_\theta(x) = \log p_z(f_k \circ \ldots \circ f_1\ (x)) + \sum_{i=1}^{k} \log \left| det \frac{\partial f_i(x)}{\partial f_{i-1}} \right|$$

where $f_0(x) = x$

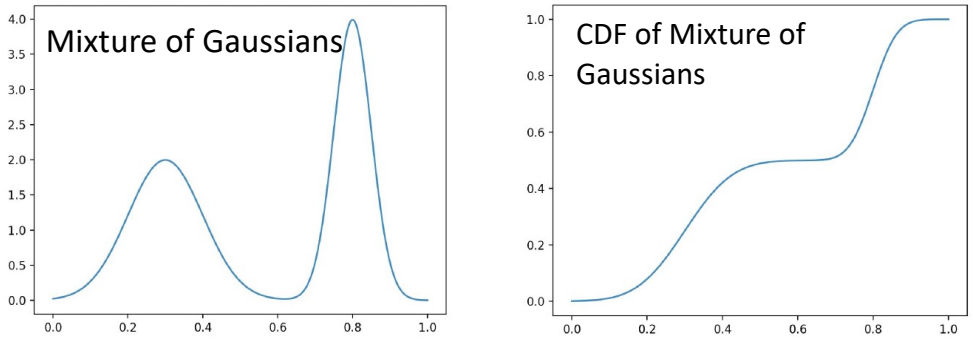
➢ Examples of flow-based generative models: **NICE, RealNVP, Glow, Flow++**

➢ **Transformation Mapping $f_\theta(x)$**

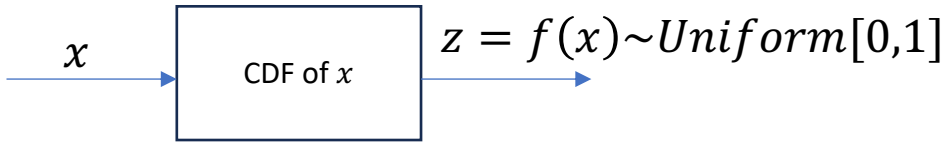
- **1-D Case:** $f: \mathbb{R} \to \mathbb{R}$ Differentiable and invertible functions
- **N-D case:** $f: \mathbb{R}^n \to \mathbb{R}^n$ Differentiable and invertible function + The Jacobian determinant must be easy to calculate and differentiate**.**
  - Note that neural networks are, in general, not invertible. So, to construct a neural mapping we can choose invertible functions whose parameters are produced by a neural network. While the neural network itself doesn't need to be invertible, the overall transformation is constructed to be invertible, e.g., in affine coupling layers, a neural network outputs the scale and translation parameters used in the transformation.

**1-D Example:**

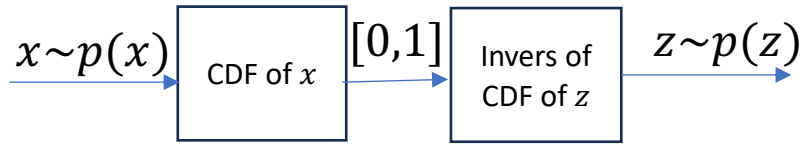
- We can define the flow as a parameterized CDF, e.g., CDF of mixture of Gaussians with unknown weights, means, and variances. These unknow parameters can be learned by neural networks.




The CDF is an invertible, differentiable map from data to [0, 1]; Hence, CDF turns any density into uniform




CDF can turn any (smooth) distribution $p(x)$ into any (smooth) distribution $p(z)$



## n-D Example 1: Autoregressive Flow Model

$$x_1 \sim p_{\theta_1}(x_1)$$
$$x_2 \sim p_{\theta_1}(x_2|x_1)$$
$$\vdots$$
$$x_n \sim p_{\theta_1}(x_n|x_1, x_2, \dots, x_{n-1})$$

$$z_1 = f_{\theta_1}(x_1)$$
$$z_2 = f_{\theta_2}(x_1, x_2)$$
$$\vdots$$
$$z_n = f_{\theta_n}(x_1, x_2, \dots, x_n)$$

$$x_1 = f_{\theta_1}^{-1}(z_1)$$
$$x_2 = f_{\theta_2}^{-1}(z_2; x_1)$$
$$\vdots$$
$$x_n = f_{\theta_n}^{-1}(z_n; x_1, x_2, \dots, x_{n-1})$$

Invertible function of $z_n$,
$x_1, \dots, x_{n-1}$ are parameters and can have arbitrary complexity (including any neural net) and no invertibility requirement (example in the next page )

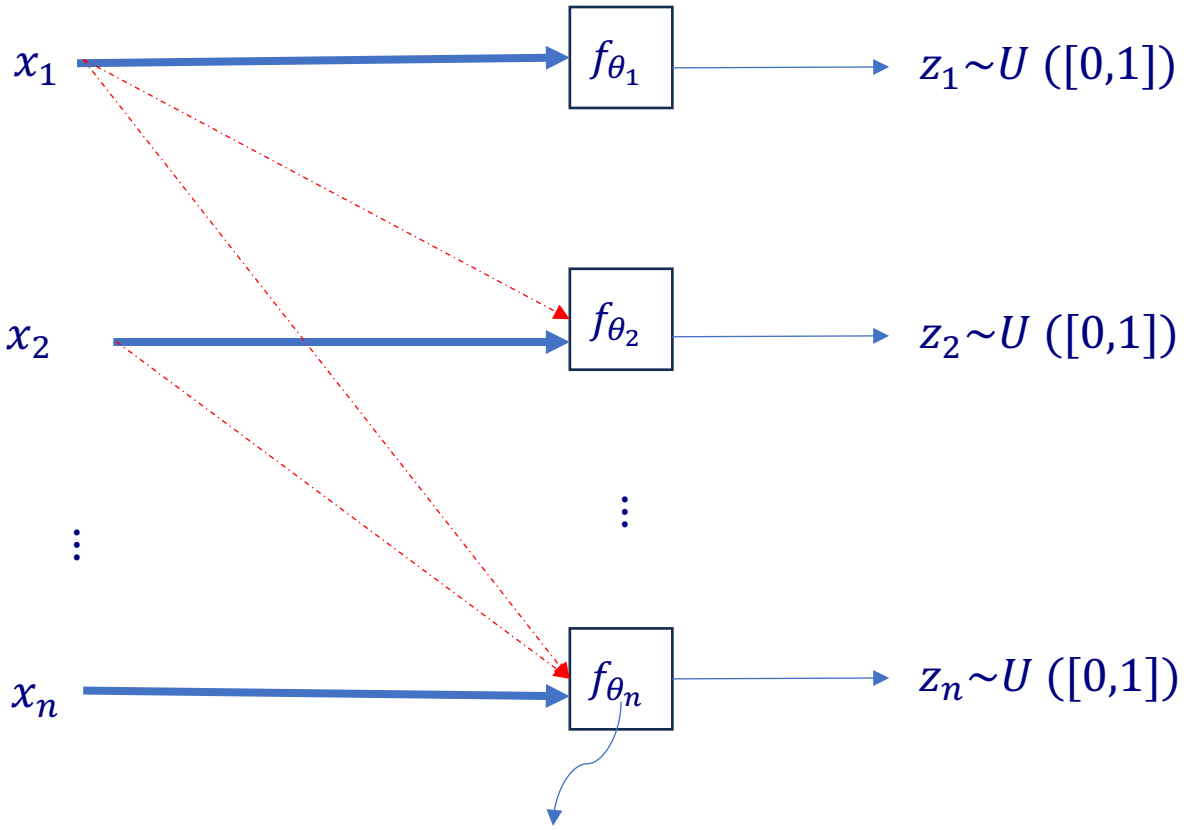
The **Jacobian is triangular**, so its determinant is the product of diagonal entries

$$p(x_1, x_2, \dots, x_n) = p(z_1)p(z_2) \dots p(z_n) \left| \frac{\partial z_1(x_1)}{\partial x_1} \frac{\partial z_2(x_1, x_2)}{\partial x_2} \dots \frac{\partial z_n(x_1, x_2, .. x_n)}{\partial x_n} \right|$$

$$\log p(x_1, x_2, \dots, x_n) = logp(z_1) + logp(z_2) + \dots + logp(z_n) + log \left| \frac{\partial z_1(x_1)}{\partial x_1} \right| + log \left| \frac{\partial z_2(x_1, x_2)}{\partial x_2} \right| + \dots + log \left| \frac{\partial z_n(x_1, x_2, .. x_n)}{\partial x_n} \right|$$

## n-D Example 1: Autoregressive Flow Model

- We choose invertible functions (Mixture of Gaussian CDFs ) whose parameters are produced by a neural network.




$f_{\theta_1}$: Mixture of Gaussian CDFs in $x_1$

$\theta_1 = [w_{11}, \mu_{11}, \sigma_{11}, \dots, w_{1m}, \mu_{1m}, \sigma_{1m}]$

$f_{\theta_2}$: Mixture of Gaussian CDFs in $x_2$

$[w_{21}, \mu_{21}, \sigma_{21}, \dots w_{2m}, \mu_{2m}, \sigma_{2m}] = NN_{\theta_2}(x_1)$

$f_{\theta_n}$: Mixture of Gaussian CDFs in $x_n$

$[w_{n1}, \mu_{n1}, \sigma_{n1}, \dots w_{nm}\mu_{nm}, \sigma_{nm}] = NN_{\theta_n}(x_1, x_2, \dots, x_{n-1})$

$f_{\theta_n}$: Mixture of $m$ Gaussian CDFs in $x_n$. The unknown parameters of the CDFs including $m$ weights, means, and variances $[w_{n1}, \mu_{n1}, \sigma_{n1}, \dots w_{nm}\mu_{nm}, \sigma_{nm}]$ are produced by a neural network whose inputs are $x_1, x_2, \dots, x_{n-1}$.

$$[w_{n1}, \mu_{n1}, \sigma_{n1}, \dots w_{nm}\mu_{nm}, \sigma_{nm}] = NN_{\theta_n}(x_1, x_2, \dots, x_{n-1})$$

## n-D Example 2: Affine Coupling Layers

- Split the data dimensions in two parts: $[x_1, \dots x_d\ ]$ and $[x_{d+1}, \dots x_n\ ]$

$$z_{1:d} = x_{1:d}$$
$$z_{d+1:n} = x_{d+1:n} \odot \exp(s_\theta(x_{1:d})) + t_\theta(x_{1:d})$$

element-wise multiplication

scale and translation parameters modeled with neural networks

- Neural network outputs the scale and translation parameters used in the transformation.

- Affine coupling layers split the input and only transform a part, making the Jacobian triangular and easy to invert.

$$\frac{\partial z}{\partial x} = \begin{bmatrix} I & 0 \\ \frac{\partial z_{d+1:n}}{\partial x_{1:d}} & diag(\exp(s_\theta(x_{1:d}))) \end{bmatrix} \Longrightarrow \det \frac{\partial z}{\partial x} = diag(\exp(s_\theta(x_{1:d})))$$

## ➢ Architectures for Normalizing Flow models:

| Model | Key Idea | Advantages | Challenges |
|---|---|---|---|
| **RealNVP** (Real-valued Non-Volume Preserving) | Uses **affine coupling layers** for invertibility. | Efficient forward and inverse pass. Works well for image generation. | Requires careful design of masking patterns. |
| **Glow** | Uses **invertible 1x1 convolutions** instead of fixed masking patterns. | Improves expressiveness of transformations, easier to train. | Still limited in learning very complex distributions. |
| **Flow++** | Introduces **mixture of logistic distributions** and **variational dequantization**. | More powerful than RealNVP and Glow for images. | More computationally expensive. |
| **FFJORD** (Free-form Jacobian of Reversible Dynamics) | Uses **continuous-time normalizing flows** (Neural ODEs). | No need for discrete layers, can learn complex transformations. | Requires solving ODEs, which can be slow due to ODE solvers. |

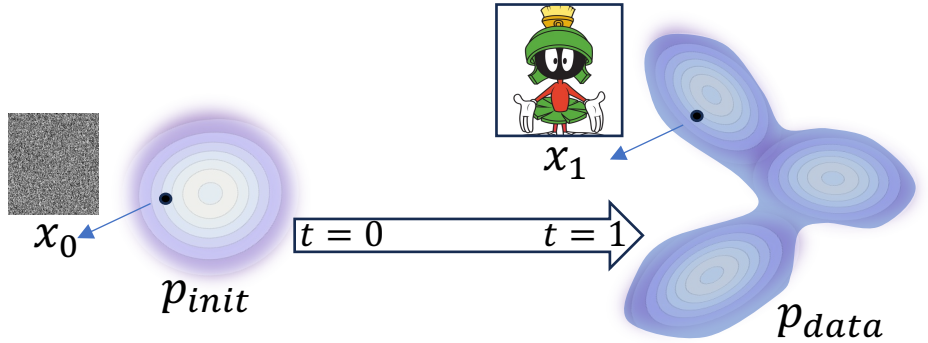

- RealNVP: "Density Estimation using Real NVP", Laurent Dinh, Jascha Sohl-Dickstein, Samy Bengio, ICLR 2017. Paper - https://arxiv.org/abs/1605.08803
- Glow: "Glow: Generative Flow with Invertible 1x1 Convolutions", Diederik P. Kingma, Prafulla Dhariwal, NeurIPS 2018. Paper - https://arxiv.org/abs/1807.03039
- Flow++: "Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design", Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, Pieter Abbeel, ICML 2019. Paper - https://arxiv.org/abs/1902.00275
- FFJORD: "FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models", Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, David Duvenaud, ICLR 2019. Paper - https://arxiv.org/abs/1810.01367

# ODE & SDE-based Flow and Diffusion Models:
# Flow and Score Matching


- Peter Holderrieth and Ezra Erives,"Generative AI With Stochastic Differential Equations" MIT, 6.S184/6.S975, IAP 2025, https://diffusion.csail.mit.edu/
- Michael S Albergo, Nicholas M Boffi, and Eric Vanden-Eijnden, "Stochastic interpolants: A unifying framework for flows and diffusions", arXiv preprint arXiv:2303.08797, 2023, https://arxiv.org/pdf/2303.08797
- Karras, T., Aittala, M., Aila, T., & Laine, S. ,"Elucidating the design space of diffusion-based generative models", Advances in neural information processing systems, 35, 26565-26577, 2022, https://arxiv.org/pdf/2206.00364
- Song, Y., Sohl-Dickstein, J., Kingma, D. P., Kumar, A., Ermon, S., & Poole, B. ," Score-based generative modeling through stochastic differential equations", *arXiv preprint arXiv:2011.13456*, 2021, https://arxiv.org/pdf/2011.13456
- Yaron Lipman et al, "Flow matching for generative modeling", arXiv preprint arXiv:2210.02747, 2022, https://arxiv.org/pdf/2210.02747

# ➢ ODE & SDE based Flow and Diffusion Models

- **From Noise into Data:**
  Construct an Ordinary Differential Equation (ODE) or a Stochastic Differential Equation (SDE) to transform samples from a simple probability distribution (e.g., Normal) into samples from a data probability distribution.




## ➢ Flow Model:

**ODE:** $dx_t/dt = u_t^\theta(x_t)$
**Initial state:** $x_0 \sim p_{init}$
**Goal:** $x_1 \sim p_{data}$
**Simulation:** $x_{t+h} = x_t + hu_t^\theta(x_t)$

- $u_t^\theta(x_t)$ : Neural network with parameter $\theta$ to model the vector field (e.g., velocity field)
- Simulation to generate $x_1 \sim p_{data}$ where $h$ is step size.

## ➢ Diffusion Model:

**SDE:** $dx_t = u_t^\theta(x_t)dt + \sigma_t dW_t$
**Initial state:** $x_0 \sim p_{init}$
**Goal:** $x_1 \sim p_{data}$
**Simulation:** $x_{t+h} = x_t + hu_t^\theta(x_t) + \sqrt{h}\sigma_t\epsilon_t$

- $u_t^\theta(x_t)$ : Neural network with parameter $\theta$ to model the vector field (e.g., velocity field)
- $\sigma_t \geq 0$: diffusion coefficient
- $W_t$ : Brownian motion stochastic process (Wiener process),
  Properties: starts from 0, continuous, Normal and independent increments ,
  Brownian motion simulation $W_{t+h} = W_t + \sqrt{h}\epsilon_t$
- Simulation to generate $x_1 \sim p_{data}$ where $\epsilon_t \sim \mathrm{N}(0, \mathrm{I_d})$ and $h$ is step size.

- If the vector field is Lipschitz continuous (continuously differentiable with bounded derivatives) and the diffusion coefficient is continuous, then a unique solution to the SDE/ODE($\sigma_t$=0) exists.

# ODE-based Flow Model

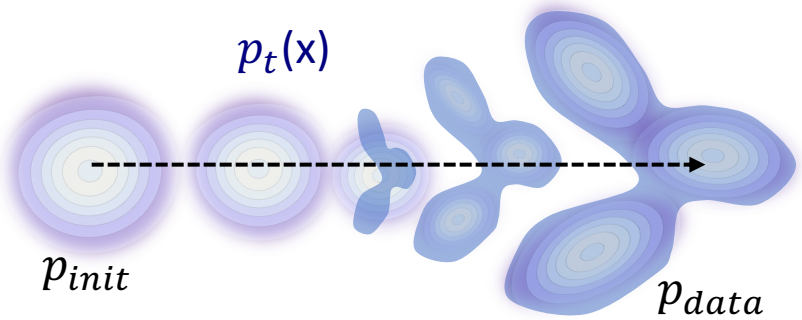
## Flow Matching

# ➢ ODE-based Flow Model

- **Flow Model: ODE:** $\frac{dx_t}{dt} = u_t^\theta(x_t)$, **Initial state:** $x_0 \sim p_{init}$ , **Goal:** $x_1 \sim p_{data}$

- **Continuity Equation:** $x_t \sim p_t$, $\frac{dp_t(x)}{dt} = -div(\, p_t(x)u_t(x)\,)$

  PDE that governs the time evolution of a probability distribution under the ODE dynamics, where div is divergence operator $div(f) = \sum_i \frac{\partial}{\partial x_i} f(x)$

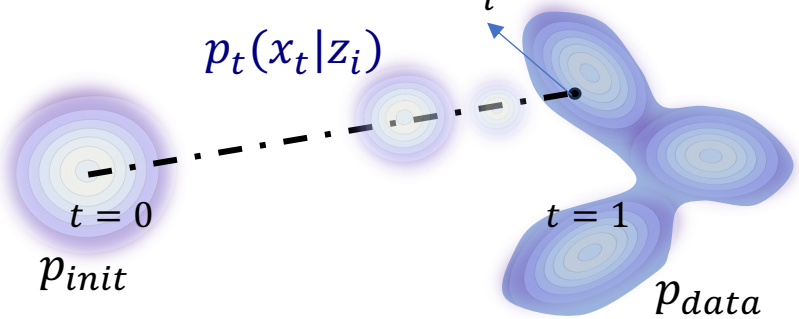



$$\frac{dx_t}{dt} = u_t(x_t), \qquad x_0 \sim p_{init}$$

$$\frac{dp_t(x)}{dt} = -div(\, p_t(x)u_t(x)\,)$$

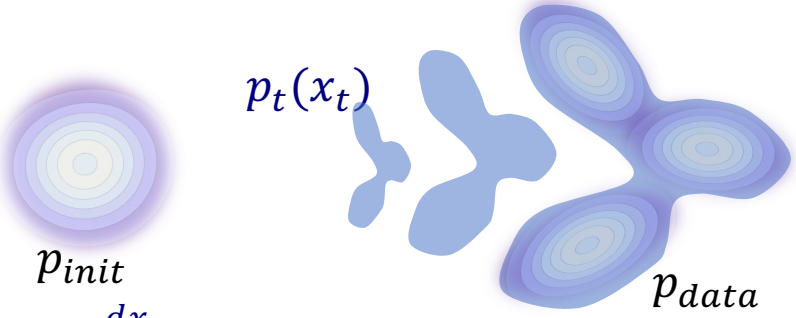
# ➢ ODE-based Flow Model - Training

To construct the neural vector field $u_t^\theta(x_t)$, we use available samples of the desired vector field, e.g., $u_t^*(x_t|z_i), i = 1, .., N$ where $z_1, \dots, z_N \sim p_{data}$ are the data samples.




- **Conditional Probability Path** $p_t(x_t|z_i)$: $p_{init} \longrightarrow z_i$
  Trajectory in the probability space from $p_{init}$ to Dirac delta distribution $\delta_{z_i}$
- **Conditional Vector field** $u_t^*(x_t|z_i)$:
  ODE under which conditional probability path is achieved.

$$\frac{dx_t}{dt} = u_t^*(x_t|z_i), \qquad x_0 \sim p_{init}, x_1 = z_i$$




- **Marginal Probability Path** $p_t(x_t)$: $p_{init} \longrightarrow p_{data}$
  Trajectory in the probability space from $p_{init}$ to $p_{data}$
- **Marginal Vector field** $u_t^*(x_t)$: ODE under which marginal probability path is achieved.

$$\frac{dx_t}{dt} = u_t^*(x_t), \qquad x_0 \sim p_{init}, x_1 \sim p_{data}$$

➢ Given the data samples $z_i$, we can often derive a conditional probability path and the corresponding conditional vector field **analytically**. Hence, they will be utilized during the training phase.

**Flow Matching:** **Loss Function** $= \mathbb{E}_{t\sim Unif[0,1],\ z\sim p_{data},\ x\sim p_t(x_t|z)} \left\| u_t^\theta(x) - u_t^*(x|z) \right\|^2$

Samples from different Conditional Probability Paths at different times

# ➢ Example: Gaussian-based Flow Model

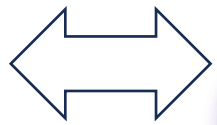
$p_1(x_1|z_i) = N(z_i, 0) = \delta_{z_i}$

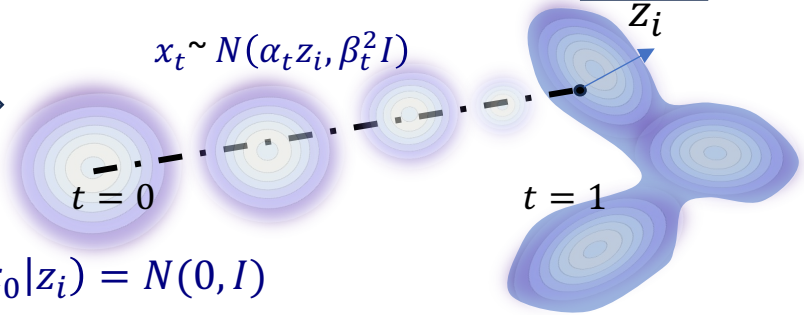
$z_i$

- **Gaussian Conditional Probability Path**
  - $p_t(x_t|z_i)$: $N(\alpha_t z_i, \beta_t^2 I)$,

Noise schedulers: continuously differentiable, monotonic functions

$$\begin{cases} \alpha_0 = 0, \alpha_1 = 1, \text{e.g.}, \alpha_t = t \\ \beta_0 = 1, \beta_1 = 0, \text{e.g.}, \beta_t = (1-t) \end{cases}$$






At each time t conditional probability path is a Gaussian distribution.

- **Path :** $x_t = \alpha_t z_i + \beta_t \epsilon$ where $\epsilon \sim N(0, I)$ , $z_i \sim p_{data}$

Random variable $x_t \sim N(\alpha_t z_i, \beta_t^2 I)$. This allows us to sample $x_t$ along the conditional probability path for training phase.

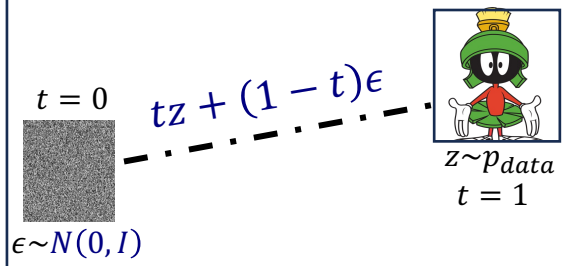
- **Gaussian Conditional Vector field:** $u_t^*(x_t|z_i) = \left(\dot{\alpha}_t - \frac{\dot{\beta}_t}{\beta_t}\alpha_t\right) z_i + \frac{\dot{\beta}_t}{\beta_t} x$

- **Loss Function:**

$$\mathbb{E}_{t\sim Unif[0,1],\ z\sim p_{data},\ x\sim N(\alpha_t z_i, \beta_t^2 I)} \left\| u_t^\theta(x) - \left(\dot{\alpha}_t - \frac{\dot{\beta}_t}{\beta_t}\alpha_t\right) z_i - \frac{\dot{\beta}_t}{\beta_t} x \right\|^2 = \mathbb{E}_{t\sim Unif[0,1],\ z\sim p_{data},\ \epsilon\sim N(0,I)} \left\| u_t^\theta(\alpha_t z + \beta_t \epsilon) - (\dot{\alpha}_t z + \dot{\beta}_t \epsilon) \right\|^2$$

➢ $\alpha_t = t, \beta_t = (1-t)$ , e.g., State-of-the-art models: Stable Diffusion 3, Meta's MovieGen

**Loss Function:** $\mathbb{E}_{t\sim Unif[0,1],\ z\sim p_{data},\ \epsilon\sim N(0,I)} \left\| u_t^\theta(\ tz + (1-t)\epsilon) \ - \ \underbrace{(z - \epsilon)}_{\text{Ground-truth velocity that transforms noise into data}} \right\|^2$




➢ **Training:**

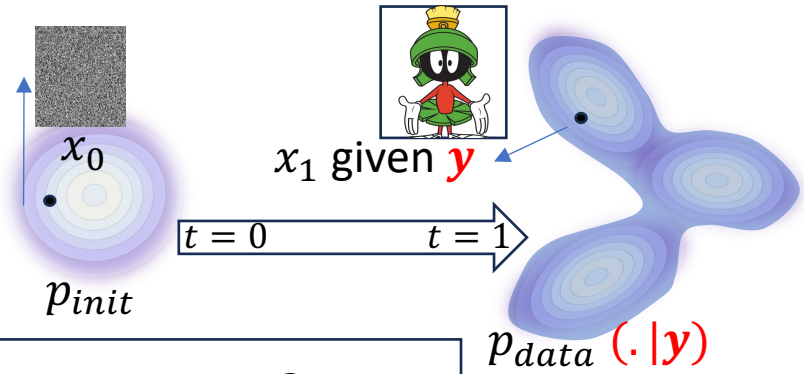
Input: A dataset of samples $z \sim p_{data}$, Output: neural network $u_t^\theta(x)$

1. **for** each mini-batch of data **do**
2. Sample a data example $z$ from the dataset
3. Sample a random time $t \sim Unif[0,1]$
4. Sample noise $\epsilon \sim N(0, I)$
5. Set $x = tz + (1-t)\epsilon$
6. Compute loss: L($\theta$)= $\mathbb{E}\left\| u_t^\theta(x) - (z - \epsilon) \right\|^2$
7. Update the model parameters $\theta$ via gradient descent on L($\theta$)
8. **end for**

# ➢ Conditional (Guided) Vector Field

- **Conditional Model:** $dx_t = \boldsymbol{u_t^\theta(x_t|y)}dt$, **Initial state:** $x_0 \sim p_{init}$, **Goal:** $x_1 \sim \boldsymbol{p_{data}(.|y)}$
  Guided vector field

where, $y$ is condition information such as prompt/text.




**Loss Function** $= \mathbb{E}_{t\sim Unif[0,1],\ (\boldsymbol{z},\boldsymbol{y})\sim \boldsymbol{p_{data}(z,y)},\ x\sim p_t(x_t|z)} \left\| u_t^\theta(x|y) - u_t^*(x|z) \right\|^2$

We sample from $\boldsymbol{(z,y)\sim p_{data}(z,y)}$, instead of $\boldsymbol{z\sim p_{data}(z)}$.

## ➢ Classifier-free Guidance: To improve the results, we will reinforce the effect of the conditioning variable y.

- Generates two predictions: one conditioned on the prompt and one unconditioned.
- Combines them to **amplify alignment with** conditioning variable y without needing a separate classifier.
- Linear Combination with the guidance scale.

- **Classifier-free Guided Vector field:** $\tilde{u}_t^*(\boldsymbol{x_t|y}) = (1-\omega)\, u_t^*(x_t|\emptyset) + \omega u_t^*(x_t|y)$ where $\omega$>1 is guidance scale
  Unguided vector field; Guided vector field

- We use same neural network $u_t^\theta(x|y)$ to approximate $u_t^*(x_t|\emptyset)$ and $u_t^*(x_t|y)$

  **Loss Function:** $\mathbb{E}_{t\sim Unif[0,1],(\boldsymbol{z},\boldsymbol{y})\sim \boldsymbol{p_{data}(z,y)},x\sim p_t(x_t|z)} \left\| u_t^\theta(x|y) - u_t^*(x|z) \right\|^2$ where $\boldsymbol{y}$ could be $\emptyset$ with non-zero probability.

- **Model:** $dx_t = \tilde{u}_t^\theta(\boldsymbol{x_t|y})dt$, **Initial state:** $x_0 \sim p_{init}$, $x_1 \sim \boldsymbol{p_{data}(.|y)}$

# SDE-based Diffusion Models

## Score Matching

# ➢ SDE-based Diffusion Model

- **Diffusion Model: SDE:** $dx_t = u_t^\theta(x_t)dt + \sigma_t dW_t$, **Initial state:** $x_0 \sim p_{init}$

- **Fokker-Planck Equation:** $x_t \sim p_t$, $\frac{\partial p_t(x)}{\partial t} = -div(\, p_t(x)u_t(x)\,) + \frac{\sigma_t^2}{2}\Delta p_t(x)$

PDE that governs the time evolution of probability distribution through the SDE, where div is the divergence operator $div(f) = \sum_i \frac{\partial}{\partial x_i} f(x)$ and Δ is Laplacian operator $\Delta f = \sum_i \frac{\partial^2}{\partial^2 x_i} f(x)$

- **Denoising Diffusion Model:** $dx_t = \left[u_t^\theta(x_t) + \frac{\sigma_t^2}{2}\nabla_x \log p_t(x)\right] dt + \sigma_t dW_t$, **Initial state:** $x_0 \sim p_{init}$ , **Goal:** $x_1 \sim p_{data}$
  - In SDE $dx_t = u_t^\theta(x_t)dt + \sigma_t dW_t$, noise impact increases by time.
  - To be able to decrease the noise impact and transform initial noise to data, we work with the Denoising Diffusion Model.
  - In the Denoising Diffusion Model, $\nabla_x$ is gradient and $s_t(x_t) = \nabla_x \log p_t(x) = \frac{\nabla_x p_t(x)}{p_t(x)}$ is the score function which pushes samples toward the real data. The score function gives a normalized gradient, pointing in the direction where the density increases the fastest.

Example: $p_t(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}$, $s_t(x_t) = \nabla_x \log p_t(x) = \nabla_x \left(-\frac{(x-\mu)^2}{2\sigma^2}\right) = -\frac{x-\mu}{\sigma^2}$

# ➢ SDE-based Diffusion Model - Training

$$dx_t = \left[u_t^\theta(x_t) + \frac{\sigma_t^2}{2} s_t^\theta(x_t)\right] dt + \sigma_t dW_t$$

We construct the neural vector field $u_t^\theta(x_t)$ and neural score function $s_t^\theta(x_t)$. We utilize available samples of the desired vector field, e.g., $u_t^*(x_t|z_i), i = 1, .., N$ and score function $s_t^*(x_t|z_i) = \nabla_x \log p_t(x|z_i)$ where $z_1, \dots, z_N \sim p_{data}$ are the data samples.

Conditional Probability Path

**Score Matching:** **Loss Function:** $\mathbb{E}_{t\sim Unif[0,1],\ z\sim p_{data},\ x\sim p_t(x_t|z)} \left\| s_t^\theta(x) - s_t^*(x|z) \right\|^2$

Samples from different Conditional Probability Paths at different times

- We can often express $u_t^\theta(x_t)$ and $s_t^\theta(x_t)$ in one single neural network with two outputs.

For Gaussian probability path, we don't need to train them separately as they can be converted into one another.

# ➢ Example: Gaussian-based Diffusion Model

- **Gaussian Conditional Probability Path** $p_t(x_t|z_i) = N(\alpha_t z_i, \beta_t^2 I)$
- **Path :** $x_t \sim N(\alpha_t z_i, \beta_t^2 I), x_t = \alpha_t z_i + \beta_t \epsilon$ where $\epsilon \sim N(0, I)$ , $z_i \sim p_{data}$
- **Gaussian Conditional Vector field:** $u_t^*(x_t|z_i) = \left(\dot{\alpha}_t - \frac{\dot{\beta}_t}{\beta_t}\alpha_t\right) z_i + \frac{\dot{\beta}_t}{\beta_t} x$
- **Score Function:** $s_t(x_t|z) = \nabla_x \log p_t(x) = \nabla_x \log N(\alpha_t z, \beta_t^2 I) = -\frac{x - \alpha_t z}{\beta^2}$

$$\boldsymbol{u_t^*(x_t|z_i) = \left(\beta_t^2 \frac{\dot{\alpha}_t}{\alpha_t} - \dot{\beta}_t \beta_t\right) s_t^*(x_t|z) + \frac{\dot{\alpha}_t}{\alpha_t} x}$$

- **Loss Function:** $\mathbb{E}_{t\sim Unif[0,1],\ z\sim p_{data},\ x\sim N(\alpha_t z_i, \beta_t^2 I)} \left\| s_t^\theta(x) + \frac{x - \alpha_t z}{\beta^2} \right\|^2 = \mathbb{E}_{t\sim Unif[0,1],\ z\sim p_{data},\ \epsilon\sim N(0,I)} \left\| s_t^\theta(\alpha_t z + \beta_t \epsilon) + \epsilon/\beta_t \right\|^2$

$$= \mathbb{E}_{t\sim Unif[0,1],\ z\sim p_{data},\ \epsilon\sim N(0,I)} \frac{1}{\beta_t^2} \left\| \beta_t s_t^\theta(\alpha_t z + \beta_t \epsilon) + \epsilon \right\|^2$$

Numerically unstable for $\beta_t$ close to 0 (due to division in loss)

- **<u>Noise Predictor</u>-based Loss Function:** Drop the constant $\frac{1}{\beta_t^2}$

$$\mathbb{E}_{t\sim Unif[0,1],\ z\sim p_{data},\ \epsilon\sim N(0,I)} \left\| \epsilon_t^\theta(\alpha_t z + \beta_t \epsilon) - \epsilon \right\|^2$$

where $\epsilon_t^\theta(x_t) = -\beta_t s_t^\theta(x_t)$.

- $\epsilon_t^\theta(x_t)$ learns to predict the noise $\epsilon$ that corrupted a data point $z$.

➢ **Training:**

Input: A dataset of samples $z \sim p_{data}$

Output: score neural network $s_t^\theta(x)$ or noise predictor $\epsilon_t^\theta(x)$

1. **for** each mini-batch of data **do**
2. Sample a data example $z$ from the dataset
3. Sample a random time $t \sim Unif[0,1]$
4. Sample noise $\epsilon \sim N(0, I)$
5. Set $x = tz + (1-t)\epsilon$
6. Compute loss: L($\theta$)= $\mathbb{E} \left\| s_t^\theta(x_t) + \frac{\epsilon}{\beta_t} \right\|^2$ **or** $\mathbb{E} \left\| \epsilon_t^\theta(x_t) - \epsilon \right\|^2$
7. Update the model parameters $\theta$ via gradient descent on L($\theta$)
8. **end for**

## ➢ Conditional (Guided) Score Function

- **Conditional Model:** $\mathrm{d}x_t = \left[u_t^\theta(x_t|y) + \frac{\sigma_t^2}{2} s_t^\theta(x|y)\right] dt + \sigma_t dW_t$, **Initial state:** $x_0 \sim p_{init}$ , **Goal:** $x_1 \sim \boldsymbol{p_{data}}(.|\boldsymbol{y})$

  Guided vector field  Guided score function

  where, $y$ is condition information such as prompt/text.

> **Loss Function** $= \mathbb{E}_{t\sim Unif[0,1],(\boldsymbol{z},\boldsymbol{y})\sim \boldsymbol{p_{data}}(\boldsymbol{z},\boldsymbol{y}),\ x\sim p_t(x_t|z)} \left\| s_t^\theta(x|y) - s_t^*(x|z) \right\|^2$
>
> We sample from $(\boldsymbol{z},\boldsymbol{y})\sim \boldsymbol{p_{data}}(\boldsymbol{z},\boldsymbol{y})$, instead of $\boldsymbol{z}\sim \boldsymbol{p_{data}}(\boldsymbol{z})$.

## ➢ Classifier-free Guidance: To improve the results, we will reinforce the effect of the conditioning variable y.

- **Classifier-free Guided Vector field:** $\tilde{\boldsymbol{s}}_{\boldsymbol{t}}^*(\boldsymbol{x_t}|\boldsymbol{y}) = (1-\omega)\, s_t^*(x_t|\emptyset) + \omega s_t^*(x_t|y)$ where $\omega$>1 is guidance scale

  Unguided
  Guided Score Function  Guided score field

- We use same neural network $s_t^\theta(x|y)$ to approximate $s_t^*(x_t|\emptyset)$ and $s_t^*(x_t|y)$

  **Loss Function:** $\mathbb{E}_{t\sim Unif[0,1],\ (\boldsymbol{z},\boldsymbol{y})\sim \boldsymbol{p_{data}}(\boldsymbol{z},\boldsymbol{y}),x\sim p_t(x_t|z)} \left\| s_t^\theta(x|y) - s_t^*(x|z) \right\|^2$, where $y$ could be ∅ with non-zero probability.

  - **Model:** $\mathrm{d}x_t = \left[\tilde{u}_t^\theta(x_t|y) + \frac{\sigma_t^2}{2} \tilde{s}_t^\theta(x|y)\right] dt + \sigma_t dW_t$, **Initial state:** $x_0 \sim p_{init}$ , **Goal:** $x_1 \sim \boldsymbol{p_{data}}(.|\boldsymbol{y})$

# Key Differential Equations

- **Flow Model: ODE:** $\frac{dx_t}{dt} = u_t^\theta(x_t)$, Initial state: $x_0 \sim p_{init}$ , $x_1 \sim p_{data}$

- **Continuity Equation:** $x_t \sim p_t$, $\frac{dp_t(x)}{dt} = -div(\, p_t(x)u_t(x)\,)$

PDE that governs the time evolution of probability distribution through the ODE, where div is divergence operator $div(f) = \sum_i \frac{\partial}{\partial x_i} f(x)$

- **Diffusion Model: SDE:** $dx_t = u_t^\theta(x_t)dt + \sigma_t dW_t$, Initial state: $x_0 \sim p_{init}$

- **Fokker-Planck Equation (Forward Kolmogorov Eq):** $x_t \sim p_t$, $\frac{\partial p_t(x)}{\partial t} = -div(\, p_t(x)u_t(x)\,) + \frac{\sigma_t^2}{2}\Delta p_t(x)$

PDE that governs the time-evolution of probability distribution through the SDE, where Δ is Laplacian operator $\Delta f = \sum_i \frac{\partial^2}{\partial^2 x_i} f(x)$

- **Denoising Diffusion Model:** $dx_t = \left[u_t^\theta(x_t) + \frac{\sigma_t^2}{2}\nabla_x \log p_t(x)\right]dt + \sigma_t dW_t$, Initial state: $x_0 \sim p_{init}$ , $x_1 \sim p_{data}$

- **Langevin dynamics:** special case of denoising diffusion model where $u_t^\theta(x_t) = 0$ $dx_t = \left[\frac{\sigma_t^2}{2}\nabla_x \log p(x)\right]dt + \sigma_t dW_t$

$p(x)$ is a stationary distribution of Langevin dynamics, $x_0 \sim p(x), x_t \sim p(x)\ t \geq 0$ (The probability distribution remains stationary over time $p_t(x) = p(x)$ )

If $x_0 \sim \tilde{p}_0(x) \neq p(x),\ x_t \sim \tilde{p}_t(x) \to p(x)$ : Distribution of $x_t$ converges to $p(x)$

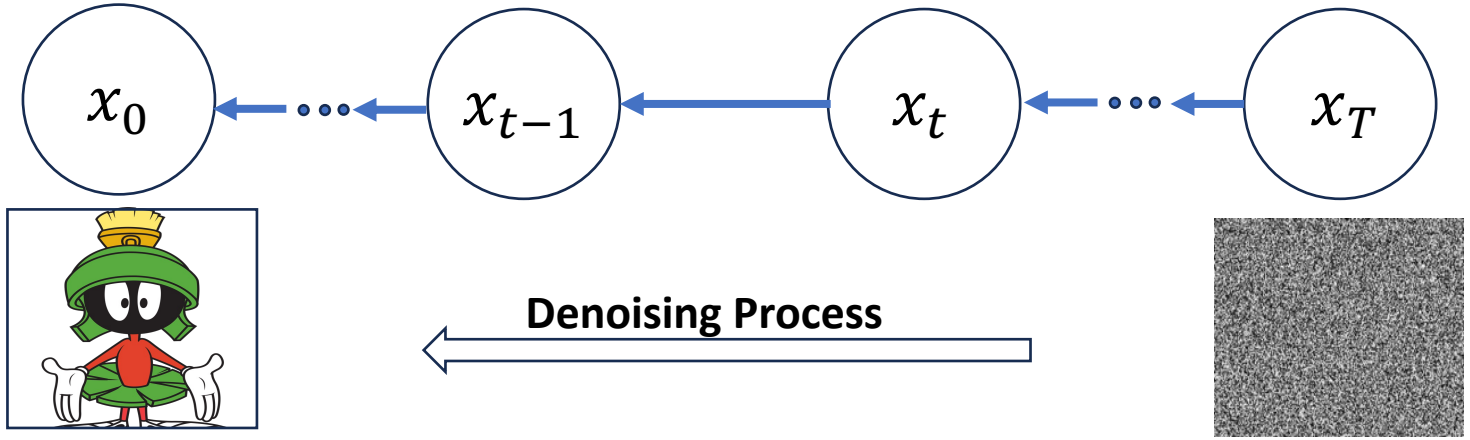
# Denoising Diffusion Models (DDM)

## Discrete-Time Formulation


- Ho, J., Jain, A., & Abbeel, P., “Denoising diffusion probabilistic models”, Advances in neural information processing systems”, 33, 6840-6851, 2020 https://arxiv.org/pdf/2006.11239
- Pieter Abbeel , CS294-158-SP24 Deep Unsupervised Learning Spring, UC Berkeley, 2024, https://sites.google.com/view/berkeley-cs294-158-sp24/home
- Foster, David. Generative deep learning. " O'Reilly Media, Inc.", 2022.
- Ho, J., Jain, A., & Abbeel, P., "Denoising Diffusion Probabilistic Models", NeurIPS 2020, https://arxiv.org/abs/2006.11239
- Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. , “High-Resolution Image Synthesis with Latent Diffusion Models”, CVPR, 2022, https://arxiv.org/abs/2112.10752
- Ho, J., Saharia, C., Chan, W., Fleet, D. J., Norouzi, M., & Salimans, T. , “Cascaded diffusion models for high fidelity image generation”, The Journal of Machine Learning Research 23.1, 2249-2281, 2022, https://arxiv.org/abs/2106.15282
- Saharia, Chitwan, et al. , “Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding”, ICML, 2022, https://arxiv.org/abs/2205.11487
- Peebles, W., & Xie, S. ,”Scalable Diffusion Models with Transformers", Proceedings of the IEEE/CVF international conference on computer vision, 2023 https://arxiv.org/abs/2212.09748

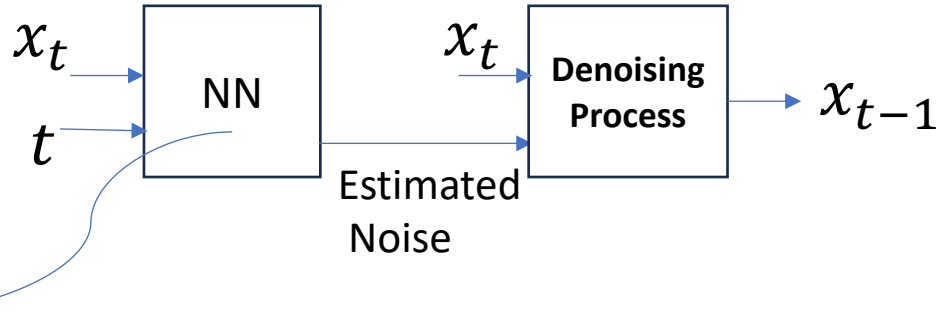
# Denoising Diffusion Models

- **Goal:** Transform noise into data




- Transform Noise $x_T$ into Data $x_0$:

For $t = T, \dots, 1$




- Train a neural network to estimate the noise

- **Forward (noising) Process:** We will train a neural net to learn the noise.

$$q(x_t|x_{t-1}) = N\left(\sqrt{1-\beta_t}x_{t-1}, \beta_t I\right), t = T, \dots, 1 \quad \Rightarrow \quad x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon_{t-1}\,,\ \epsilon_{t-1} \sim N(0, I)$$

  - Adds a small amount of Gaussian noise with variance $\beta_t$ to data $x_{t-1}$
  - We also scale input data $x_{t-1}$ to ensure that the variance of the output data $x_t$ remains constant over time, e.g., if we normalize the original data $x_0$ to have zero mean and unit variance, then $x_T$ will approximate a standard Gaussian distribution for large enough $T$, i.e., $Var(x_t) = (1-\beta_t)Var(x_{t-1}) + \beta_t Var(\epsilon_{t-1}) = 1$

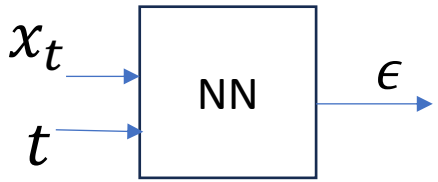
  - We can jump straight from data $x_0$ to noisy version of data $x_t$

$q(x_t|x_0) = N(\alpha_t x_0, \sigma_t^2 I)$ , diffusion schedules: $\alpha_t = \sqrt{\bar{\alpha}_t} = \sqrt{\Pi_{i=1}^{t}(1-\beta_t)}, \sigma_t = \sqrt{1-\bar{\alpha}_t} = \sqrt{1-\Pi_{i=1}^{t}(1-\beta_t)}$

$x_t = \alpha_t x_0 + \sigma_t \epsilon, \epsilon \sim N(0, I)$

For diffusion schedules, instead of linear schedules where $\beta_t$ increases linearly with $t$ (e.g., $\beta_1$=0.0001 to $\beta_T$=0.02), cosine schedules can also be used, e.g., $x_t = \cos(\frac{\pi}{2}\frac{t}{T})x_0 + sin(\frac{\pi}{2}\frac{t}{T})\epsilon$

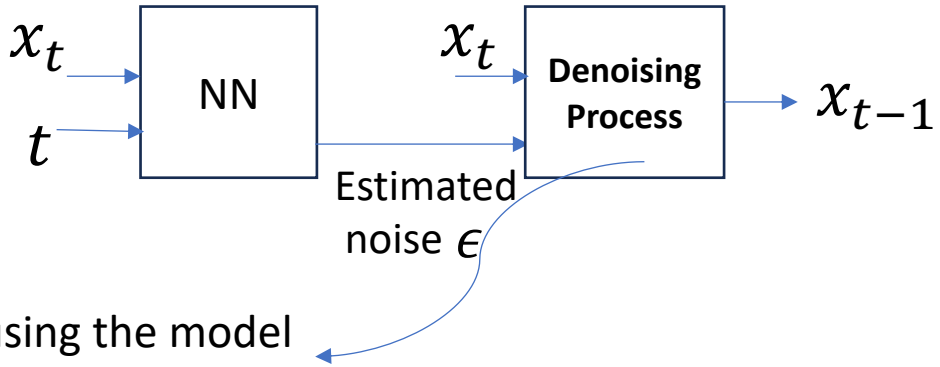
- **Training:** Given noisy version of data $x_t$ and time step $t$, neural network will estimate noise $\epsilon$ (the total amount of the noise that has been added to a given noisy data at time $t$, not just the noise that was added at the last time step of the noising process $\epsilon_{t-1}$)




**Training:**

0. Output: neural network $\epsilon_\theta$
1. **repeat**
2. Sample a data example $x_0$ from the dataset
3. Sample a random time $t \sim Unif(\{1, \dots, T\})$
4. Sample noise $\epsilon \sim N(0, I)$
5. Construct noisy data $x_t = \alpha_t x_0 + \sigma_t \epsilon$
6. Compute loss: L($\theta$)= $\|\epsilon - \epsilon_\theta(x_t, t)\|^2$
7. Update the model parameters $\theta$ via gradient descent on L($\theta$)
8. **until** converged

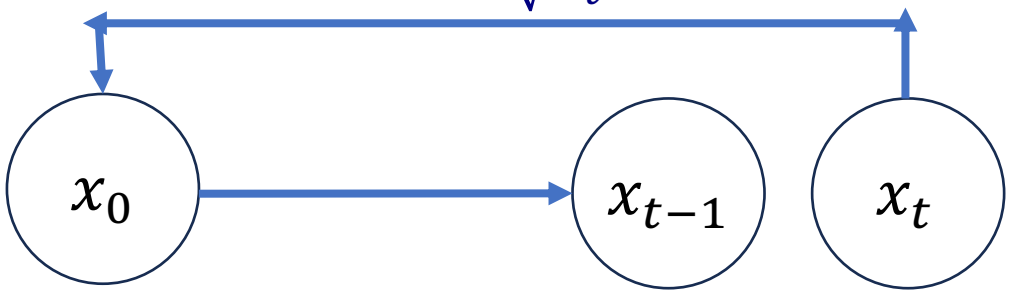
- **Denoising/Sampling Process and Sampling:** Given a trained neural network that estimates noise, we can transform noise into data




For $t = T, \dots, 1$

(1)Estimate the noise $\epsilon$ from $x_t$ using the model and use it to estimate $x_0$

$$x_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta(x_t, t)}{\sqrt{\bar{\alpha}_t}}$$




(2)Use the estimated noise $\epsilon$ to estimate $x_{t-1}$

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}x_0 + \sqrt{1-\bar{\alpha}_{t-1}-c_t^2}\epsilon_\theta(x_t, t) + c_t z_t$$

To make the sampling process random, we can add Gaussian noise $c_t z_t$ with the factor $c_t$.

**Sampling:**

0. Output: data $x_0$
1. Sample noise $x_T \sim N(0, I)$
2. **For** $t = T, \dots, 1$ **do**
3. Sample noise $z_t \sim N(0, I)$ if $t$>1, else $z_t = 0$
4. **Langevin Sampling**

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta(x_t,t)}{\sqrt{\bar{\alpha}_t}}\right) + \sqrt{1-\bar{\alpha}_{t-1}-c_t^2}\epsilon_\theta(x_t,t) + c_t z_t$$

5. **end for**
6. **return** $x_0$

- Similar to the SDE based diffusion model, we can show that learning noise $\epsilon_\theta$ approximates the score function $\nabla_x \log p_t(x)$.

- Similar to the SDE based diffusion model, we can leverage **Classifier-free Guidance** to improve the results.

  $\tilde{\epsilon}_\theta = (1-\omega)\epsilon_\theta(x_t) + \omega\epsilon_\theta(x_t, y)$ where $y$ is conditioning (guidance) information and $\omega$ is guidance strength

- Because of Gaussian noise assumption, we need many diffusion steps
  - The forward and backward process are Gaussian, only for small step sizes.
  - The data distribution is non-gaussian, thus, the noise required to transform the data into a normal distribution is inherently non-Gaussian

$$x_T = \alpha_t x_0 + \sigma_t \epsilon$$

noisy image ~ normal distribution

data ~ non-Gaussian

**noise ~ non-Gaussian**

  - Hence, to reduce the diffusion steps, we need to be able to work with non-Gaussian noises.

## ➢ Architectures for Diffusion Models:

| Model | Key Idea | Advantages | Challenges |
|---|---|---|---|
| **U-Net** | CNN-based architecture with an encoder-decoder structure for denoising. | Efficient feature extraction, widely used in diffusion models (DDPM, Stable Diffusion). | Limited ability to capture global dependencies. |
| **Latent Diffusion** | Performs diffusion in a compressed latent space using a VAE encoder-decoder. | Reduces computational cost, enables high-resolution image generation. | Requires a strong pre-trained VAE, potential loss of fine details. |
| **Hierarchical Generation** | Generates a low-resolution image first, then refines details progressively. | Improves quality and scalability, used in text-to-image models like Imagen & DALL·E 2. | More complex training pipeline, requires multiple models. |
| **Transformers in Diffusion** | Uses self-attention instead of CNNs for denoising and feature extraction. | Captures long-range dependencies, better for multi-modal tasks (text-to-image). | High memory usage, slower inference compared to CNN-based U-Nets. |


- Ho, J., Jain, A., & Abbeel, P., "Denoising Diffusion Probabilistic Models", NeurIPS 2020, https://arxiv.org/abs/2006.11239
- Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. , "High-Resolution Image Synthesis with Latent Diffusion Models", CVPR, 2022, https://arxiv.org/abs/2112.10752
- Ho, J., Saharia, C., Chan, W., Fleet, D. J., Norouzi, M., & Salimans, T. , "Cascaded diffusion models for high fidelity image generation", The Journal of Machine Learning Research 23.1, 2249-2281, 2022, https://arxiv.org/abs/2106.15282
- Saharia, Chitwan, et al. , "Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding", ICML, 2022, https://arxiv.org/abs/2205.11487
- Peebles, W., & Xie, S. ,"Scalable Diffusion Models with Transformers", Proceedings of the IEEE/CVF international conference on computer vision, 2023 https://arxiv.org/abs/2212.09748

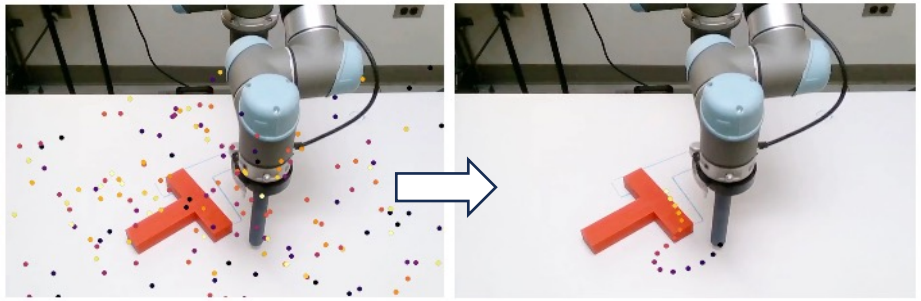
# ➢ Diffusion Models for Robotic Systems:

- In robotics models based on diffusion, the same principle applies, but instead of denoising an image, the model denoises a robot trajectory which is a sequence of states or actions over time.

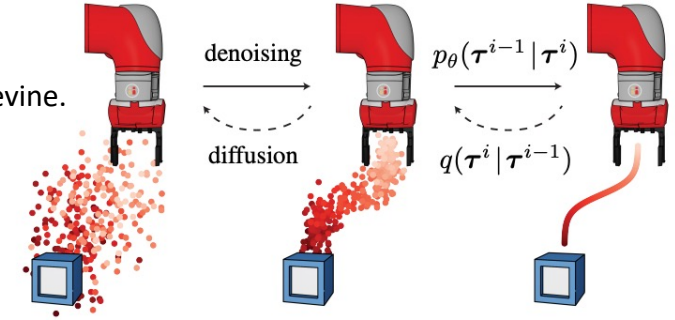
*Standard Diffusion Model*: Noisy Image → Clean Image

*Robotic Diffusion Model*: Noisy robot trajectory → Clean (**successful**) robot trajectory

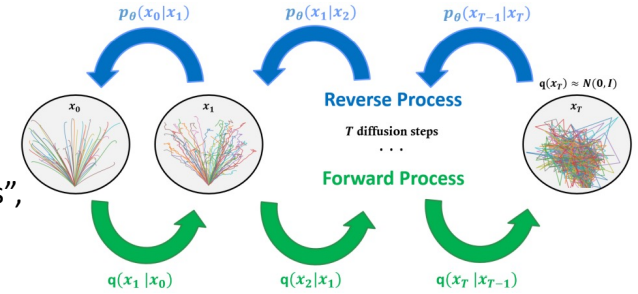
Example:
- At each timestep: where the robot arm is (joint positions), what it should do (move forward, close gripper), etc.
- A full sequence is like: $(x_0, u_0), (x_1, u_1), (x_2, u_2), \ldots$, where $x$ = robot state, $u$ = action
- Instead of pixels, each "pixel" is now a robot's pose, velocity, or action.



- Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, Shuran Song, "Diffusion Policy: Visuomotor Policy Learning via Action Diffusion", The International Journal of Robotics Research (IJRR), 2024, arXiv:2303.04137v5, https://arxiv.org/pdf/2303.04137v5 , https://diffusion-policy.cs.columbia.edu/




- Janner, Michael, Yilun Du, Joshua B. Tenenbaum, and Sergey Levine. "Planning with diffusion for flexible behavior synthesis.", 2022, arXiv:2205.09991, https://arxiv.org/pdf/2205.09991 .



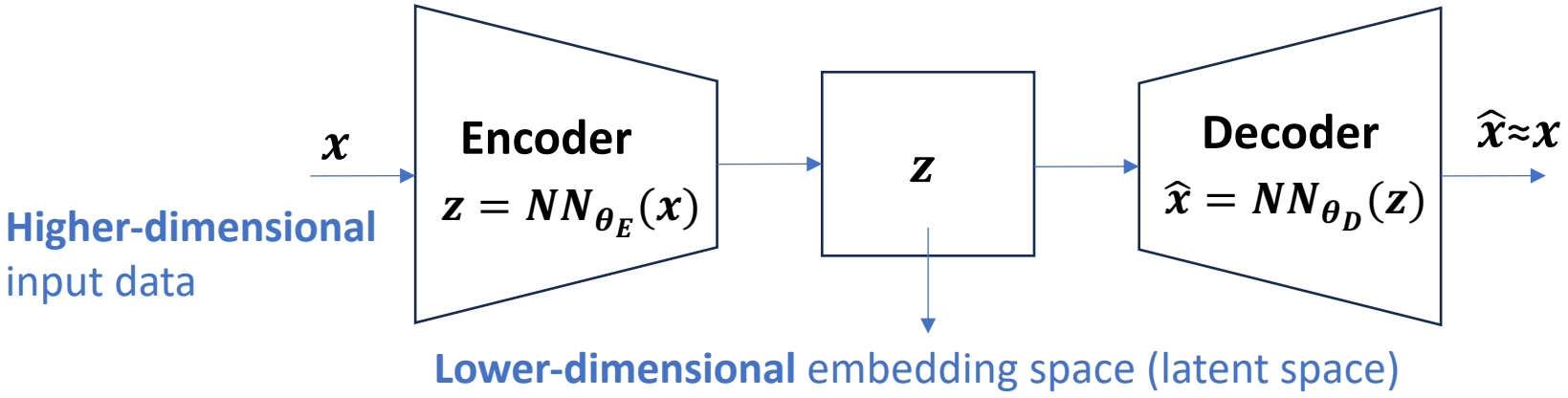

- Briden, J., Johnson, B. J., Linares, R., & Cauligi, A., "Diffusion Policies for Generative Modeling of Spacecraft Trajectories", In AIAA SCITECH 2025 Forum, 2025, https://arxiv.org/pdf/2501.00915

# Autoencoders and Variational Autoencoders (VAEs)


- Kingma, D. P., & Welling, M. ,”Auto-Encoding Variational Bayes”, In Proceedings of the 2nd International Conference on Learning Representations (ICLR), 2014, arXiv preprint arXiv:1312.6114, https://arxiv.org/pdf/1312.6114
- Pieter Abbeel , CS294-158-SP24 Deep Unsupervised Learning Spring, UC Berkeley, 2024, https://sites.google.com/view/berkeley-cs294-158-sp24/home
- Foster, David. Generative deep learning. " O'Reilly Media, Inc.", 2022.
- Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B., “High-resolution image synthesis with latent diffusion models”, In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2022. arXiv:2112.10752, https://arxiv.org/pdf/2112.10752

## ➢ Autoencoders:

a neural network that is trained to perform the task of encoding and decoding such that the output from the process is as close to the original input as possible.




Training: Find the parameters of the neural networks to minimize the difference between input and output data.

**Embedding Space (Latent Space):** Low dimensional vector space that models the data (captures simpler representation of data). Encoder is trained to map the input data to a point (lower-dimensional) in the embedding space. Decoder is trained to map the point in the embedding space to the data space. In sampling phase, we simply pick any point in the embedding space and transform it into the original data space using the decoder.

## ➢ Motivating Example:

- Input space: circle images (pixel space)
- Embedding space: 3D dimensional space $\boldsymbol{z} = [\boldsymbol{c_x}, \boldsymbol{c_y}, \boldsymbol{r}]$ where encodes the center coordinates and radius of the circles. Encoder and decoder is trained to learn the mapping (e.g., $f = (x - c_x)^2 + \left(y - c_y\right)^2 - r^2$ ) to transform the circle images into 3D space of $\boldsymbol{z}$ and back to the original pixel space.
- To generate a new circle image, sample any point in $\boldsymbol{z}$-space and decode it back to the image space using the decoder.

## ➢ Variational Autoencoders (VAEs):

- **Key Issues with Standard Autoencoder:**
  - **Discontinuous and Unstructured Latent Space:** In a standard autoencoder, the encoder maps each input $x$ to a single fixed point $z$ in the latent space. Since there's no constraint on the structure of $z$, different input data points can be scattered randomly in latent space, leading to gaps between encoded data points. When we attempt to generate new samples by interpolating in latent space, the decoder may receive unfamiliar latent representations, producing unrealistic outputs.
    Moreover, due to lack of structure, the decoder may not generalize well if latent space representations of similar inputs are far apart (closeness in the latent space should imply closeness in the data space).

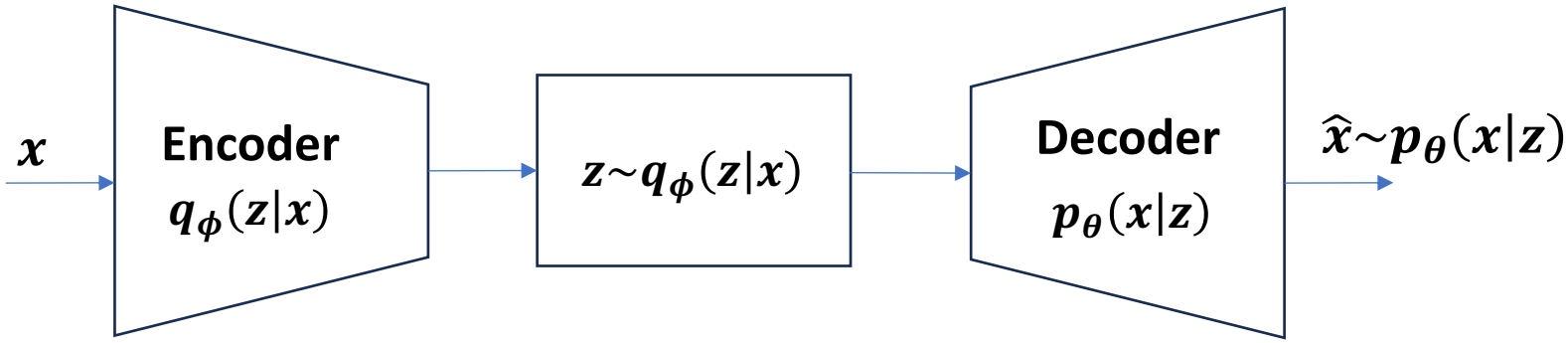
  - **No Stochastic Sampling:** Prevents uncertainty modeling and limits generative capability.

  - **Overfitting and Lack of Generalization:** Standard autoencoders can easily overfit to the training data, memorizing input features rather than learning meaningful latent representations.

- **VAE Solution:**
  - Instead of mapping each $x$ to a single $z$, VAE maps it to a distribution $\boldsymbol{q(z \mid x)}$ in the latent space. The latent space is continuous and smooth, allowing meaningful interpolation and sample generation.
  - Probabilistic Encoding and Decoding
  - Latent Space Regularization (KL Divergence): Prevents the model from memorizing data by constraining the latent distribution to match $N(0, I)$ improving generalization and encourages a structured latent space

- Since VAEs learn distributions rather than fixed mappings, they generalize better to unseen data and avoid overfitting.

## ➢ Variational Autoencoder (VAE):

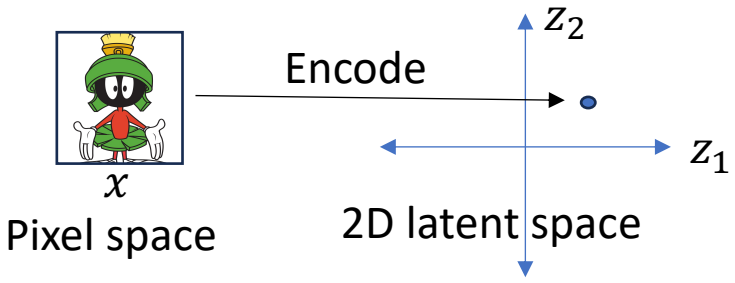



$$q_\phi(z|x) = N(\mu_\phi(x), \sigma^2_\phi(x))$$

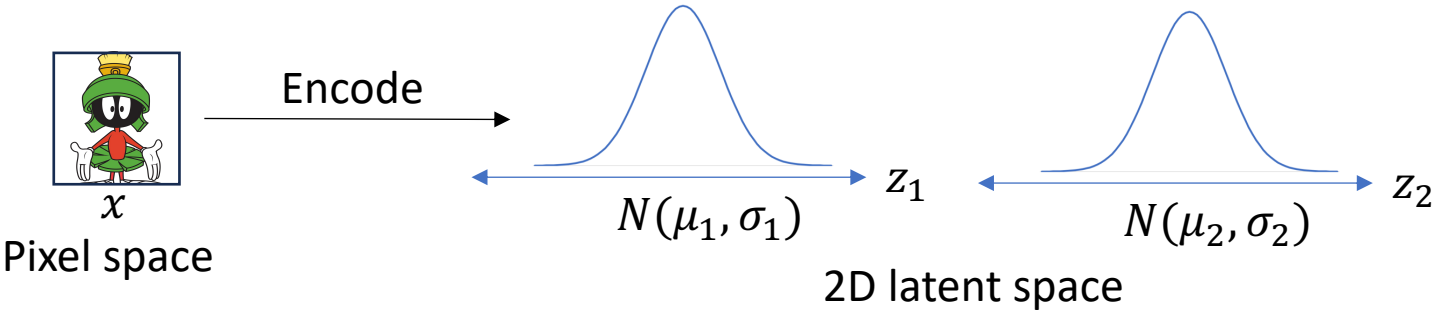
$$p_\theta(x|z) = N(\mu_\theta(z), \sigma^2_\theta(z))$$

- **VAE:** Instead of mapping each $x$ to a single $z$, VAE maps it to a distribution $q(z \mid x)$ in the latent space. The latent space is continuous and smooth, allowing meaningful interpolation and sample generation.

## ➢ Autoencoder:



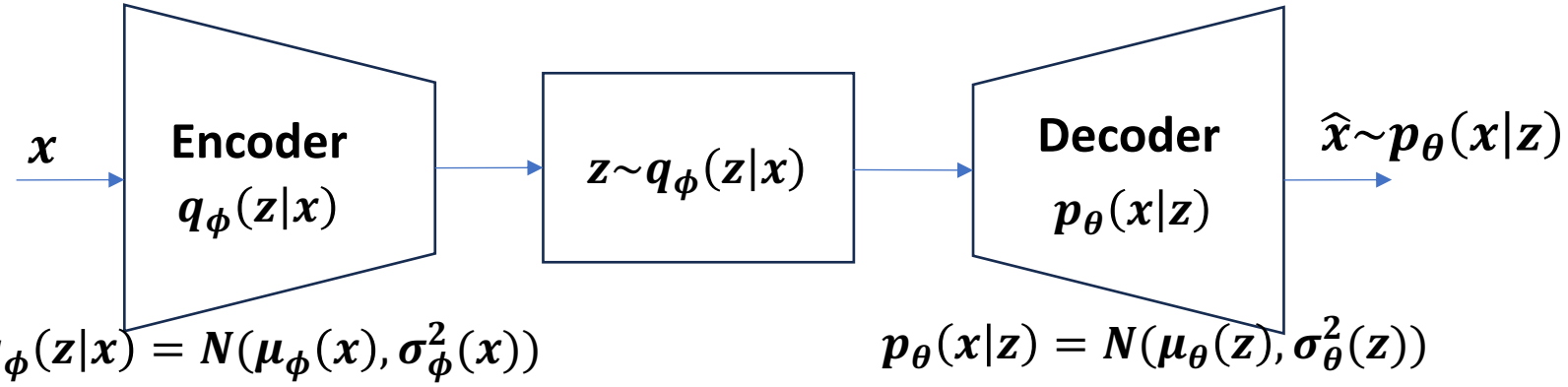

## ➢ Variational Autoencoder (VAE):




$$q_\phi(z|x) = N([\mu_1, \mu_2], [\sigma_1, \sigma_2]I_2)$$

In latent space, all dimensions are independent.
(restrictive — ignores correlations between data dimensions)

## ➢ Variational Autoencoder (VAE):




- **Loss Function:** $\mathrm{E}_{x\sim p_{data}}[logp(x)] \approx -\frac{1}{N}\sum_{i=1}^{N}\log p_\theta(x_i)$ maximum likelihood of data

$logp_\theta(x) = \log \int p_\theta(x|z)p(z)dz$ , computing this integral is intractable because it requires summing over all possible latent variables.

Since directly maximizing log $p$ $(x)$ is difficult, we instead maximize a lower bound on log $p$ $(x)$, called the **Evidence Lower Bound (ELBO)**:

$$\log p_\theta(x) \geq \mathrm{E}_{q(z|x)}\left[logp_\theta(x|z)\right] - KL\left(q_\theta(z|x)||p(z)\right)$$

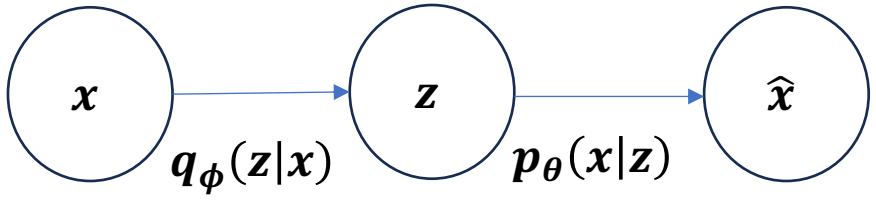
**Reconstruction loss:** Measures how well the decoder reconstructs $x$ from latent $z$

**Regularization:** Regularizes $q$ ( $z$ | $x$ ) to be close to $p$ ( $z$ ) (known distribution e.g., N(0,I)); Encourages a structured latent space.

**Loss Function :** $-\mathrm{E}_{q_\theta(z|x)}[logp_\theta(x|z)] + KL\left(q_\theta(z|x)||p(z)\right)$

**Reconstruction loss** **Regularization**

## ➢ Variational Autoencoder (VAE):




$$q_\phi(z|x) = N(\mu_\phi(x), \sigma^2_\phi(x))$$ Encoder: generates the mean and variance of each dimension of latent space

$$p_\theta(x|z) = N(\mu_\theta(z), \sigma^2_\theta(z))$$ Decoder: Decoder generates the mean and variance of each dimension of data space

**Loss Function :** $-\mathrm{E}_{q_\theta(z|x)}[logp_\theta(x|z)] + KL\ (q_\theta(z|x)||p(z))$

**Reconstruction loss** **Regularization** → N(0,1)

**Training:** For each data point $x$, the VAE architechture will generate $\mu_\phi(x), \sigma^2_\phi(x)$ (for latent space) and $\mu_\theta(z), \sigma^2_\theta(z)$ (for output space). For latent space, we want to push $\mu_\phi(x) \to 0, \sigma^2_\phi(x) \to \mathrm{I}$ (by minimizg the KL). For the output space, we want to push $\hat{x} \to x$

- **Training:**

**1. Sampling z:** $z \sim q_\phi(z|x) = N(\mu_\phi(x), \sigma^2_\phi(x))$

For each data point $x$, produce a latent distribution $q_\phi(z|x) = N(\mu_\phi(x), \sigma^2_\phi(x))$, then sample the correspond latent variable $z$,

$\mathrm{z} = \mu_\phi(x) + \sigma_\phi(x)\epsilon, \epsilon$~N(0,1)

**2. Evaluate Reconstruction Loss:** $\mathrm{E}_{q_\theta(z|x)}\ [logp_\theta(x|z)] = \sum_i logp_\theta(x|z_i) = \frac{1}{2}\sum_i (\frac{(x_i - \mu_i(z))^2}{\sigma_i^2(z)} + log\sigma_i^2(z))$

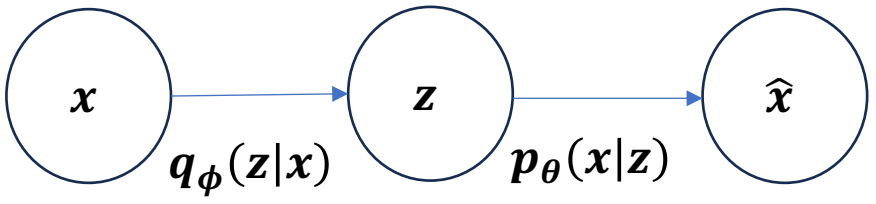
Samples of z

If we assume a **fixed variance** $\sigma^2_\theta(z)$, the **log term is constant** and can be ignored, simplifying the loss to: Mean Squared Error (MSE) **loss** $||x - \hat{x}||^2$

**3. Evaluate Regularization:** $KL\left(N\left(\mu_\phi(x), \sigma_\phi(x)\right)||N(0, I)\right) = -\frac{1}{2}\sum_i^d (1 + \log(\sigma_i^2(x)) - \mu_i^2(x) - \sigma_i^2(x))$

where the sum is taken over all dimensions in the latent space

## ➢ Variational Autoencoder (VAE):



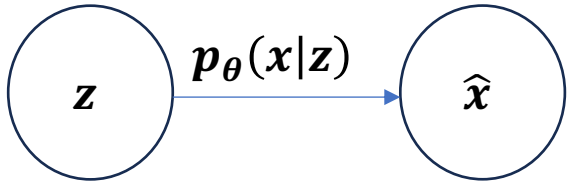

$$q_\phi(z|x) = N(\mu_\phi(x), \sigma^2_\phi(x))$$ Encoder: generates the mean and variance of each dimension of latent space

$$p_\theta(x|z) = N(\mu_\theta(z), \sigma^2_\theta(z))$$ Decoder: Decoder generates the mean and variance of each dimension of data space

**Loss Function :** $-\mathrm{E}_{q_\theta(z|x)}[log p_\theta(x|z)] + KL\,(q_\theta(z|x)||p(z))$

**Reconstruction loss** **Regularization** → N(0,1)

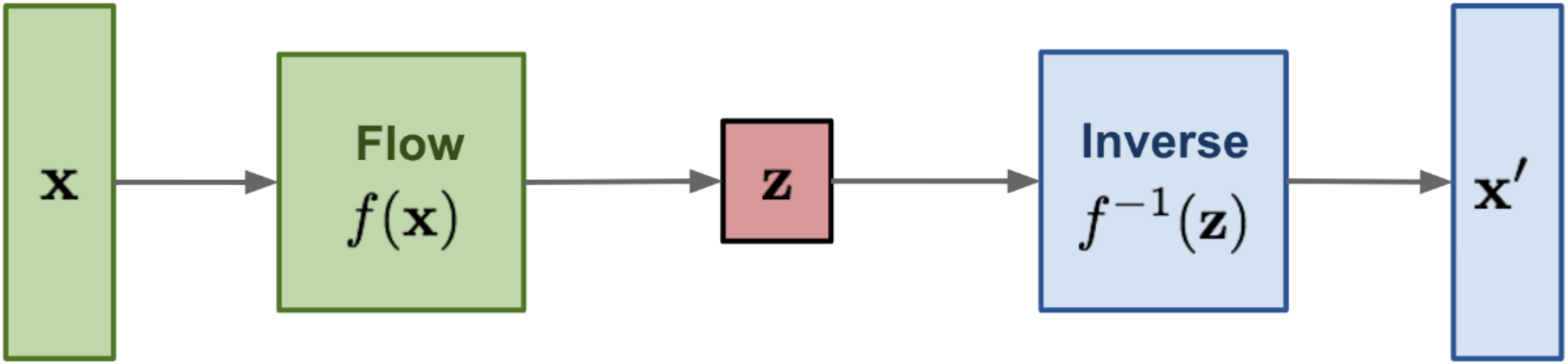
- **Sampling:**




$z \sim N(0, I)$ → Path through → $p_\theta(x|z)$ → Sample $p_\theta(x|z) = N(\mu_\theta(z), \sigma^2_\theta(z))$

**Connection to Flow Models:**

In Variational Autoencoder, we learn an encoder mapping function between a complex distribution of data and a much simpler distribution (e.g., Gaussian) that we can sample from(mapping from data to embedding space). We then learn a decoder mapping function from the simpler distribution to the complex distribution of data, so that we can generate a new data by sampling a point from the simpler distribution and applying the learned transformation. In flow models, the decoding function is designed to be the exact inverse of the encoding function.



**Example:** VAE Encoder and Decoder Architecture for Face Image Generation

**Encoder**

| Layer (type) | Output shape | Param # |
|---|---|---|
| InputLayer (input) | (32, 32, 3) | 0 |
| Conv Layer | (16, 16, 128) | 3,584 |
| BatchNormalization | (16, 16, 128) | 512 |
| LeakyReLU | (16, 16, 128) | 0 |
| Conv Layer | (8, 8, 128) | 147,584 |
| BatchNormalization | (8, 8, 128) | 512 |
| LeakyReLU | (8, 8, 128) | 0 |
| Conv Layer | (4, 4, 128) | 147,584 |
| BatchNormalization | (4, 4, 128) | 512 |
| LeakyReLU | (4, 4, 128) | 0 |
| Conv Layer | (2, 2, 128) | 147,584 |
| BatchNormalization | (2, 2, 128) | 512 |
| LeakyReLU | (2, 2, 128) | 0 |
| Flatten (flatten) | 512 | 0 |
| Dense (z_mean) | 200 | 102,600 |
| Dense (z_log_var) | 200 | 102,600 |
| Sampling (z) | 200 | 0 |

↓Generates a sample of z using latent space distribution

**Decoder**

| Layer (type) | Output shape | Param # |
|---|---|---|
| InputLayer | 200 | 0 |
| Dense | 512 | 102,912 |
| BatchNormalization | 512 | 2,048 |
| LeakyReLU | 512 | 0 |
| Reshape | (2, 2, 128) | 0 |
| TransposeConv Layer | (4, 4, 128) | 147,584 |
| BatchNormalization | (4, 4, 128) | 512 |
| LeakyReLU | (4, 4, 128) | 0 |
| TransposeConv Layer | (8, 8, 128) | 147,584 |
| BatchNormalization | (8, 8, 128) | 512 |
| LeakyReLU | (8, 8, 128) | 0 |
| TransposeConv Layer | (16, 16, 128) | 147,584 |
| BatchNormalization | (16, 16, 128) | 512 |
| LeakyReLU | (16, 16, 128) | 0 |
| TransposeConv Layer | (32, 32, 128) | 147,584 |
| BatchNormalization | (32, 32, 128) | 512 |
| LeakyReLU | (32, 32, 128) | 0 |
| TransposeConv Layer | (32, 32, 3) | 3,459 |

Latent space vector ←

Unflatten ←

1: See MLPs and CNNs for more information.

## ➢ Latent-Space based Methods

- To reduce memory usage during training, a common strategy is to operate in a latent space—a compressed, lower-resolution representation of the original data.
- The standard procedure begins by encoding the training data into the latent space using a pretrained autoencoder. Then, a generative model (e.g., a diffusion model) is trained directly within that latent space.
- For sampling, one first draws samples from the latent space using the trained generative model, and then reconstructs the final output using the decoder.
- A well-trained autoencoder filters out semantically irrelevant details, allowing the generative model to focus on the most perceptually meaningful features.

# Latent Space Diffusion Models (LDMs)

## Efficient Generation in Compressed Spaces

## ➢ Latent-Space Diffusion Model:

**Data Representation:**

• **Standard Diffusion Models** apply diffusion directly in the pixel space of images, meaning the model operates directly on raw, high-dimensional image data.

• **Latent Diffusion Models (LDMs)** operate in a compressed, lower-dimensional latent space, typically created by encoding raw data through an autoencoder. The latent representation preserves essential semantic content while significantly reducing dimensionality.

• **Stable Diffusion** is a well-known example of a latent diffusion model. It leverages a Variational Autoencoder (VAE) to encode high-dimensional images into a compact latent space, where the diffusion process is applied—enabling more efficient generation, manipulation, and reconstruction of high-quality outputs.

- The latent representation is lower-dimensional, semantically richer, and more efficient.
- Noise prediction in the latent space is typically simpler and more computationally efficient, allowing LDMs to generate high-quality outputs faster and more efficiently.

| Aspect | Standard Diffusion Model | Latent Diffusion Model (LDM) |
|---|---|---|
| Diffusion applied to | Pixel space | Latent (compressed) space |
| NN learns | Noise prediction in pixels | Noise prediction in latent space |
| Autoencoder usage | No | Yes, to move between latent and pixel space |

## ➢ Latent-Space Diffusion Model:

**1) Encoding Input Data to Latent Space**

• First, the high-dimensional data (e.g., images) is encoded into a lower-dimensional latent representation using an autoencoder:

- **Encoder**: $x \rightarrow z$
  - x: Input image data (pixel space), z: Latent representation (compressed, low-dimensional latent space). Typically, a **Variational Autoencoder (VAE)** is used here to encode and decode between x and z.

**2) Diffusion in Latent Space**

• Once encoded, the diffusion process (adding controlled noise step-by-step) is applied to the latent representation z instead of directly to pixel space x:

- Forward Diffusion Process (adding noise incrementally): $z_0 \rightarrow z_1 \rightarrow z_2 \rightarrow \cdots \rightarrow z_T$
- Each latent $z_t$ is a progressively noisier version of the original latent representation $z_0$.

**3) Training Neural Network (Noise Prediction)**

• Just like a standard diffusion model, a neural network (usually a **U-Net** architecture) is trained to reverse the diffusion process by predicting the noise added at each step:

- The network learns the conditional distribution: $p_\theta(z_{t-1} \mid z_t, t)$. Practically, it learns to predict the added Gaussian noise at time step t: $\epsilon_\theta(z_t, t)$
- The loss function is structurally identical to that of standard diffusion models: $L(\theta) = \mathrm{E}_{z_0,\epsilon\sim N(0,I),t}[\ ||\epsilon - \epsilon_\theta(z_t, t)||^2\ ]$
- Here: $\epsilon$ represents the added noise at time step t, $\epsilon_\theta$ is the neural network that predicts this noise. Minimizing this loss helps the model accurately predict and remove noise.
- In the **training phase**, the latent representation $z_0$ is obtained by encoding the **input image directly through the encoder**.

**4) Sampling from the Latent Diffusion Model**

• Once trained, sampling involves starting from pure Gaussian noise in the latent space and progressively denoising it step-by-step using the learned model:

- Reverse diffusion (denoising): $z_T \rightarrow z_{T-1} \rightarrow z_{T-2} \rightarrow \cdots \rightarrow z_0$
- The learned neural network iteratively denoises the latent variable representation until a meaningful latent representation $z_0$ is obtained.

**5) Decoding to Pixel Space**

• The final latent representation $z_0$ is decoded back into the original high-resolution image space using the decoder part of the autoencoder:

- **Decoder**: $z_0 \rightarrow x'$

➢ **Latent-Space Diffusion Model:**

| **Feature** | **Traditional VAE** | **Latent Diffusion (e.g., DALL·E 3)** |
|---|---|---|
| Latent space shape | Vector (e.g., [B, 128]) | Tensor (e.g., [B, 4, 64, 64]) |
| Structure | No spatial structure | Compressed 2D spatial grid |
| Encoder | Fully connected or simple CNN | Deep CNN |
| Decoder | Fully connected or simple CNN | Deep CNN |
| Used for | Basic generation or feature compression tasks | High-resolution, semantically rich image generation tasks |

# Autoregressive Models

## ➢ Autoregressive (AR) models

- Autoregressive (AR) models in Generative AI are a class of models that generate data sequentially, predicting each new data point based on previously generated ones.
- Autoregressive models generate outputs step by step, where each step conditions on prior steps. The neural network in an autoregressive model follows a **causal** dependency structure, meaning it predicts the next output only based on past inputs, never future ones.
    - Given a sequence of data points $x_1, x_2, \dots, x_n$ the probability of generating the sequence is factorized as:

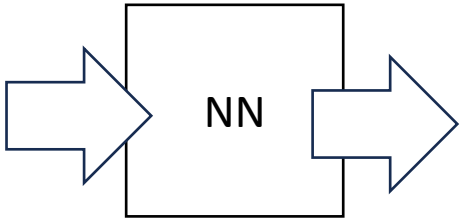
$$P(x_1, x_2, \dots, x_n) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2, x_3) \dots P(x_n|x_1, x_2, \dots, x_{n-1})$$

Instead of modeling the entire sequence at once, e.g., $P(x_1, x_2, \dots, x_n)$, an AR model predicts each element based on previous elements e.g., $P(x_i|x_1, x_2, \dots, x_{i-1})$.

- High-dimensional distributions are modeled as the product of one-dimensional conditional distributions

- An **autoregressive neural network** learns to predict the next token (word, pixel, audio sample) in a sequence based on previously generated tokens. The process involves **probabilistic modeling**, where the neural model learns a probability distribution over the next token at each step.

Sequence of token (e.g., word, pixels)




A probability distribution over all possible tokens to decide the next token.

- **Inference Process:** To generate output using the trained model
  1. Convert input tokens into **embeddings** (numerical representations).
  2. Feed embeddings into a neural network
  3. The network produces a probability distribution over the tokens.
  4. A token is sampled from this distribution.
  5. The chosen token is appended to the sequence and fed back as input.
  6. Steps 1-5 are repeated until a stopping criterion is met (e.g., maximum length or end-of-sequence token)

    - **Example:**
      - Input: “The cat sat on the“
      - Model predicts next word: {"mat": 0.8, "floor": 0.1, "table": 0.05, "cloud": 0.05}
      - Sampled: "mat“
      - New input: "The cat sat on the mat“
      - Model predicts next word: ...

- Each token is mapped to a high-dimensional vector using an **embedding layer**.

- **Tokenization and Embedding in Autoregressive Models**
  Before feeding data into an autoregressive neural network, raw data must be converted into numerical representations that the model can process. This happens in **two main steps**: **tokenization** and **embedding**.

- **Tokenization:** Tokenization is the process of splitting data (e.g., text) into smaller units called **tokens** (words, subwords, or characters). These tokens are then mapped to numerical indices.

  - **Example:**
    - **Input:** "The cat sat on the mat."
    - Tokens: ["The", "cat", "sat", "on", "the", "mat", "."]
    - Each token is mapped to an integer ID using a pre-defined vocabulary.
    - ["The" → 320, "cat" → 124, "sat" → 567, "on" → 87, "the" → 231, "mat" → 943, "." → 2]
    - **Output:** [320, 124, 567, 87, 231, 943, 2]

- **Embedding:** Converting Tokens into Continuous Vectors
  Once the tokens are converted into numerical IDs, they are **mapped into dense vectors** through an **embedding layer**. This step ensures that the model captures **semantic meaning** instead of treating words as arbitrary numbers.

An **embedding matrix** $E \in \mathrm{R}^{V \times d}$ is a learnable lookup table that maps each token ID to a high-dimensional vector, where d is the embedding size and V is the Vocabulary size (number of tokens in the model).

Example: If "cat" has token ID 124, we look up the 124th row in the embedding matrix to get a vector.

**Example:** For a vocabulary of size V = 50,000 and embedding size d = 768, we get:

"The" → [ 0.2, -0.1, 0.5, ..., 0.3]
"cat" → [ 0.8, -0.3, 0.1, ..., -0.2]
"sat" → [-0.1, 0.7, -0.5, ..., 0.6]

Each word gets a **768-dimensional vector**.

- **Example:** The embedding layer maps each token index to a vector:

**Output = EmbeddingMatrix[token indices]**

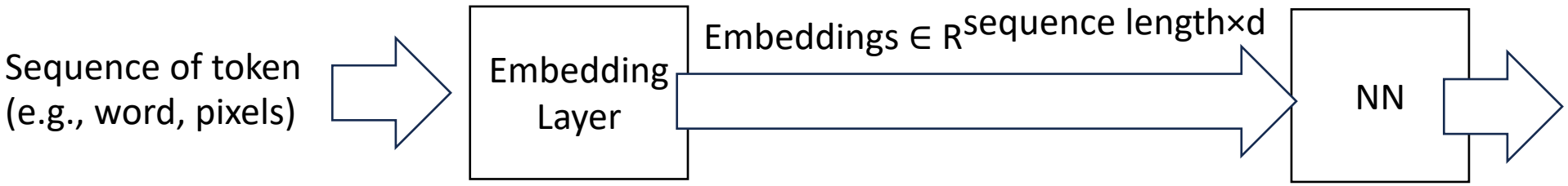
Sequence A: [123, 456, 789]
Sequence B: [123, 456, 789, 101, 202, 303]

- **Sequence A (3 tokens)** → Output shape: **(3, d)**
- **Sequence B (6 tokens)** → Output shape: **(6, d)**

Embedding Output $\in R^{\text{sequence length} \times d}$

- Words with similar meanings have embeddings close in space. This reflects learned semantic similarity in vector space

- The embedding matrix **E** is a set of trainable parameters in the model. It is updated using **gradient descent** just like any other model parameters ( e.g., layer weights).



➢ **Training:**

**Loss Function**: Cross-entropy loss between the predicted probability and actual token.

$$L = -\sum_{i}^{V} y_i log p_i$$

where, $y_i$ is 1 for the true next token, 0 for all others (one-hot encoding) and $p_i$ is the predicted probability for token $i$.

- The target distribution (ground truth) is a **one-hot vector**, where all values are **0** except for a **1** at the position corresponding to the correct next token.
- The neural network's output is a **predicted probability distribution** over the vocabulary, and during training, it learns to make this predicted distribution as close as possible to the one-hot ground truth distribution.

Example:
Vocabulary: ["cat", "dog", "fish", "elephant"]
Ground Truth (One-hot Vector) : [0, 1, 0, 0] (next token: dog)
Model's Predicted Probabilities: [0.1, 0.7, 0.15, 0.05]

**Loss Calculation: L=−log(0.7)=0.356**

## ➢ Autoregressive (AR) Models:

| Model | Type | Relation to Autoregression | Use Cases |
|---|---|---|---|
| **Bayes' Net** | Probabilistic Graphical Model | Can be autoregressive if structured sequentially | Probabilistic inference, generative modeling |
| **MADE** | Neural Network (Masked Autoencoder) | Enforces autoregression via masks | Density estimation, generative modeling |
| **Causal Masked NN** | Transformer-based Neural Network | Uses masks to prevent future information from leaking | Language models (**GPT**), Image generation (**PixelCNN**) |
| **RRN** | Recurrent Neural Network | Processes sequences autoregressively with residual connections | Speech synthesis, sequential data modeling |

# Generative Adversarial Networks (GANs)


- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. ,"Generative adversarial nets", In Advances in Neural Information Processing Systems (NeurIPS), 2014, arXiv:1406.2661, https://arxiv.org/pdf/1406.2661
- Pieter Abbeel , CS294-158-SP24 Deep Unsupervised Learning Spring, UC Berkeley, 2024, https://sites.google.com/view/berkeley-cs294-158-sp24/home
- Foster, David. Generative deep learning. " O'Reilly Media, Inc.", 2022.

## ➢ Generative Adversarial Networks (GANs)

- Like flow models and VAEs, GANs aim to generate samples by transforming noise into data.
- Implicit Generative Model: Does not require explicit likelihood modeling or density estimation.

- GANs consist of two neural networks—**a generator** and **a discriminator**—that compete in a min-max game framework.
- The generator receives a **random noise vector (latent space representation)**, often sampled from a Gaussian or uniform distribution. The generator outputs **synthetic data** (e.g., images, text, audio) that mimic real data (similar to decoder of a VAE).
- The discriminator receives two types of inputs: **Real data samples** from the actual dataset and **Fake data samples** generated by the generator. The discriminator outputs a **probability score (0 to 1)** indicating whether the input is real or fake.


**Discriminator:** It is trained to distinguish the real and fake images.
**Generator:** The generator is trained to produce images that the discriminator classifies as real.

**Training Process:**

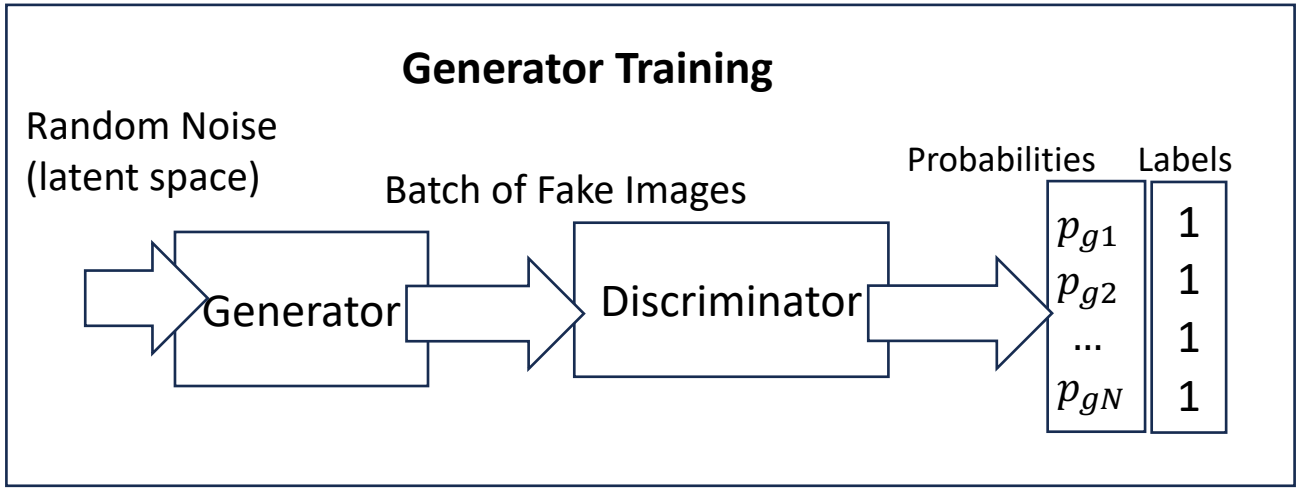
**1) Discriminator:**

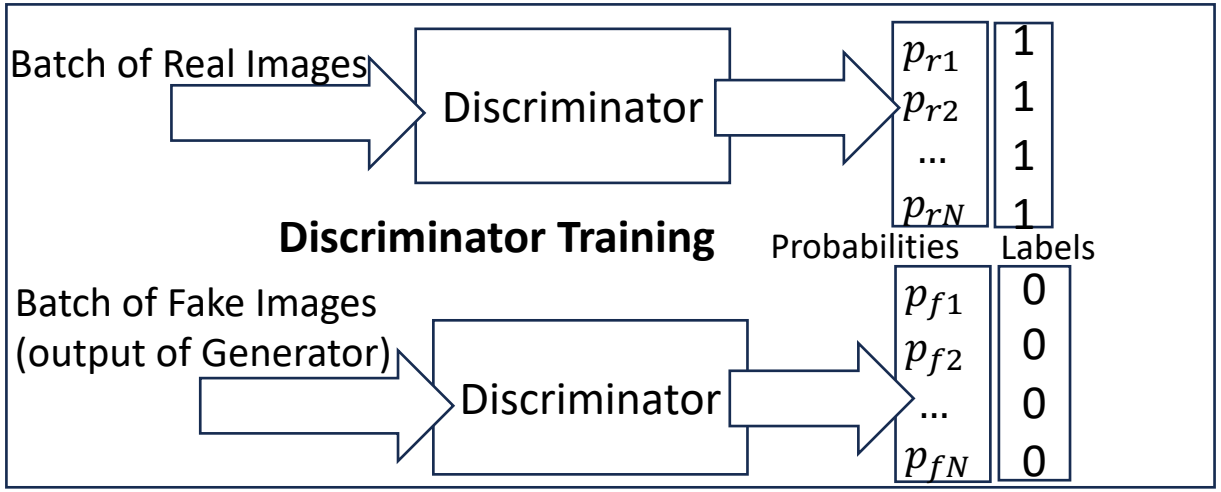
- Discriminator should output the probability of the input image being real; Hence, for the real images it should output probability of 1 and for the fake images it should output probability of 0.
- We can treat this as a supervised learning problem, where the labels are 1 for the real images and 0 for the fake images, with binary cross-entropy as the loss function.

**2) Generator:**

- We want to train the generator to produce images that the discriminator thinks are real. Hence, to train the generator, its output images are passed to the discriminator to compute scores
- The loss function for the generator is the binary cross-entropy between the discriminator's predicted probabilities and the vector of ones.

**3) Repeat:** The process iterates, improving both networks over time. We must alternate the training of these two networks and only update the weights of the one network at a time. The goal is to generate images to be predicted close to one because the generator is strong not because the discriminator is weak.



➢ **Loss Functions:**

- $D(x)$ is the discriminator's probability estimate that $x$ is real.
- $G(z)$ is the generator's output given a random noise $z$.
- $D(G(z))$ is the discriminator's probability estimate that $G(z)$ is real.
- $P_{data}(x)$ represents the real data distribution.
- $P_z(z)$ represents the latent space distribution (random noise input to generator)

- **Discriminator** should be trained to output $D(x) = 1$ and $D\big(G(z)\big) = 0$

$$\max_{\mathrm{D}} \mathrm{E}_{x\sim P_{data}(x)}[\log D(x)] \approx \max_{\mathrm{D}} \frac{1}{N} \sum_{i}^{N} \log D(x_i) \quad \& \quad \max_{D} \mathrm{E}_{z\sim P_z(z)} \left[\log\left(1 - D\big(G(z)\big)\right)\right] \approx \max_{\mathrm{D}} \frac{1}{N} \sum_{i}^{N} \log\left(1 - D\big(G(z_i)\big)\right)$$

Encourages the discriminator to assign higher probability (closer to 1) to real samples.

Encourages the discriminator to assign lower probability (closer to 0) to fake samples.

- **Generator** should be trained to output $D\big(G(z)\big) = 0$

$$\min_{G} -\mathrm{E}_{z\sim P_z(z)}[\log D(G(z))] = \min_{G} \mathrm{E}_{z\sim P_z(z)} \left[\log\left(1 - D\big(G(z)\big)\right)\right]$$

This encourages the generator to maximize D(G(z)), making fake samples appear real to the discriminator (Hence, discriminator outputs a probability close to 1 for fake samples).

➢ **Min-Max Game:** $\min_{\mathrm{G}} \max_{\mathrm{D}} \mathrm{E}_{x\sim P_{data}(x)}[\log D(x)] + \mathrm{E}_{z\sim P_z(z)}[\log\left(1 - D\big(G(z)\big)\right)]$

At equilibrium, the **generator** produces samples that are **indistinguishable** from real data, and the **discriminator** assigns a probability of **0.5** to both real and fake data.

**Algorithm 1** Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, $k$, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

**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**

- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. ,"Generative adversarial nets", In Advances in Neural Information Processing Systems (NeurIPS), 2014, arXiv:1406.2661, https://arxiv.org/pdf/1406.2661

**Example:**

- Batch size = 5
- Discriminator outputs:
  Real images: $D\ (\ x\ )$ = [ 0.9 , 0.8 , 0.95 , 0.7 , 0.85 ]
  Fake images: $D\ (\ G\ (\ z\ )\ )$ = [ 0.1 , 0.2 , 0.05 , 0.3 , 0.15 ]

  The discriminator loss is composed of two parts:

  **Loss for Real Images**: $-\frac{1}{N}\sum_{i=1}^{N}\log D(x_i) = -1/5(\log 0.9+\log 0.8+\log 0.95+\log 0.7+\log 0.85)$

  **Loss for Fake Images**: $-\frac{1}{N}\sum_{i=1}^{N}\log(1-D(G(z_i))) = -1/5(\log(1-0.1)+\log(1-0.2)+\log(1-0.05)+\log(1-0.3)+\log(1-0.15))$

- Generator outputs (trying to fool the discriminator): $D\ (\ G\ (\ z\ )\ )$ = [ 0.4 , 0.5 , 0.45 , 0.55 , 0.6 ]

  - **Compute Generator Loss:** $-\frac{1}{N}\sum_{i=1}^{N}\log(D(G(z_i))) = -1/5(\log 0.4+\log 0.5+\log 0.45+\log 0.55+\log 0.6)$

- **Sampling Process:** The generator takes a random latent vector (noise) as input. It outputs high-fidelity synthetic data. The trained discriminator is usually discarded during inference.

- **Training Challenges:**

- **Discriminator Overpowers the Generator**

When the discriminator becomes too strong, loss function of generator becomes too weak to improve the generator (the generator receives vanishing gradients and cannot improve effectively).

- **Generator Overpowers the Discriminator**
  Mode collapse: The generator starts mapping many latent inputs to a limited set of outputs, that fools the discriminator, reducing diversity.

**Example:** GAN Discriminator and Generator Architectures for Image Generation.

**Discriminator**

| Layer (type) | Output shape | Param # |
|---|---|---|
| InputLayer | (64, 64, 1) | 0 |
| Conv Layer | (32, 32, 64) | 1,024 |
| LeakyReLU | (32, 32, 64) | 0 |
| Dropout | (32, 32, 64 ) | 0 |
| Conv Layer | (16, 16, 128) | 131,072 |
| BatchNormalization | (16, 16, 128) | 512 |
| LeakyReLU | (16, 16, 128) | 0 |
| Dropout | (16, 16, 128) | 0 |
| Conv Layer | (8, 8, 256) | 524,288 |
| BatchNormalization | (8, 8, 256) | 1,024 |
| LeakyReLU | (8, 8, 256) | 0 |
| Dropout | (8, 8, 256) | 0 |
| Conv Layer | (4, 4, 512) | 2,097,152 |
| BatchNormalization | (4, 4, 512) | 2,048 |
| LeakyReLU | (4, 4, 512) | 0 |
| Dropout | (4, 4, 512) | 0 |
| Conv Layer | (1, 1, 1) | 8,192 |
| Flatten | 1 | 0 |

**Generator**

| Layer (type) | Output shape | Param # |
|---|---|---|
| InputLayer | 100 | 0 |
| Reshape | (1, 1, 100) | 0 |
| Transpose Conv Layer | (4, 4, 512) | 819,200 |
| BatchNormalization | (4, 4, 512) | 2,048 |
| ReLU | (4, 4, 512) | 0 |
| Transpose Conv Layer | (8, 8, 256) | 2,097,152 |
| BatchNormalization | (8, 8, 256) | 1,024 |
| ReLU | (8, 8, 256) | 0 |
| Transpose Conv Layer | (16, 16, 128) | 524,288 |
| BatchNormalization | (16, 16, 128) | 512 |
| ReLU | (16, 16, 128) | 0 |
| Transpose Conv Layer | (32, 32, 64) | 131,072 |
| BatchNormalization | (32, 32, 64) | 256 |
| ReLU | (32, 32, 64) | 0 |
| Transpose Conv Layer | (64, 64, 1) | 1,024 |

1: See MLPs and CNNs for more information.

## • GAN Variants

| Model | Improvement | Architecture | Use Case |
|---|---|---|---|
| **Vanilla GAN** | Basic GAN architecture | Simple MLP-based Generator & Discriminator | General image generation |
| **DCGAN** | Uses CNNs for better stability | Convolutional layers, BatchNorm, LeakyReLU | Image synthesis |
| **cGAN** | Conditional inputs | Adds class labels to both Generator & Discriminator | Controlled generation (e.g., digit-to-image) |
| **InfoGAN** | Learns disentangled features | Introduces mutual information loss for latent variables | Structured data generation |
| **WGAN** | Better loss function for stability | Uses Wasserstein distance with weight clipping | Stable training and high-quality generation |
| **WGAN-GP** | Improves WGAN training | Uses gradient penalty instead of weight clipping | Large-scale training, better convergence |
| **LSGAN** | Uses least squares loss | Replaces cross-entropy loss with L2 loss | Super-resolution, stable training |
| **EBGAN** | Energy-based discriminator | Uses an autoencoder-based Discriminator | Image denoising, generative tasks |
| **BEGAN** | Balances training of G & D | Uses equilibrium constraint for autoencoder-based Discriminator | Artistic image generation |
| **BigGAN** | High-resolution images | Scaled-up GAN with larger batch sizes and self-attention | Photorealistic image generation |
| **StyleGAN** | Fine-grained style control | Introduces Adaptive Instance Normalization (AdaIN) | DeepFake, high-quality face synthesis |
| **SAGAN** | Uses self-attention | Self-attention mechanism for long-range dependencies | High-quality image synthesis |
| **CycleGAN** | Unpaired image-to-image translation | Two GANs with cycle-consistency loss | Image transformation (e.g., horse ↔ zebra) |
| **Pix2Pix** | Paired image-to-image translation | U-Net Generator with PatchGAN Discriminator | Sketch-to-image, image enhancement |
| **StarGAN** | Multi-domain translation | Single model trained with domain labels | Face attribute editing (e.g., age, hair color) |
| **SRGAN** | High-resolution image synthesis | Uses perceptual loss and deep CNNs | Super-resolution image enhancement |
| **PGGAN** | Progressive growing for high-resolution images | Starts with low-res images and increases resolution gradually | High-resolution face synthesis |
| **DualGAN** | Unpaired image translation | Two generators and two discriminators like CycleGAN | Domain adaptation, cross-domain mapping |
| **Few-Shot GAN** | Learns from few samples | Pretrained GANs adapted for few-shot learning | Personalized image generation |
| **MoCoGAN** | Video synthesis with motion-content separation | Uses separate latent variables for motion and content | Video generation (e.g., animations) |
| **DVD-GAN** | Autoregressive video synthesis | Uses autoregressive latent space modeling with deep CNNs | High-quality video generation |

# Topics:

## ➢ Generative AI - Algorithms

- Flow Models
- Ordinary Differential Equation (ODE)-based Flow Models
- Denoising Diffusion Models (DDMs)
- Stochastic Differential Equation(SDE)-based Denoising Diffusion Models
- Autoencoders and Variational Autoencoders (VAEs)
- Latent Space Diffusion Models
- Autoregressive Models
- Generative Adversarial Networks (GANs)

## ➢ Generative AI - Architectures

- Multilayer Perceptrons (MLPs)
- Training and Loss Functions Types
- Backpropagation Algorithm, Stochastic Gradient Descent (SGD), and Adam Optimizer
- Common Training Issues, Regularization in Deep Learning, and Scaling Laws for Deep Learning
- Convolutional Neural Networks (CNNs) and PixelCNN
- U-Net Denoising Model
- Recurrent Neural Networks (RNNs), LSTM (Long Short-Term Memory), and GRU (Gated Recurrent Unit)
- Transformers: Self-Attention, Multi-Head Attention, and Cross-Attention
- Diffusion Transformers (DiTs), Vision Transformers (ViTs), and Attention-Based U-Nets
- Multimodal Models
- Foundation Models

# Generative AI - Architectures

# Topics:

## ➢ Generative AI - Architectures

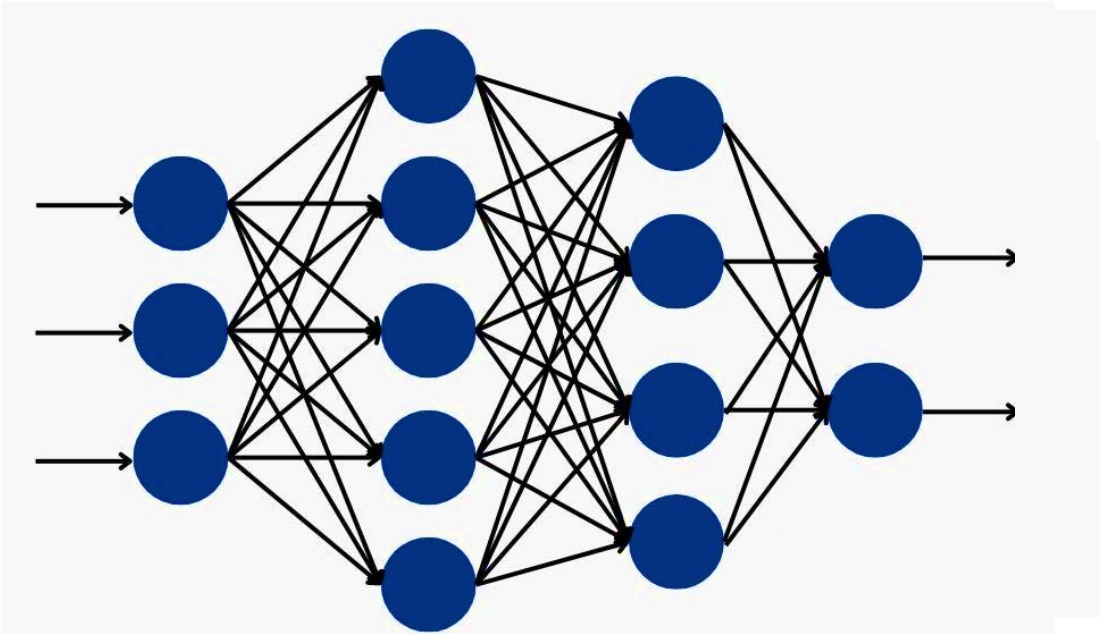
- Multilayer Perceptrons (MLPs)
- Training and Loss Functions Types
- Backpropagation Algorithm
- Stochastic Gradient Descent (SGD) and Adam Optimizer
- Common Training Issues, Regularization in Deep Learning, and Scaling Laws for Deep Learning
- Convolutional Neural Networks (CNNs)
- PixelCNN
- U-Net Denoising Model
- Recurrent Neural Networks (RNNs)
- LSTM (Long Short-Term Memory)
- GRU (Gated Recurrent Unit)
- Transformers: Self-Attention, Multi-Head Attention, and Cross-Attention
- Diffusion Transformers (DiTs)
- Vision Transformers (ViTs)
- Attention-Based U-Nets
- Multimodal Models
- Foundation Models

# Multilayer Perceptrons (MLPs)

## Basic Building Block for Fully Connected Neural Networks

# ➢ Multilayer Perceptron (MLP)

- **Structure:** MLP consists of three types of layers:
  - **Input Layer:** The first layer, which receives the input features.
  - **Hidden Layers:** One or more intermediate layers that apply transformations using weighted sums followed by activation functions.
  - **Output Layer:** The final layer that produces the network's prediction, often as logits or probabilities.
- **Key Characteristics**
  - **Fully Connected:** Each neuron in one layer is connected to every neuron in the next layer.
  - **Nonlinear Activation Functions:** Typically uses activation functions such as **ReLU**, **sigmoid**, **Softmax**, or **tanh** to introduce non-linearity.



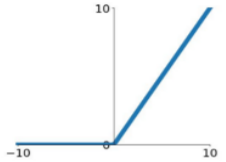
Each neuron computes a weighted sum of its inputs and applies an activation function, i.e., $y = f(Wx + b)$

$x \in R^n$: Input vector, $y \in R^o$: output vector , $W \in R^{o \times n}$ = Weight matrix, $b \in R^o$ = Bias term, $f$ Activation function

- **Backpropagation:** Trains the network using the **backpropagation algorithm** with an optimization technique like **Stochastic Gradient Descent (SGD)**

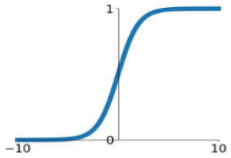
**ReLU**

$\max(0, x)$



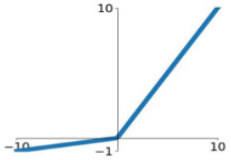

**Sigmoid**

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



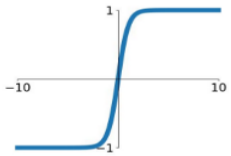

**Leaky ReLU**

$\max(0.1x, x)$




**tanh**

$\tanh(x)$



## ➢ Softmax Activation Function

- When a neural network uses **softmax** in the output layer, it transforms the final layer's raw scores (logits) into normalized probability distributions.

A neural network with **n** output neurons.

The pre-activation outputs (logits) of the final layer as $z = [z_1, z_2, \ldots, z_n]$

$$z_i = W_i^T x + b_i$$

The weight matrix $W$ and bias $b$.

Softmax Activation Function

Outputs:

$$y_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \quad i = 1, \ldots, n$$

- This ensures: $0 \leq y_i \leq 1$ for all $i$ and $\sum_{i=1}^{n} y_i = 1$

Example:

$$z = [z_1, z_2, \ldots, z_n] \rightarrow e^z = [e^{z_1}, e^{z_2}, \ldots, e^{z_n}] \rightarrow \frac{e^z}{e^{z_1} + e^{z_2} + \cdots + e^{z_n}} = [\frac{e^{z_1}}{e^{z_1} + e^{z_2} + \cdots + e^{z_n}}, \frac{e^{z_2}}{e^{z_1} + e^{z_2} + \cdots + e^{z_n}}, \ldots, \frac{e^{z_n}}{e^{z_1} + e^{z_2} + \cdots + e^{z_n}}]$$

- Softmax is widely used in generative models that involve **discrete data generation**, especially in scenarios where the model needs to output categorical distributions.

- In **Autoregressive Models** softmax is used to compute probabilities for the next tokens.

- **MLP (Multilayer Perceptron)**

A Multilayer Perceptron (MLP) is a type of fully connected feedforward neural network with at least one hidden layer.

- **Dense Layer**

A Dense layer is a fully connected layer where each neuron receives input from all neurons of the previous layer.

$$y = Wx + b$$

Activation Function (usually applied after each dense layer to introduce non-linearity)

- **Feedforward Layer**

A Feedforward layer is a general term for any layer in a neural network where information flows in one direction (forward only, no cycles).
All Dense layers are feedforward layers.
An MLP is a type of feedforward network.

# ➢ Training

**Step 1: Initialize Weights**

- Randomly initialize weights $W$ and biases $b$

For each batch in epoch:

**Step 2: Forward Pass**

- **Compute outputs by applying the forward pass for each input in the batch**: $Y = f(WX + b)$
- Compute loss $L(Y, Y_{true})$

**Step 3: Backpropagation**

- Compute gradients: $\frac{\partial L}{\partial W}, \frac{\partial L}{\partial b}$
- Adjust weights using **gradient descent**.

**Step 4: Update Weights**

- Using the learning rate $\eta$, update parameters: $W = W - \eta \frac{\partial L}{\partial W}$, $b = b - \eta \frac{\partial L}{\partial b}$

Weight updates happen after observing all the data in the batch

**Step 5: Repeat for all batches in an epoch**

- Process all batches in one epoch.
- After one epoch, the model has seen **all training samples once**.

**Step 6: Repeat for Multiple Epochs**

- Iterate multiple times until convergence.

| Term | Definition | Impact on Training |
|---|---|---|
| **Batch Size** | Number of samples per weight update | Smaller batch → Noisy updates but better generalization |
| **Epoch** | One full pass through the dataset | More epochs → Better learning, but risk of overfitting |
| **Weight Updates** | How often model parameters are updated | More updates → Faster convergence, but risk of instability |

## ➢ Loss functions

- **Regression Loss Functions** used when predicting continuous values.

  ➢ **Mean Squared Error (MSE):** $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ $y \in R^n$

- **Classification Loss Functions** used for multi-class or binary classification (when predicting **categorical** labels).

  ➢ **Categorical Cross-Entropy:** $L = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) = -\log(\hat{y}_{correct\ class})$ $y \in R^C$

  - The model outputs a probability distribution over C classes (using softmax activation).
  - $y_i \in \{0,1\}$ one-hot encoded labels
  - $\hat{y}_i \in (0,1)$ predicted probability for class $i$=1,..,C

  **Example (3-class classification):**
  True label: $y$ ="cat" → One-hot encoded: y=[1,0,0], Model prediction (after softmax): $\hat{y}$= [0.8, 0.1, 0.1], Loss = −(1·log0.8+0·log0.1+0·log0.1)=−log0.8

  ➢ **Binary Cross-Entropy (Log Loss):** $L = -\sum_{i=1}^{2} y_i \log(\hat{y}_i) = -\left( y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \right)$

  - Binary classification: BCE is a specific type of CCE that has only 2 classes
  - The model outputs a probability distribution over 2 classes (using sigmoid activation).
  - $y_i \in \{0,1\}$ true label (class 0 or class 1,e.g., fake or real image)

  **Example (Image Classification)**
  True label: y=1 (Fake Image), Model Prediction: $\hat{y}$ =0.9 (90% confidence that image is fake), Loss = −log0.9

- **Batch Loss**: $\frac{1}{N}\sum_{i=1}^{N} Loss(i)$ where $N$ is the batch size, and $Loss(i)$ is the single sample loss.

## ➢ Loss functions for Data distribution

- **Kullback-Leibler (KL) divergence** measures the distance between the true and predicted distributions:

$$KL(P||Q) = \sum_i P(i) log \frac{P(i)}{Q(i)}$$

True distribution

Predicted distribution

- **Cross-entropy** measures the distance between the true and predicted distributions:

$$H(p, Q) = -\sum_i P(i) log Q(i)$$

- Minimizing **cross-entropy** is equivalent to minimizing **KL divergence**. $H(p, Q) = KL(P||Q) - \underbrace{\sum_i P(i) log P(i)}$

Fixed (entropy of the true distribution)

- **Earth Mover's Distance (EMD):** measures the minimum cost to transform one distribution into another:

$$EMD(P, Q) = \inf_{\gamma \in \Gamma(P,Q)} \mathrm{E}_{(x,y)\sim\gamma}[\|x - y\|]$$

The set of all joint distributions whose marginals are P and Q

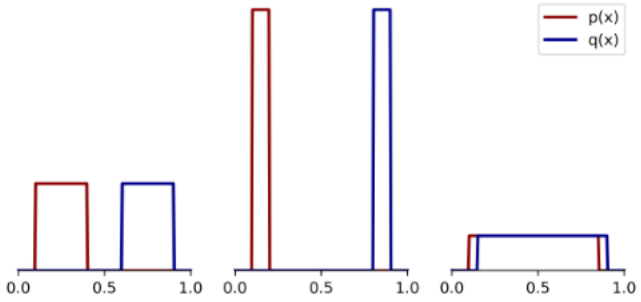
  - Also called **Wasserstein-1 distance.**
  - Takes the **geometry** of the distributions into account
  - Used in **Wasserstein GANs (WGAN)** for more stable training
  - Always finite and more **robust to disjoint supports** than KL

## ➢ Loss functions for Data distribution

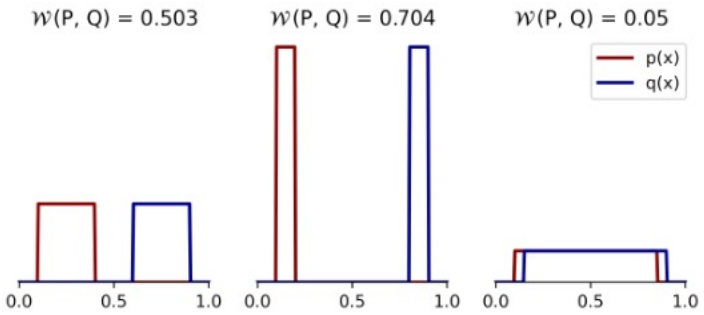
- **KL divergence** VS **Earth Mover's Distance (EMD):**

- KL divergence is sensitive to **support mismatch**. If the model (Q) assigns zero probability to something the true distribution (P) says is possible, **KL divergence becomes infinite**. That's what makes it *sensitive to support mismatch*.

- Even if P and Q have **disjoint supports**, **EMD is still finite**.
- That's one reason **Wasserstein distance** is more robust than KL divergence in practice (e.g., in WGANs).




**Three examples with infinite KL divergence.** These density functions are infinitely far apart according to KL divergence, since in each case there exist intervals of $x$ where $q(x) = 0$ but $p(x) > 0$, leading to division by zero.




**Examples in 1D revisited** Unlike KL divergence, the Wasserstein distances in these examples are finite and intuitive.

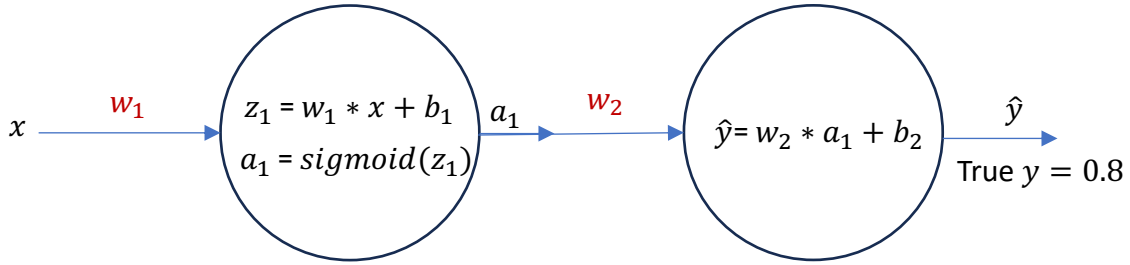
- Alex Williams, "A Short Introduction to Optimal Transport and Wasserstein Distance", 2020

- **Backpropagation Algorithm**

Backpropagation is the algorithm used to compute the gradients of the loss function with respect to each weight in the network, so we can update them using gradient descent.

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W} \qquad \text{where } \hat{y}\text{: NN output}$$

- Example of forward and backward passes in a 2-layer MLP:




Forward pass

Backward pass: $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_2}$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial W_1}$$

| Step | Value/Explanation |
|---|---|
| Input | $x = 2$ |
| Target Output | $y = 0.8$ |
| Initial Weights | $w_1 = 0.5, w_2 = -1.0$ |
| Initial Biases | $b_1 = 0, b_2 = 0$ |
| Hidden Layer Linear | $z_1 = w_1 * x + b_1 = 0.5 * 2 = 1.0$ |
| Hidden Layer Activation | $a_1 = sigmoid(z_1) \approx 0.731$ |
| Output Layer | $\hat{y} = w_2 * a_1 + b_2 = -1.0 * 0.731 = -0.731$ |
| **Squared Error Loss** | $Loss = 0.5 * (\hat{y} - y)^2 \approx 1.311$ |
| Gradient $\frac{\partial L}{\partial \hat{y}}$ | $\hat{y} - y = -0.731 - 0.8 = -1.531$ |
| Gradient $\frac{\partial \hat{y}}{\partial w_2}$ | $a_1 = 0.731$ |
| Gradient $\frac{\partial L}{\partial w_2}$ | $-1.531 * 0.731 \approx -1.120$ |
| Gradient $\frac{\partial \hat{y}}{\partial a_1}$ | $w_2 = -1.0$ |
| Gradient $\frac{\partial a_1}{\partial z_1}$ | $sigmoid(z_1)(1 - sigmoid(z_1)) \approx 0.197$ |
| Gradient $\frac{\partial z_1}{\partial w_1}$ | $x = 2$ |
| Gradient $\frac{\partial L}{\partial w_1}$ | $-1.531 * -1.0 * 0.197 * 2 \approx 0.603$ |
| Learning Rate | $\eta = 0.1$ |
| Weight Update w2 | $w_2 := -1.0 - 0.1 * (-1.120) = -0.888$ |
| Weight Update w1 | $w_1 := 0.5 - 0.1 * 0.603 = 0.4397$ |

- **Stochastic Gradient Descent (SGD)**

Stochastic Gradient Descent (SGD) is an optimization algorithm used to find the minimum of the loss function.
It's a variation of the traditional Gradient Descent, but instead of using the entire dataset to calculate the gradient, SGD uses a randomly selected data point or a small subset (mini-batch) for each update. This makes it particularly efficient for large datasets.

➢ Traditional **Gradient Descent:**

1.Loop over all $N$ training data (e.g., 10,000 examples)
2.Compute the gradient of the loss for each data sample with respect to each weight
3.Average the gradients across all samples
4.Update the weights using the mean gradient

$$\text{Average gradient: } \frac{1}{N}\sum_{i=1}^{N}\nabla_w \mathcal{L}(x^{(i)}, y^{(i)}; w)$$

$$\text{Weight update: } w := w - \eta \cdot \left(\frac{1}{N}\sum_{i=1}^{N}\nabla_w \mathcal{L}_i\right)$$

➢ **Stochastic Gradient Descent (SGD):**

- Update after every single sample
- Faster updates, but more noise in training

1.Randomly shuffle the dataset
2.For each individual training sample $(x^{(i)}, y^{(i)})$:
  a. Compute the gradient of the loss
  b. Immediately update the weights using that single gradient

$$w := w - \eta \cdot \nabla_w \mathcal{L}(x^{(i)}, y^{(i)}; w)$$

➢ **Mini-Batch Stochastic Gradient Descent:**

1. Divide the training data into **mini-batches** (e.g., batch size = 100)
2. For **each mini-batch** $\mathrm{B} = \{(x^{(i)}, y^{(i)})\}_{j=1}^{m}$:
   a. Compute the gradient for each sample in the batch
   b. **Average gradients in the mini-batch**
   c. Update the weights using the average

$$w := w - \eta \cdot \left(\frac{1}{m}\sum_{j=1}^{m}\nabla_w \mathcal{L}(x^{(j)}, y^{(j)}; w)\right)$$

- **Adam Optimizer**

- The Adam optimizer is an adaptive variant of stochastic gradient descent (SGD).
- It operates identically to SGD except that it introduces two additional variables, $s$ and $r$, representing exponential moving averages of the gradients. These are known as the first and second moments of inertia.
- The first moment of inertia of a distribution refers to its mean, while the second is the uncentered variance. More specifically, $s_i$ is the moving average of the mean of the gradient and $r_i$ is the moving average of the variance.

**Require:** Step size $\epsilon$ (Suggested default: 0.001)
**Require:** Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
**Require:** Small constant $\delta$ used for numerical stabilization (Suggested default: $10^{-8}$)
**Require:** Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$
**while** stopping criterion not met **do**
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$ element-wise square
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
**end while**


- Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization, December 22, 2014, https://arxiv.org/abs/1412.6980v8.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. Retrieved from http://www.deeplearningbook.org

## ➢ Common Training Issues

**1. Covariance Shift:** Happens when the distribution of input data changes between the training and test data distributions, making the model struggle with generalization. **Batch Normalization**[1] helps reduce this by stabilizing feature distributions across batches.

**2. Exploding Gradient:**Occurs when gradients become excessively large during backpropagation, causing unstable training or divergence. This happens in deep networks with large weight updates, and is mitigated using **gradient clipping**, proper weight initialization, or **batch normalization**.

**3. Vanishing Gradient:** Occurs when gradients become extremely small in early layers, often due to deep network depth or saturated activation functions (e.g., sigmoid or tanh), causing the weights to update minimally and slowing or halting learning. This can be mitigated by techniques like ReLU activation, proper initialization, and **batch normalization**.

**4. Overfitting**: Model memorizes training data instead of generalizing. Fix: **Regularization** (Dropout, L2: Penalizes large weights by adding an L2 penalty to the loss function to encourage smaller weight values), Data Augmentation, More Data.

**5. Underfitting**: Occurs when the model is too simple to capture underlying patterns in the data. Fix: Increase model capacity (more layers, neurons), Reduce regularization.

**6. Dead Neurons**: Neurons output zero values permanently often due to ReLU units getting stuck in the non-active region (outputting 0). Fix**:** Use Leaky ReLU or Parametric ReLU, better weight initialization.

**7. Saturated Activations** : Activations (e.g., sigmoid, tanh) get stuck near extremes, causing small gradients. Fix: Use ReLU, Batch Normalization, Proper Weight Initialization.

**8. Slow Convergence:** Model takes too long to learn. Fix**:** Learning rate scheduling, Adam optimizer, Proper initialization.

- **Degradation Problem in Deep Networks:** The **degradation problem** refers to the phenomenon where **increasing the depth** of a deep neural network does not necessarily lead to better performance. Instead, deeper networks often experience: Vanishing Gradient, Overfitting, Performance Saturation, Optimization Difficulties.

1: See CNNs for batch normalization.

## ➢ Regularization in Deep Learning

- Regularization in Deep Learning refers to a set of techniques used to **prevent overfitting**—which occurs when a neural network learns to perform well on training data but fails to generalize to unseen data.

- To improve the **generalization** ability of a deep learning model by discouraging it from fitting noise or overly complex patterns in the training data.

Without regularization:

- Neural networks can have large weight magnitudes, allowing them to fit the training data extremely well—even the noise.
- This leads to low training loss but high validation/test loss (poor generalization).

With regularization:

- The model is guided to learn simpler and more generalizable patterns that capture the underlying structure of the data.

| Technique | Description | Effect |
|---|---|---|
| **L1 Regularization (Lasso)** | Adds a penalty proportional to the **absolute values** of weights to the loss function: loss + λ | |
| **L2 Regularization (Ridge)** | Adds a penalty proportional to the **squared values** of weights: loss + λ | |
| **Dropout**[1] | Randomly disables a fraction of neurons during training | Reduces dependency on specific neurons; prevents co-adaptation |
| **Early Stopping** | Stops training when performance on validation set stops improving | Avoids overfitting beyond optimal epoch |
| **Data Augmentation** | Increases training data by transforming inputs (e.g., rotation, cropping) | Makes model more robust and less likely to memorize training data |
| **Batch Normalization** | Normalizes inputs to each layer | Has regularization effects by adding noise during training |
| **Weight Decay** | A form of L2 regularization often implemented directly in optimizers (like in Adam or SGD) | Penalizes large weights during optimization |

- For more info: Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. Retrieved from http://www.deeplearningbook.org

1: See CNNs for Dropout.

## ➢ Scaling Laws for Deep Learning

- **Scaling laws** describe how the performance of machine learning models — especially **large language models (LLMs)** — improves as you scale up core factors such as **model size, dataset size, and compute power**.
- They are empirical relationships showing that model performance improves predictably—often logarithmically or according to a power law—as the scale increases. This is typically expressed as: $Loss \approx A.N^{-\alpha}$,
  where, $N$ represents model size, data size, or compute budget, $\alpha$ is a positive scaling exponent, and $A$ is a fitting constant. As $N$ increases, the **loss decreases smoothly** on a log-log scale.

- **Key Ideas from Scaling Laws in LLMs:**
  - **Bigger Models Perform Better:** As you increase the number of parameters, test loss and downstream performance improve — assuming enough data.
  - **More Data Helps, But Diminishing Returns:** After a certain point, increasing data without increasing model size helps less.
  - **Compute Budget Must Be Balanced:** There's an optimal trade-off between model size, training steps, and data — for a fixed compute budget.
    - Proposed **Chinchilla scaling law**: for a fixed compute budget, smaller models trained on **more data** outperform very large models trained on less data.
    - **The Optimal Trade-off Rule (from Chinchilla):** For a fixed compute budget C, the optimal model size $N$ and training tokens $D$ satisfy approximately: $D \propto N$. That is, If we double the model size, we should also double the amount of training data (and compute). Otherwise, we're not using your compute efficiently.

- Kaplan et al. (Open AI 2020) in *"Scaling Laws for Neural Language Models"*
- Hoffmann et al. (2022), in the Chinchilla paper *"Training Compute-Optimal Large Language Models"*

# Convolutional Neural Networks (CNNs)


- Kaiming He et al., “Deep Residual Learning for Image Recognition," December 10, 2015, https://arxiv.org/abs/1512.03385
- Alex Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,, April 8, 2009, https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.
- Vincent Dumoulin and Francesco Visin, “A Guide to Convolution Arithmetic for Deep Learning”, 2018, https://arxiv.org/abs/1603.07285.
- Van Den Oord, Aäron, Nal Kalchbrenner, and Koray Kavukcuoglu. “Pixel recurrent neural networks”, International conference on machine learning. PMLR, 2016. https://arxiv.org/abs/1601.06759
- Sergey loffe and Christian Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, 2015, https://arxiv.org/abs/1502.03167 .
- Nitish Srivastava et al., “Dropout: A Simple Way to Prevent Neural Networks from Overfitting," Journal of Machine Learning Research 15 (2014), https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf
- Foster, David. Generative deep learning. " O'Reilly Media, Inc.", 2022.

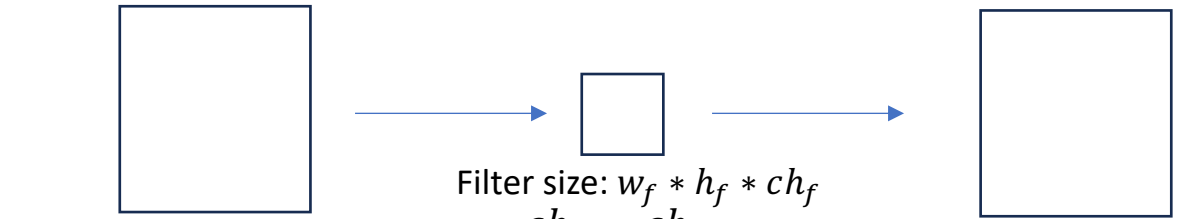
# ➢ Convolutional Neural Network (CNN)

- Primarily used for processing structured grid data, such as images.
- CNNs account for the spatial structure of the input data.

➢ **Key Component**

- **Convolution Layers:**
  - Apply convolution operations using **small filters (kernels)** that slide over the input.
  - Filters affect the input data dimensions (width, height, and number of channels).
  - These filters learn spatial hierarchies of features such as edges, textures, and shapes.

- **Filter Parameters:** include width, height, and number of input channels (which must match the input's depth)
  - Stride: Step size used to move the filter across the input data. Increasing the stride, reduces the size of output data.
  - Padding: Adds extra pixels (usually zeros) around the input to control the spatial size of the output feature map image before applying a convolution operation.



Input data size: $w_I * h_I * ch_I$

Filter size: $w_f * h_f * ch_f$

$ch_f = ch_I$

Output data size: $\frac{w_I}{stride} * \frac{h_I}{stride} * number\ of\ filters$

Padding=same

- General case: Output data size: $\frac{w_I - w_f + 2Padding_w}{stride_w} + 1 * \frac{h_I - h_f + 2Padding_h}{stride_h} + 1 * number\ of\ filters$

**Example:**

**Input:** 3×3×3 Image (Each Pixel Has 3 RGB Channels)

$$\begin{bmatrix} (1,2,3) & (4,5,6) & (7,8,9) \\ (10,11,12) & (13,14,15) & (16,17,18) \\ (19,20,21) & (22,23,24) & (25,26,27) \end{bmatrix}$$

**Filter** 2×2×3 : $\begin{bmatrix} (1,0,-1) & (1,0,-1) \\ (0,1,0) & (-1,1,-1) \end{bmatrix}$ **Padding=0** **Stride=1**

Output data size: $\frac{3-2}{1}+1 * \frac{3-2}{1}+1 * 1$

Element-wise product

Filter=$\begin{bmatrix} (1,0,-1) & (1,0,-1) \\ (0,1,0) & (-1,1,-1) \end{bmatrix}$

$$\begin{bmatrix} (1,2,3) & (4,5,6) & (7,8,9) \\ (10,11,12) & (13,14,15) & (16,17,18) \\ (19,20,21) & (22,23,24) & (25,26,27) \end{bmatrix} \Rightarrow \begin{bmatrix} (1,2,3).*(1,0,-1) & (4,5,6).*(1,0,-1) \\ (10,11,12).*(0,1,0) & (13,14,15).*(-1,1,-1) \end{bmatrix} = \begin{bmatrix} (1,0,-3) & (4,0,-6) \\ (0,11,0) & (-13,14,-15) \end{bmatrix}$$

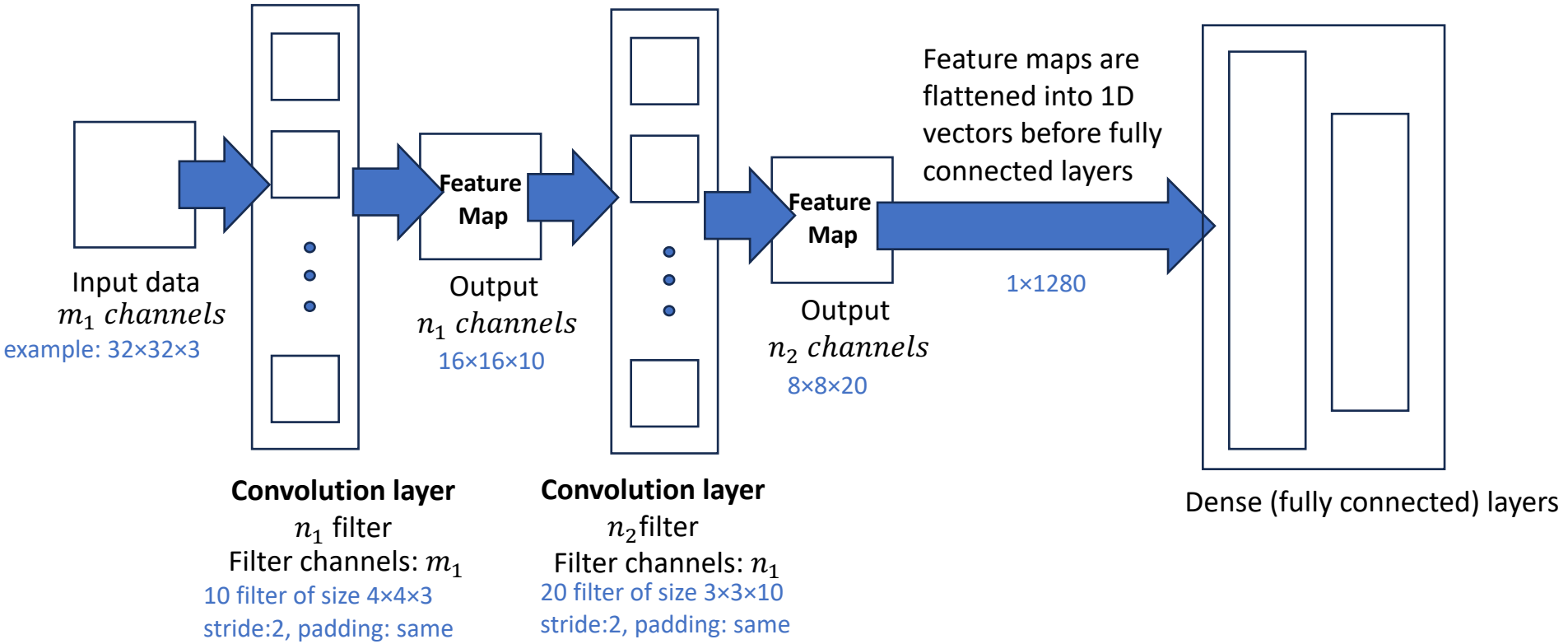
Summing all values ↔ 1+0−3+4+0−6+0+11+0−13+14−15=−7

move the filter across the input and perform the same operation:

$$\begin{bmatrix} (1,2,3) & (4,5,6) & (7,8,9) \\ (10,11,12) & (13,14,15) & (16,17,18) \\ (19,20,21) & (22,23,24) & (25,26,27) \end{bmatrix} \quad \begin{bmatrix} (1,2,3) & (4,5,6) & (7,8,9) \\ (10,11,12) & (13,14,15) & (16,17,18) \\ (19,20,21) & (22,23,24) & (25,26,27) \end{bmatrix} \quad \begin{bmatrix} (1,2,3) & (4,5,6) & (7,8,9) \\ (10,11,12) & (13,14,15) & (16,17,18) \\ (19,20,21) & (22,23,24) & (25,26,27) \end{bmatrix}$$

**Output** 2×2×1 : $\begin{bmatrix} -7 & -7 \\ -7 & -7 \end{bmatrix}$

# Convolutional Neural Network




**Convolution Layers:**

- Filter depth should match the input depth
- The number of filters determines the output depth
- Standard convolution reduces spatial dimensions when stride >=2
- In CNNs, we typically increase the number of channels and decrease the spatial size of the feature map before transitioning to Dense (fully connected) layers.

**Dense (Fully Connected) Layers:**

- After convolution, the high-level features are flattened into a vector.
- This vector is passed through one or more fully connected layers.

## ➢ Optional Components:

Input → Conv Layer→ Batch Normalization → Activation → Pooling → Flatten → Dense Layer → Dropout → Dense Layer→ Output

**Batch Normalization (optional):**

- Normalizes feature maps across a batch to **speed up training** and **stabilize learning**.
- Helps in **faster convergence** and **reducing internal covariate shift**.

➢ **Algorithm:**

**Feature Map**

$x \in R^{m \times n_1 \times n_2 \times C}$

m = batch size

$C$ = number of channels

$n_1$ , $n_2$ = spatial dimensions (height, width)

- **For each Channel $C$ perform:**

**1. Mean:** Batch Normalization is applied **per channel** C, meaning the **mean is computed over all** $n_1 \times n_2 \times m$ spatial elements per channel

$$\mu_c = \frac{1}{m * n_1 * n_2} \sum_{i=1}^{m} \sum_{j=1}^{n_1} \sum_{k=1}^{n_2} x_{ijk}$$

a single element

**2. Variance:** Similarly, the **variance per channel**

$$\sigma_c^2 = \frac{1}{m * n_1 * n_2} \sum_{i=1}^{m} \sum_{j=1}^{n_1} \sum_{k=1}^{n_2} (x_{ijk} - \mu_c)^2$$

**3. Normalization Step:** Each element in each channel is **normalized** using: $\hat{x}_{ijk} = \frac{x_{ijk} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$ small constant for numerical stability

**4. Scale and Shift Step**: The learnable scale $\gamma_c \in R$ and shift $\beta_c \in R$ parameters are applied: $y_{ijk} = \gamma_c \hat{x}_{ijk} + \beta_c$

- **Training:** Uses batch statistics (mean/variance computed from the batch).
- **Inference:** Uses learned moving average statistics (fixed mean/variance).

$\mu_{new} = (1 - \alpha)\mu_{old} + \alpha\mu_c$ For current channel and batch

$\sigma^2_{new} = (1 - \alpha)\sigma^2_{old} + \alpha\sigma_c^2$ $\alpha$ ~0.1 momentum

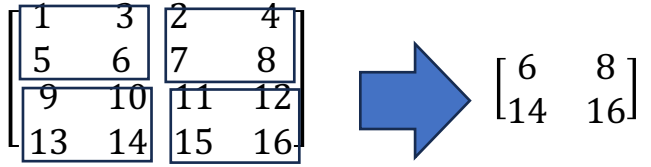
➢ **Activation Function** (e.g., ReLU)

- Introduce non-linearity
- Applies element-wise to the elements of input tensor

➢ **Pooling Layer** (optional)

- It is used to reduce the spatial size of feature maps while retaining the most important information.
- Reducing computation (fewer parameters)
- A pooling operation moves a small window (filter) over the input feature map, applying an operation like max (Takes the **maximum value** in each window) or average (Takes the **average** of values in each window) to each window.

Output data size: $\frac{Input\ size - Pool\ Size}{Stride} + 1$




4×4 feature map
2×2 filter
stride=2

**Max Pooling**

➢ **Dropout Layer (optional):**

- Randomly disables neurons during training to prevent overfitting.
- Example: A dropout rate of 0.5 means 50% of the neurons in the specific layer where dropout is applied are randomly turned off during each forward pass.
- **Training:** Randomly disables neurons to prevent overfitting. **Inference:** During inference, dropout is disabled — all neurons remain active.

**Input → Conv Layer→ Batch Normalization → Activation → Pooling → Flatten → Dense Layer → Dropout → Dense Laye r→ Output**

**Input:** Image data
**Conv Layer:** Extracts features using filters (kernels)
**Batch Normalization:** Normalizes activations before activation
**Activation Function** (e.g., ReLU): Introduces non-linearity
**Pooling** (e.g., Max Pooling): Reduces spatial dimensions
(Optional) More Conv Layers + Pooling: Deeper feature extraction
**Flatten:** Converts feature maps into a vector for Dense layers
**Dense Layers + Dropout** (Fully Connected Layers): Processes extracted features
**Output:** Classification (Softmax) or Regression (Linear Activation).

- **Example:**

CNN Architecture of Image Classification (10-Class) Problem

| Layer (type) | Output shape | Param # |
|---|---|---|
| InputLayer | (32, 32, 3) | 0 |
| Conv Layer | (32, 32, 32) | 896 |
| BatchNormalization | (32, 32, 32) | 128 |
| LeakyReLU | (32, 32, 32) | 0 |
| Conv Layer | (16, 16, 32) | 9,248 |
| BatchNormalization | (16, 16, 32) | 128 |
| LeakyReLU | (16, 16, 32) | 0 |
| Conv Layer | (16, 16, 64) | 18,496 |
| BatchNormalization | (16, 16, 64) | 256 |
| LeakyReLU | (16, 16, 64) | 0 |
| Conv Layer | (8, 8, 64) | 36,928 |
| BatchNormalization | (8, 8, 64) | 256 |
| LeakyReLU | (8, 8, 64) | 0 |
| Flatten | (4096) | 0 |
| Dense | (128) | 524,416 |
| BatchNormalization | (128) | 512 |
| LeakyReLU | (128) | 0 |
| Dropout | (128) | 0 |
| Dense | (10) | 1290 |
| Softmax | (10) | |

➢ **Transpose Convolution Layer:**

Also called **Deconvolution** or **Fractionally Strided Convolution** is used to **increase the spatial resolution** of feature maps, commonly in **upsampling tasks** like image generation and segmentation

- Standard Convolution reduces the spatial size
- **Transpose Convolution does the reverse**—it **increases** the spatial size
- It is commonly used in Decoder networks (e.g., U-Net, GANs, Autoencoders).

- Instead of **sliding a filter over an image** like in normal convolution, **Transpose Convolution spreads each input pixel into a larger area**.

**Example (3×3 Input, 2×2 Filter, Stride = 2)**

1. Each input pixel is expanded — resembling the inverse of convolution — to cover a larger output region.
2. The **filter values are multiplied and added** in overlapping regions.
3. The output size becomes **larger** than the input.

Output data size:$(w_I-1)Stride + w_f - 2Padding_w * (h_I-1)Stride + h_f - 2Padding_h * number\ of\ filters$

- **Example**

$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$ Input

$\begin{bmatrix}0 & 1\\2 & 3\end{bmatrix}$ filter

$\begin{bmatrix}\ & \ \\ \ & \ \end{bmatrix}$ Output: ((2-1)×1+2)*((2-1)×1+2)

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

$\begin{bmatrix}0 & 1\\2 & 3\end{bmatrix}\begin{bmatrix}0 & 2\\4 & 6\end{bmatrix}\begin{bmatrix}0 & 3\\6 & 9\end{bmatrix}\begin{bmatrix}0 & 4\\8 & 12\end{bmatrix}$

$\begin{bmatrix}0 & 1\\2 & 3\end{bmatrix}+\begin{bmatrix}0 & 2\\4 & 6\end{bmatrix}+\begin{bmatrix}0 & 3\\6 & 9\end{bmatrix}+\begin{bmatrix}0 & 4\\8 & 12\end{bmatrix}$

$\begin{bmatrix}0 & 1 & 2\\2 & 10 & 10\\6 & 17 & 12\end{bmatrix}$

# ➢ Masked Convolutional Layer

A **masked convolutional layer** is a specialized type of convolutional layer used primarily in autoregressive models, such as **PixelCNN**, where it ensures that the output at any given position does not depend on future (yet-to-be-predicted) values.

- In a standard convolutional layer, the entire convolutional kernel is applied to the input, so each output pixel is influenced by all pixels within the kernel's receptive field. Standard convolutions can freely access information from all surrounding pixels.
- In a masked convolutional layer, certain weights are zeroed out (masked) to enforce constraints, such as preventing information from future pixels from influencing the current pixel. Masked convolutions enforce **causality** by limiting access to past or adjacent data points, which is crucial for sequential generation.

- To apply convolutional layers to an image in an autoregressive manner, we must first define an ordering for the pixels and ensure that the filters can only access past pixels while preventing any influence from future pixels.
- Pixel ordering from top-left to bottom-right ensures autoregressive flow

**Example**

3×3 Filter

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

3×3 Masked Filter

$$\begin{bmatrix} a & b & c \\ d & e & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

**Types of Masks:**

**Mask Type A:** Used in the first convolutional layer to ensure that each pixel does not have access to itself or future pixels.

**Mask Type B:** Used in deeper layers where self-access is allowed but future access is still blocked.

# ➢ Residual Block

It introduces **skip (or shortcut) connections** which allow a layer's input to bypass one or more intermediate layers and be added directly to the output. It helps to build deeper networks that can learn more complex patterns without suffering as greatly from vanishing gradient and degradation problems.

- Standard Neural Network Block: $y = f(x)$

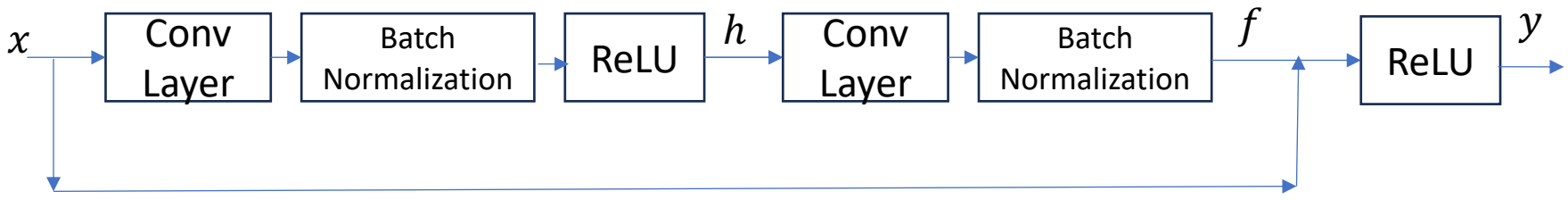
  where $f$ represents one or more layers like convolution, batch normalization, and activation

- Residual Block: $y = x + f(x)$

  The skip connection directly adds $x$ to the output of $f(x)$. It allows gradients to flow more easily (mitigating vanishing gradients)

- **Basic Residual Block**

$$h = ReLU(BN(Conv(x)))$$
$$f(x) = BN(Conv(h))$$
$$y = ReLU(x + f(x))$$

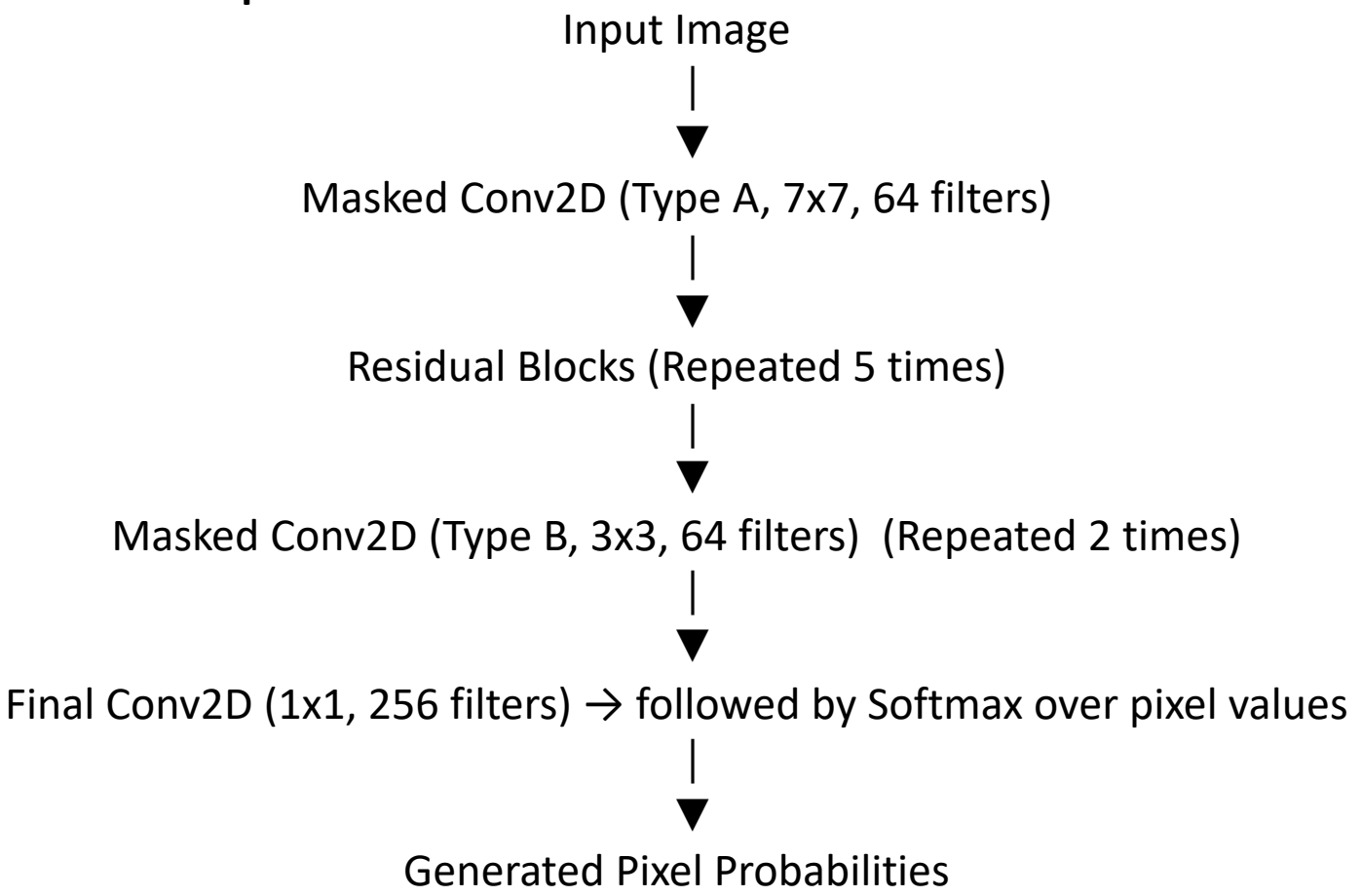


# ➢ PixelCNN

Combination of masked convolution layers, residual blocks, and a standard convolution layer with a 1×1 filter.

- **Pixel-by-pixel generation:** The network generates an image sequentially, one pixel at a time. At each step, it predicts a probability distribution over possible pixel values for the next pixel.
- **Autoregressive Dependency:** Ensures that each pixel is conditioned only on previously generated pixels, enforced through masked convolutions.
- **Discrete Softmax Output:** Instead of predicting continuous pixel values, PixelCNN outputs a probability distribution over 256 discrete intensity levels per pixel (0–255)

**Example:**




- More Information: Van Den Oord, Aäron, Nal Kalchbrenner, and Koray Kavukcuoglu. “Pixel recurrent neural networks”, International conference on machine learning. PMLR, 2016. https://arxiv.org/abs/1601.06759

# U-Net Denoising Model


- Ronneberger, O., Fischer, P., & Brox, T.,“U-Net: Convolutional networks for biomedical image segmentation”, In Medical Image Computing and Computer-Assisted Intervention (MICCAI), 2015, https://arxiv.org/abs/1505.04597
- Ho, J., Jain, A., & Abbeel, P. ,”Denoising Diffusion Probabilistic Models”, In Advances in Neural Information Processing Systems (NeurIPS), 2020. https://arxiv.org/abs/2006.11239
- Ho, J., & Salimans, T., “Classifier-Free Diffusion Guidance”, OpenAI, 2022 https://openai.com/research/guided-diffusion
- Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B., “High-Resolution Image Synthesis with Latent Diffusion Models”, In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2022 https://arxiv.org/abs/2112.10752
- Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., & Chen, M.,”Hierarchical Text-Conditional Image Generation with CLIP Latents”, 2022 https://arxiv.org/abs/2204.06125
- Foster, David. Generative deep learning. " O'Reilly Media, Inc.", 2022.

# ➢ U-Net Denoising Model (used in diffusion models)

This model has two parts:

- **Downsampling** : input images are compressed spatially but expanded channel-wise
- **Upsampling** : representations are expanded spatially while the number of channels is reduced.
- **Skip Connections:** There are skip connections between equivalent spatially shaped layers in the upsampling and downsampling parts of the network. Skip connections allow information to shortcut parts of the network and flow through to later layers.

➢ **Architecture:**

**Inputs: Noisy image and Noise variance (scaler), output: predicted noise on image**

**1) Input Layers:**

- Input image (e.g., 64x64x3) → Conv Layer to increase the number of channels (e.g., 64x64x32)
- Noise variance input (scalar) → sinusoidal embedding to covert it into a vector (e.g., 1x1x32) → upsampling where the embedding vector is copied across spatial dimensions to match the size of the input image (e.g., 64x64x32)
- Concatenate: The two inputs are concatenated across channels (e.g., 64x64x64)

**2) Downsampling**: The tensor is passed through a series of DownBlock layers that reduce the size of the tensor, while increasing the number of the channels, e.g., (e.g., 64x64x64 → → → → 8x8x128). The skip connections will create a shortcuts to Upsampling layers along the way.

**3) Residual Block**: The tensor is then passed through two ResidualBlock layers that hold the image size and number of channels constant (e.g., 8x8x128 → 8x8x128).

**4) Upsampling**: Next, the tensor is passed through a series of UpBlock layers that increase the size of the image, while decreasing the number of channels e.g., (e.g., 8x8x128 → 64x64x32). The skip connections incorporate output from the earlier DownBlock Layers.

**5) Output Layer:** The final Con layer reduces the number of channels to three (RGB)**,** (e.g., 64x64x32 → 64x64x3)

# ➢ U-Net Denoising Model

➢ **Components:**

**1) sinusoidal embedding:** Goal is to convert a scalar value into a distinct higher dimensional vector.

A scalar value x is encoded as follows:

$$y(x) = \left(\sin\left(2\pi e^{0f}x\right), \ldots, \sin\left(2\pi e^{(L-1)f}x\right), \cos\left(2\pi e^{0f}x\right), \ldots, \cos\left(2\pi e^{(L-1)f}x\right)\right)$$

where $L$ is chosen to be half the size of the desired noise embedding length and $f = \frac{\ln 1000}{L-1}$ which controls the maximum frequency used for encoding.

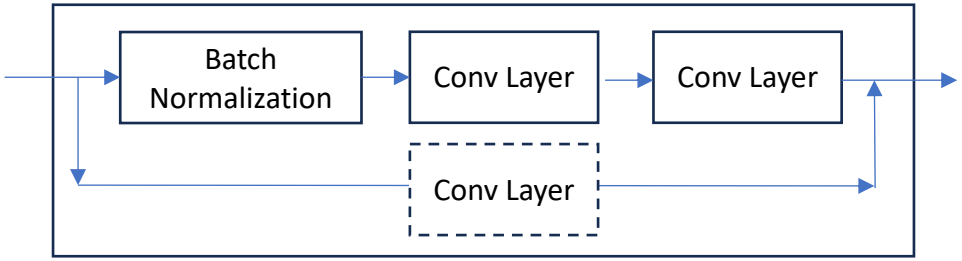
**2) Residual Blocks:** are group of layers that contains a skip connection that adds the input to the output.

An additional 1x1 convolution layer can be used on the skip connection to align channel dimensions with the main path.

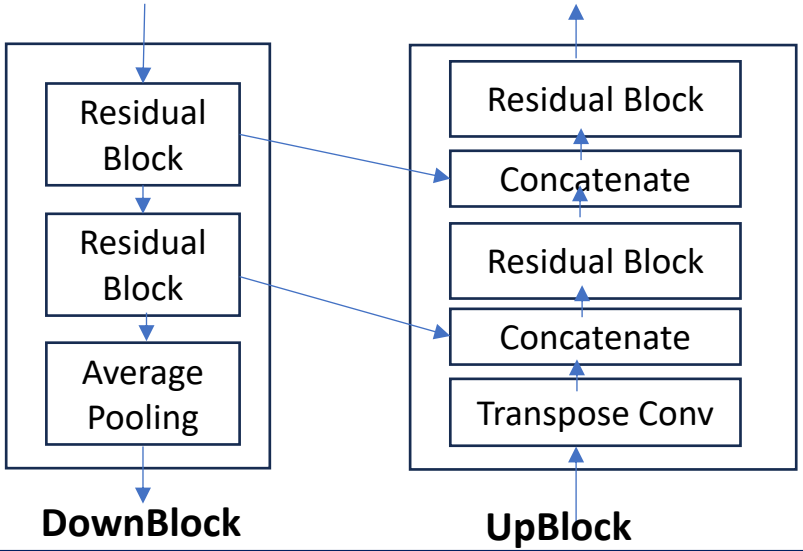
e.g., $y = x + f(x)$ where $f(x)$ has 128 channels and $x$ has 256 channels

$y = conv(x) + f(x)$ to match the channel numbers

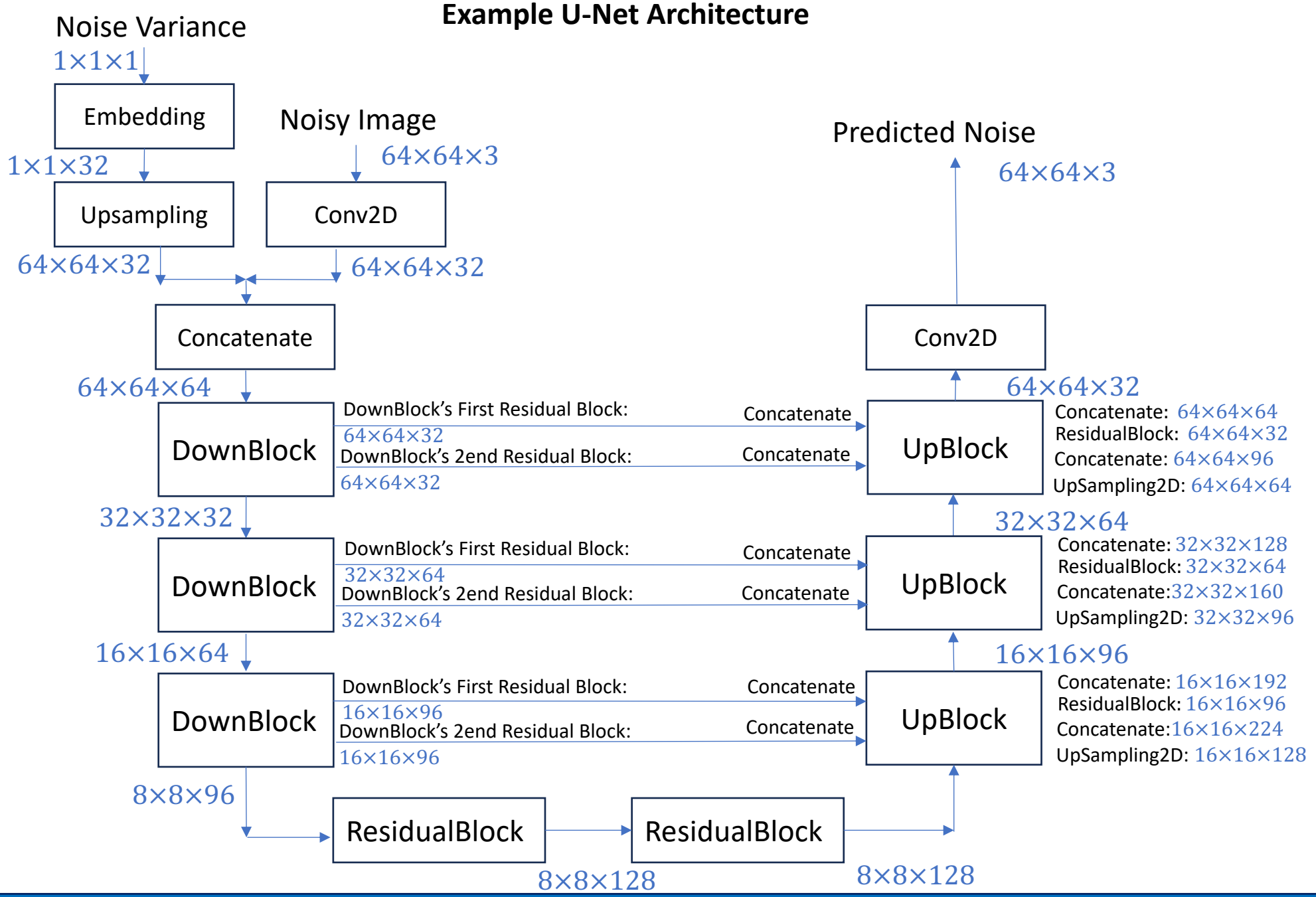
Conv layer including 128 filters of size 1x1 reduces the channel number of $x$ to 128




**3) Down and Up Blocks**



# Example U-Net Architecture



# Recurrent Neural Networks (RNNs)


- Sepp Hochreiter and Jürgen Schmidhuber, “Long Short-Term Memory”, Neural Computation 9, 1735-1780, 1997, https://www.bioinf.jku.at/publications/older/2604.pdf
- Kyunghyun Cho et al., “Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation”, 2014, https://arxiv.org/abs/1406.1078
- Aaron van den Oord et al., “Pixel Recurrent Neural Networks”, In International conference on machine learning (pp. 1747-1756). PMLR, 2016, https://arxiv.org/abs/1601.06759.
- Tim Salimans et al., "PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications”, 2017, http://arxiv.org/abs/1701.05517

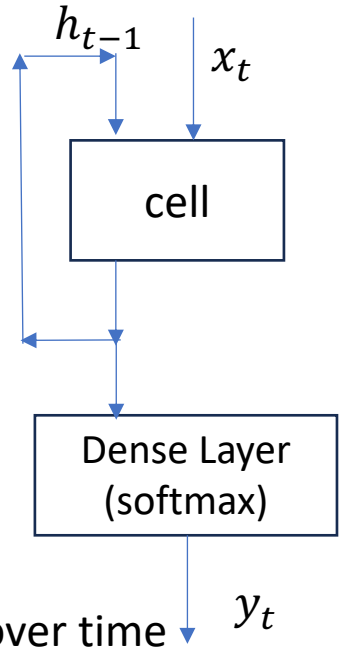
## ➢ Recurrent Neural Networks

RNNs are a type of neural network designed for processing sequential data. Unlike feedforward neural networks, RNNs have recurrent connections that allow information to persist over time, making them suitable for tasks involving sequences, such as time-series prediction, speech recognition, and language modeling.

- Standard RNN
- LSTM (Long Short-Term Memory)
- GRU (Gated Recurrent Unit)

# ➢ Standard RNN

- **Structure:**
  - **Input layer:** Takes sequential data as input.
  - **Hidden layer with recurrent connections:** Maintains a hidden state that evolves over time.
  - **Output layer:** Generates predictions at each time step.




At each time step t, an RNN processes:

**1. Current Input ($\boldsymbol{x_t} \in \boldsymbol{R^n}$)**: The feature vector at time t, e.g., embedding vector of a token

**2. Previous Hidden State ($\boldsymbol{h_{t-1}} \in \boldsymbol{R^m}$)**: Captures past information, m: number of neurons in the dense layer

**3. New Hidden State ($\boldsymbol{h_t}$ )**: Updated using the current input and previous hidden state.

- **Hidden State Update:** $\boldsymbol{h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h)}$, Learnable parameters ($W_h$, $W_x$, $b_h$) are fixed over time

where, $h_t$ = new hidden state, $W_h$ = weight matrix for the hidden state, $W_x$ = weight matrix for the input, $b_h$ = bias , and $\sigma$= activation function, typically tanh or ReLU

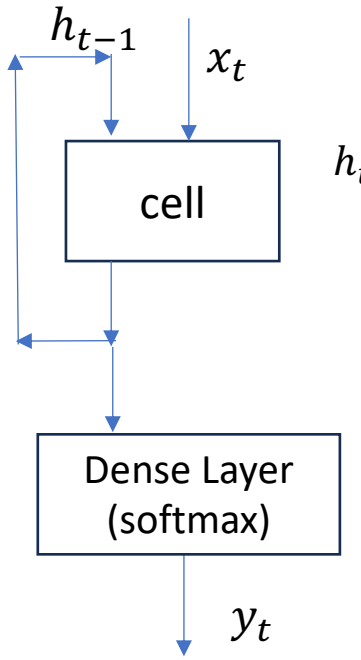
- **Output Calculation:** $\boldsymbol{y_t = softmax(W_y h_t + b_y)}$, Learnable parameters ($W_y$, $b_y$) are fixed over time

where, $y_t$ = predicted output (e.g., probability distribution over words in a language model), $W_y$ = weight matrix for the output, $b_y$ = bias, and the softmax function ensures the output probabilities sum to 1.

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h) \quad \Longleftrightarrow \text{Similar} \quad h_t = f(h_{t-1}, x_t)$$
$$y_t = softmax(W_y h_t + b_y) \qquad\qquad y_t = g(h_t)$$

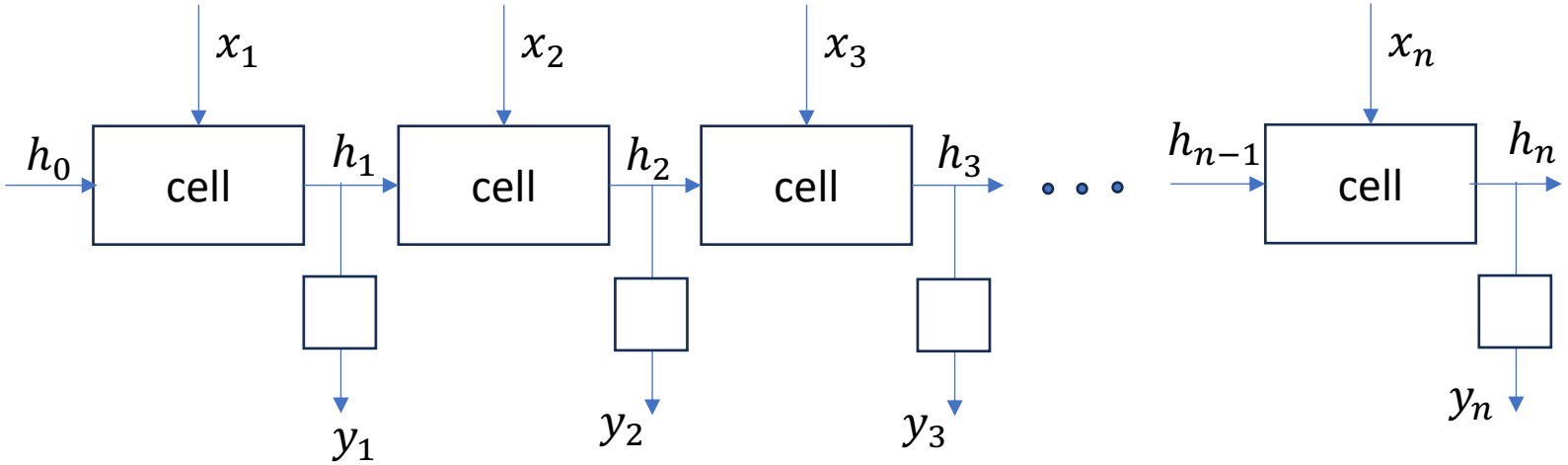
**RNN** **State-space model of dynamical systems**

## Single RNN Cell



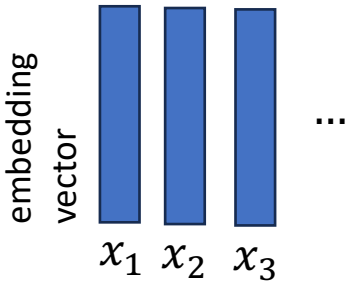

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h)$$
$$y_t = sofmax(W_y h_t + b_y)$$

## Unrolled RNN




Example:
$x_t$: embedding vector of token
$y_t$: probability vector over the possible tokens to predict the next token

Input Token seq: $\{Id_1, Id_2, Id_3, \dots\}$




Initialize $h_0$ (hidden state) with zeros.

- For each time step $t$: i) Compute the new hidden state $h_t$, ii) Compute the output $y_t$, iii) Pass $h_t$ to the next time step.
- Final output can be: i) at each time step (e.g., in speech recognition), ii) Only at the last step (e.g., text classification).

- **When training** an RNN, we **unroll** it over multiple time steps to explicitly show how information propagates over time. This is necessary because **the same weight matrix $W_h$ is used at every time step**. Since $W_h$ is reused, the **gradient** computation must be **accumulated** across all time steps, which is done using **Backpropagation Through Time (BPTT)**.

$\frac{\partial L}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial W_h}$ this is because **we use same $W_h$ over time**, otherwise we would have $\frac{\partial L}{\partial W_h^t} = \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial W_h^t}$

## Example: Text generation

- **Step 1: Data Preparation**

**1. Collect a corpus** (set of sentences) for training. Example

"I love machine learning"
"I love deep learning"
"I love natural language processing"
"Deep learning is powerful"
"Machine learning is fascinating"

**2. Tokenization:** Convert words into numerical tokens using a word index. Example:

{'i': 1, 'love': 2, 'machine': 3, 'learning': 4, 'deep': 5, ...}
Each word is assigned a unique integer.

**3. Create training sequences:** Given a sentence, we split it into a set of input-output training sequences. Each sequence is a partial sentence where the first n−1 words are input, and the nth word is the target. Example:
Input: ["I"] → Target: "love“ **t=1**
Input: ["I", "love"] → Target: "machine“ **t=2**
Input: ["I", "love", "machine"] → Target: "learning“ **t=3**

**4. Padding sequences (optional):** Since sequences vary in length, shorter sequences are padded to ensure uniform input size.

- **Step 2: Training the RNN**

**1. Embedding Layer:** Words are represented using dense word embeddings (learned during training). Example: "love" → Vector representation (e.g., [0.1, 0.4, -0.3, ...]).

**2. Recurrent Layer:** The RNN processes sequences step by step. At each step: The **input word's embedding** and the hidden state from the previous step are used to compute a **new hidden state**. Example of hidden state update $h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h)$.This enables the network to retain contextual information from previous words.
**3. Output Layer:** A fully connected (Dense) layer maps the hidden state to a probability distribution over the vocabulary. The softmax activation function is applied to predict the most likely next word. $y_t = softmax(W_y h_t + b_y)$

**4.Loss Calculation & Backpropagation:** The predicted word is compared to the actual next word. The cross-entropy loss is computed and used to update the model's weights using backpropagation through time (BPTT).

**Step 3: Generating Text (Sampling Process)**

Once the model is trained, we can generate text by **predicting the next word iteratively**:
i) Choose a seed phrase, e.g., "I love".
ii) Convert it to tokens using the same word index.
ii) Feed it into the trained RNN,
iv) The model predicts the next word's probability distribution. The most probable word is selected (or sampled with some randomness).
v) Append the predicted word to the seed phrase.
vi) Repeat the process until reaching a desired length. This process continues until a **termination condition** is met (e.g., reaching a **max length** or **predicting an end-of-sequence token**).

## ➢ Limitations of Standard RNNs

**1. Vanishing Gradient Problem**
During backpropagation, gradients become **exponentially smaller** as they propagate **back in time**.
This means **early time steps** have **almost no influence** on updates, making it hard to learn long-range dependencies.
The recurrent weight matrix $W_h$ is repeatedly multiplied during backpropagation. If its eigenvalues are <1, gradients shrink toward zero → **vanishing gradient problem** .
**Impact**: The RNN **forgets** long-term dependencies and struggles with sequences that require long memory (e.g., long sentences, paragraphs). **Solution:** Use **LSTM / GRU** (gated architectures)

**2. Short-Term Memory:** Due to vanishing gradients, Standard RNNs **tend to capture only short-term dependencies.**
Example problem: Given the sentence: "The cat, which was sitting on the mat, is very cute.“ A standard RNN may struggle to associate "cat" with "cute" because too many words separate them.
**Impact:** Standard RNNs struggle with long-term dependencies. **Solution:** Use **LSTM / Transformer models**

**3. Exploding Gradient Problem:** If the weight matrix $W_h$ has **large eigenvalues**, gradients can become **excessively large**. This leads to **unstable training**, where weight updates diverge instead of converging. **Solution**: Use **gradient clipping** to cap the gradient values.

**4. Difficulty in Parallelization:** RNNs process inputs **sequentially**, meaning **each step depends on the previous step**. Unlike CNNs (which process in parallel), RNNs **cannot fully utilize GPUs** for fast computation.
**Impact**: Training on large datasets is **slow and inefficient**. **Solution:** Use **Transformer-based architectures**

**5. Fixed-Length Memory (Hidden State Size):** The **hidden state ht** has a **fixed size**, regardless of input length.
This means: i) The network has **limited capacity** to store information., ii) It **compresses too much information**, leading to information loss. **Solution:** Use **Attention Mechanisms** to dynamically focus on relevant information

**6. Sensitivity to Input Order:** RNNs are **order-sensitive**, meaning a small word order change can alter predictions drastically. **Solution:** Use **Bidirectional RNNs** or **Transformers**, which capture both forward and backward context.

## ➢ LSTM (Long Short-Term Memory)

LSTMs are an advanced variant of Recurrent Neural Networks (RNNs) designed to handle **long-term dependencies** by overcoming the **vanishing gradient problem**. Unlike standard RNNs, which only have a single **hidden state** ($\boldsymbol{h_t}$ ), LSTMs introduce a **cell state** ($\boldsymbol{c_t}$) and **gating mechanisms** to control the flow of information.

- **Key Components of an LSTM Cell**

Each LSTM cell contains:

1. **Cell State ($\boldsymbol{c_t}$)**: Stores long-term information.
2. **Hidden State ($\boldsymbol{h_t}$)**: Short-term memory used for predictions.
3. **Forget Gate ($\boldsymbol{f_t}$ )**: Determines how much of the previous cell state $\boldsymbol{c_{t-1}}$ to retain or discard. $\boldsymbol{f_t}$ contains values between 0 (forget everything) and 1 (keep everything).
4. **Input Gate ($\boldsymbol{i_t}$ )**: Decides how much new information to store. Contains values between 0 and 1.
5. **Candidate Cell State ($\boldsymbol{\tilde{c}_t}$ )**: New candidate memory. Contains values.
6. **Output Gate ($\boldsymbol{o_t}$ )**: Controls what part of the cell state is passed to the hidden state.

- **Why Is LSTM Better for Long-Term Dependencies?**
  - Cell state ($c_t$) carries information across many time steps without being completely overwritten.
  - Forget gate ($f_t$) prevents memory overflow, ensuring old, unnecessary information is removed.
  - Input ($i_t$) and output gates ($o_t$) regulate memory updates, making training more stable.

- **Update Cell State:** long-term information

$$c_t = f_t . c_{t-1} + i_t . \tilde{c}_t$$

Element-wise products

Previous Cell state

candidate memory content $\tilde{c}_t = tanh(W_c[h_{t-1}, x_t] + b_c)$

Forget gate: Decides how much of the previous cell state should be **forgotten**

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

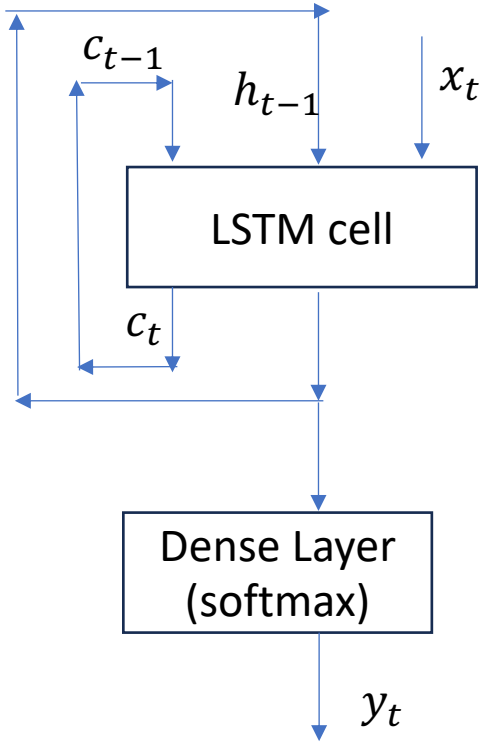
Input gate: Decides how much new information should be **added** to the cell state.

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

- **Update Hidden State:** short term information $h_t = o_t . \tanh(c_t)$

Determines which portion of the cell state contributes to the hidden state.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$




- The **probability distribution** over all possible outputs (e.g., words in vocabulary for text generation).

$$y_t = sofmax(W_y h_t + b_y)$$

# ➢ Stacked LSTMs

A **Stacked LSTM** consists of **multiple LSTM layers** where the **hidden state** $h_t$ from one LSTM layer is used as the **input** to the next LSTM layer. This allows the model to learn more complex representations.

2-Layer Stacked LSTM:

$$h_t^{(1)}, c_t^{(1)} = LSTM^{(1)}(\boldsymbol{x_t}, h_{t-1}^{(1)}, c_{t-1}^{(1)})$$

$$h_t^{(2)}, c_t^{(2)} = LSTM^{(2)}(\boldsymbol{h_t^{(1)}}, h_{t-1}^{(2)}, c_{t-1}^{(2)})$$

$$y_t = sofmax(W_y \boldsymbol{h_t^{(2)}} + b_y)$$

# ➢ Gated Recurrent Unit (GRU)

GRUs are similar to LSTMs and are designed to solve the **vanishing gradient problem** while being computationally **more efficient** than LSTMs (good for real-time application).

**How is GRU Different from LSTM?**

- The forget and input gates are replaced by reset and update gates.
- There is no cell state and output gates. Instead, it **directly updates the hidden state**
- GRUs are computationally more efficient than LSTMs due to having fewer parameters

➢ **GRU has only two gates** (LSTM has three including Forget, Input, Output):

- **Reset Gate** ($r_t$): Controls how much past information to forget. The values are between 0 and 1.
- **Update Gate** ($z_t$ ) : Decides how much past information to keep and how much new information to add.

$$h_t = (1 - z_t).h_{t-1} + z_t.\tilde{h}_t$$

Update Gate

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z)$$

Candidate Hidden State

$$\tilde{h}_t = tanh(W_h[r_t.h_{t-1}, x_t] + b_h)$$

Reset Gate $r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$

$$y_t = sofmax(W_y h_t + b_y)$$

# Transformers


- Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin, “Attention is all you need”, Advances in neural information processing systems 30, 2017, https://arxiv.org/pdf/1706.03762
- Radford, Alec, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. "Improving language understanding by generative pre-training." 2018, https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf
- J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” Proceedings of NAACL-HLT, 2019. https://arxiv.org/abs/1810.04805

# ➢ Transformers

"Attention is All You Need", Google Brain, 2017

For sequential modeling, we don't need complex recurrent or convolutional architecture, but instead only rely on attention mechanism.

- Attention Mechanism
- Transformer Block Structure
- Full Transformer Model Architecture
- Transformer Model Training
- Transformer Inference

## ➢ Attention Mechanism

**Attention** is a mechanism that allows a model to focus on specific parts of an input sequence when processing data. It computes a weighted sum of input values based on their relevance. **Self-attention** determines relationships of words within the same sequence.

- **Self-Attention:**
  - Captures relationships between words regardless of distance.
  - Computes attention scores to determine the importance of words in a sentence.

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

Input matrix: $X \in R^{n\times d_e}$ → **Learnable Weights:** $W_Q \in R^{d_e\times d_k}$, $W_K \in R^{d_e\times d_k}$, $W_e \in R^{d_e\times d_v}$ → **Query:** $Q \in R^{n\times d_k}$, **Key:** $K \in R^{n\times d_k}$, **Value:** $V \in R^{n\times d_v}$ → **Attention Weights:** $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) \in R^{n\times n}$ → $Z = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \in R^{n\times d_v}$

Usually $d_v = d_e$

Each token's embedding representation is updated based on its relationships with other tokens.

Initial embeddings $X \in R^{n\times d_e} \rightarrow Z \in R^{n\times d_v}$ Updated embeddings

- Input sequence consists of $n$ tokens, where each token is represented as an embedding vector of size $d_e$
- **Input matrix**: $X \in R^{n\times d_e}$
- **Learnable Weight** matrices: $W_Q, W_K \in R^{d_e\times d_k}$ and $W_V \in R^{d_e\times d_v}$
- **Query** matirx: $Q \in R^{n\times d_k} = XW_Q$
- **Key** matrix : $K \in R^{n\times d_k} = XW_k$
- **Value** matrix: $V \in R^{n\times d_v} = XW_V$
- $d_k$: dimension of each query/key vector and $d_v$: dimension of each value vector, while $d_e$ is the dimension of embedding vector of each token.
- **Raw Attention Scores:** $QK^T \in R^{n\times n}$ pair-wise attention score for all input tokens. Each value in $QK^T$ represents the raw relevance score of how much a token (Query row) should give to another token (key column), Higher values → Stronger relevance or focus between two tokens.
- **Scaled Attention Scores:** $\frac{QK^T}{\sqrt{d_k}} \in R^{n\times n}$
- **Attention Weights (normalized Scores):** $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) \in R^{n\times n}$ , The softmax normalizes attention scores across each row.
- **Attention output:** Context Matrix $Z = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \in R^{n\times d_v}$ weighted sum of $V$ weighted by the attention scores.

# ➢ Interpretation

## ➢ Query, Key, Value

- **Query (Q):** Represents **what a token is looking for** in the input sequence.
- **Key (K):** Represents the **encoded content of each token**, used for comparison.
- **Value (V):** Represents the **actual content** of each token that is passed to the next layer.

➢ **Analogy: Library Book Search**

Imagine you're in a library looking for a book:
• **Query (Q)** → What you are searching for (your question).
• **Key (K)** → The labels/tags in the library catalog (the system's index for books).
• **Value (V)** → The actual content of the book you find.

**1.** You have a question **(Query, Q)**:
- Let's say you're searching for "Quantum Mechanics".
- Your question represents the Query (Q).

**2.** The librarian searches in the catalog **(Key, K)**:
- The library system contains many Keys (K) that describe different books (e.g., "Quantum Mechanics," "Machine Learning," etc.).
- Each Key (K) helps match a query to the right book.

**3.** The librarian retrieves the book **(Value, V)**:
- Once a match is found (best matching Key), the librarian gives you the actual book.
- The book's content represents the Value (V).

➢ After training, the model learns original embedding vectors (word/token representations) and learnable weight matrices that result in **Attention scores** that correctly capture token relationships in various sentences.

## ➢ Raw Attention Scores $QK^T \in R^{n \times n}$

- Computes a similarity measure between each query and every key.
- Each value in $QK^T$ represents the **raw relevance score** of how much attention a token **(Query row)** should give to another token **(Key column)**.
- **Higher values** → Stronger relevance or focus between two tokens.

## ➢ Dot Product ($QK^T$) Interpretation

- If Q and K are **aligned** in the same direction → $QK^T$ **is positive (high similarity)**.
- If Q and K are **orthogonal (unrelated)** → $QK^T$ **is near zero.**
- If Q and K are **opposite in direction** → $QK^T$ **is negative (low similarity).**

Example:
Sequence: "The cat sat on the mat"

$QK^T$ :

| Tokens | The | cat | sat | on | the | mat |
|---|---|---|---|---|---|---|
| **The** | 1.2 | 0.8 | 0.3 | -0.1 | 1.0 | 0.4 |
| **cat** | 0.9 | 1.5 | 0.7 | 0.2 | 0.8 | 0.3 |
| **sat** | 0.2 | 0.5 | 1.8 | 1.0 | 0.6 | 0.9 |
| **on** | 0.1 | 0.3 | 1.2 | 2.0 | 0.5 | 0.8 |
| **the** | 0.7 | 0.6 | 0.4 | 0.2 | 1.3 | 1.1 |
| **mat** | 0.5 | 0.4 | 0.8 | 1.2 | 1.0 | 1.6 |

**row: "sat", column: "mat"**
Value = 0.9
This means "sat" attends to "mat" with moderate importance. This makes sense because "sat" relates to "mat" as the object of the action.

**Value (sat → on) = 1.0** (higher than sat → mat)
"sat" has a stronger relationship with "on" because "sat on" is a direct phrase.

## ➢ Causal Masking

Since GPT and other autoregressive models generate text **one token at a time**, they must ensure that each token:

- Can **attend to past tokens** (leftward attention).
- **Cannot attend to future tokens** (prevents cheating).

**Without causal masking**, the model **would see future words** while training, making the learning process unrealistic for real-world text generation.

➢ The attention score matrix $QK^T$ has a shape of $(n \times n)$, where n is the sequence length.

A **triangular mask** is applied to set all future token attention scores to **negative infinity (−∞)** before softmax.

This forces the softmax function to **assign zero probability** to future tokens.

$$QK^T = \begin{bmatrix} ✓ & ✗ & ✗ & ✗ & ✗ \\ ✓ & ✓ & ✗ & ✗ & ✗ \\ ✓ & ✓ & ✓ & ✗ & ✗ \\ ✓ & ✓ & ✓ & ✓ & ✗ \\ ✓ & ✓ & ✓ & ✓ & ✓ \end{bmatrix}$$

Where:

- ✓ **(Allowed)**: The current token can attend to itself and past tokens.
- ✗ **(Blocked)**: The current token **cannot** attend to future tokens.

➢ The **masked attention** scores become: $QK^T + M$

Where M is a **lower triangular matrix.** It has

- **0 in the lower triangular part** (tokens can attend to themselves and previous tokens).
- **−∞ in the upper triangular part** (future tokens are masked).

- After applying **softmax**, all **−∞ values become zero probability**, preventing future tokens from being attended to.

## ➢ Multi-Head Attention

Multi-head attention is an extension of the **self-attention** mechanism in Transformers that improves the model's ability to **capture different types of relationships** between words in a sequence.

- This allows the model to focus on multiple relationships simultaneously, e.g., One head may focus on grammatical structure, another head may focus on semantic meaning and another head may focus on long-range dependencies.

➢ Instead of applying **a single self-attention operation**, **multi-head attention (MHA)**:

- **Splits the input embeddings** into multiple smaller subspaces, e.g., $d_h = \frac{d_e}{h}$ where h is number of the attention heads and $d_e$ is the size the original embedding vectors
- **Applies self-attention separately** to each subspace (each called an "attention head").
- **Combines the outputs** from all heads before passing them to the next layer.

where:

$$MultiHead(Q, K, V) = Concat(head_1, head_2, \dots, head_h)W^o$$

- Each **head** performs independent self-attention.
- The outputs are **concatenated** and **transformed** using a weight matrix $W^o$.

**1.** Instead of using a **single Q,K, V** for self-attention, we create **multiple versions**: $Q_i \in R^{n\times d_h} = XW_Q^i$ , $K_i \in R^{n\times d_h} = XW_k^i$, $V_i \in R^{n\times d_h} = XW_V^i$ $\quad i = 1, \dots, h$

where, $h$ is the number of heads and $d_h = \frac{d_e}{h}$, and $W_Q^i$, $W_k^i$ , $W_V^i$ are learnable weight matrices.

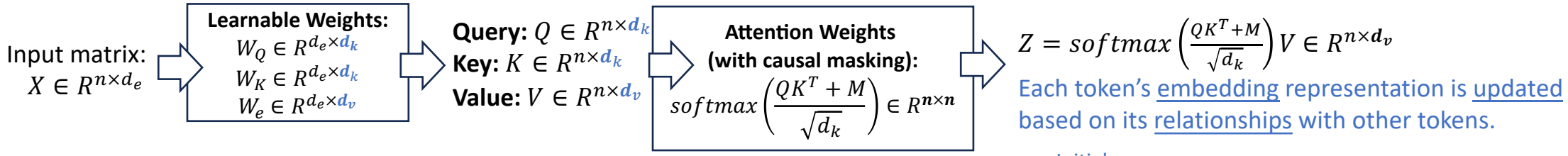
**2.** Attention for Each Head: $Attenstion_i(Q, K, V) = softmax\left(\frac{Q_i K_i^T}{\sqrt{d_h}}\right) V_i \in R^{n\times d_h}, i = 1, \dots, h$

**3.** Concatenate Heads: $Concat(\underbrace{head_1}_{\in R^{n\times d_h}}, head_2, \dots, \underbrace{head_h}_{\in R^{n\times d_h}}) \in R^{n\times(h*d_h)} = R^{n\times d_e}$

$$MultiHead(Q, K, V) = Concat(head_1, head_2, \dots, head_h)\, W^o \in R^{n\times d_e}$$

where $W^o$ is a learned weight matrix

## Self Attention:



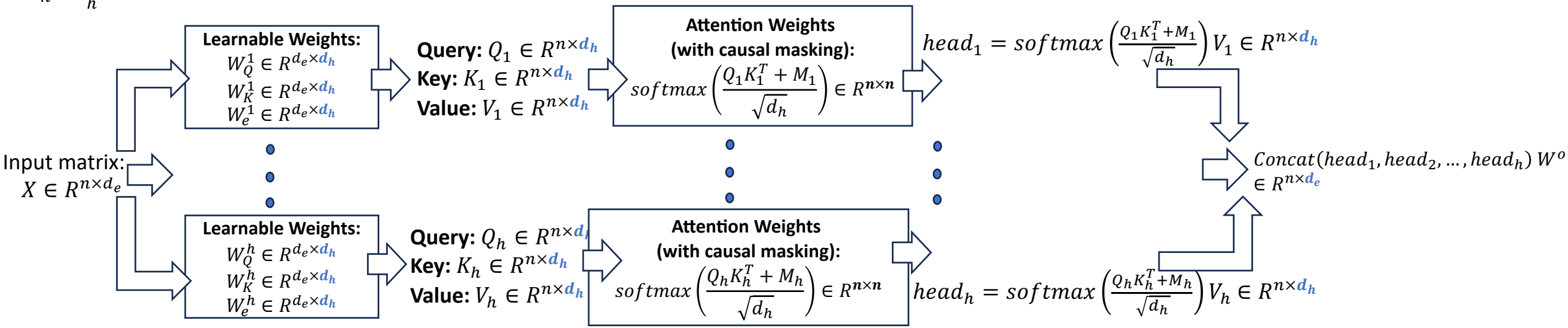

Each token's embedding representation is updated based on its relationships with other tokens.

Initial embeddings $X \in R^{n \times d_e} \rightarrow Z \in R^{n \times d_v}$ Updated embeddings

## Multi-Head Attention:

$h$: number of heads

$d_h = \frac{d_e}{h}$: dimension of each attention head




Initial embeddings $X \in R^{n \times d_e} \rightarrow Z \in R^{n \times d_e}$ Updated embeddings

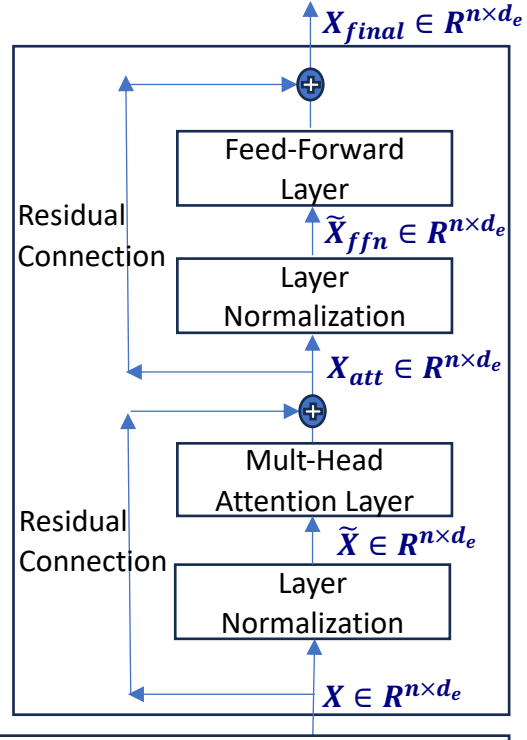
## ➢ Transformer Block Structure

A single Transformer block consists of:

1. **Input** $X \in R^{n \times d_e}$ where $n$ = Sequence length (number of tokens in input) and $d_e$ = Embedding dimension (hidden size).
2. **Layer Norm:** $\tilde{X}_{att} = LayerNorm(X)$
3. **Multi-Head Self-Attention** layer to compute attention across tokens (Context learning): $MultiHead(\tilde{X}_{att})$
4. **Add Residual Connections**: $X_{att} = X + MultiHead(\tilde{X}_{att})$
5. **Layer Norm:** $\tilde{X}_{ffn} = LayerNorm(X_{att})$
6. **Feedforward Neural Network (FFN)** (to introduce non-linearity).

   Apply a two-layer feedforward network with **ReLU/GELU** activation: $FFN(\tilde{X}_{ffn}) = \max(0, \tilde{X}_{ffn}W_1 + b_1)W_2 + b_2$

   where, feedforward weight $W_1$ is a projection matrix to a larger dimension, and $W_2$ is a projection back to $d_e$, and $b_1$ and $b_2$ are biases.
7. **Add Residual Connections**: $X_{final} = X_{att} + FFN(\tilde{X}_{ffn})$




$$X_{att} = X + MultiHead(LayerNorm(X))$$
$$X_{final} = X_{att} + FFN(LayerNorm(X_{att}))$$

(Pre-LayerNorm) **Transformer Block**

- LayerNorm is applied BEFORE both Multi-Head Attention and Feed-Forward layers
- Residual connections are added AFTER each sub-layer
- More stable training for deep networks. Gradient-friendly due to normalization before each operation.

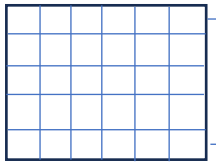
➢ Note that The input and output of a **Transformer Block** have the same shape $R^{n \times d_e}$

➢ **Layer Normalization:** Normalizes activations across the feature dimension (i.e., embedding dimension $d_e$), ensuring that the mean is 0 and variance is 1.

$$LayerNorm(X) = \frac{X - \mu}{\sigma}\gamma + \beta \quad \text{where} \quad \gamma_i, \beta_i \; i = 1, \dots d_e \text{ are Learnable scaling and shifting parameters.}$$

- Each token's embedding vector is normalized independently using its own mean and variance.

$X \in R^{n \times d_e}$



Embedding vector of token 1, normalize the vector using its mean and variance → [Normalized embedding vector] $\odot[\gamma_{1:d_e}] + \beta_{1:d_e}$

Embedding vector of token n, normalize the vector using its mean and variance → [Normalized embedding vector] $\odot[\gamma_{1:d_e}] + \beta_{1:d_e}$

$\odot$ element-wise multiplication

# ➢ Full Transformer Model Architecture

- A **Transformer model** consists of multiple **stacked Transformer blocks** with an embedding layer at the input and a final output layer.

**1. Token + Positional Embeddings**

- Token embedding: Convert input tokens into dense vector representations (**learned embedding vector**).
- Positional Embedding: Add position information to embeddings. Since Transformers have **no recurrence**, **positional encoding** is added to maintain word order.

**2. Stacked Transformer Blocks:** $X \in R^{n \times d_e} \rightarrow X_{final} \in R^{n \times d_e}$

**3. Output Layer:**

- Convert processed embeddings into predictions.
- Fully connected layer followed by softmax to generate the distribution over token space.
- Softmax is applied separately to each row of the logits matrix, converting them into probability distributions over the vocabulary.

$Logits \in R^{n \times V} = X_{final} W_{out} + b$

$X_{final} \in R^{n \times d_e}$

$W_{out} \in R^{d_e \times V}$ (independent of input sequence size)

$b \in R^{V}$ the same bias vector is added to each row of matrix,

e.g., $X_{final} W_{out}(i,:) + b, i = 1, \dots n$

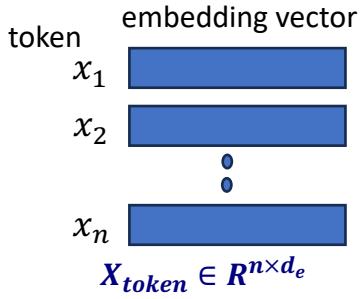
$V$: Vocabulary size

$\rightarrow Softmax(Logits(i,:)), i = 1, \dots n$

## ➢ Token + Positional Embeddings

Turns integer into a dense vector

$x_i \rightarrow embedding\ vector$




**Token Embedding:**

- Each input word (token) is mapped to a dense vector representation.
- $X_{token} \in R^{n \times d_e}$ where $n$ = Sequence length (number of tokens in input) and $d_e$ = Embedding dimension (hidden size).
  These embeddings are learned during training.

**Positional Encoding (PE):**

- Since Transformers **do not have recurrence** (like RNNs), they need **Positional Encodings** to inject **word order** information into the model.

$X_{input} \in R^{n \times d_e} = X_{token} + X_{position}$

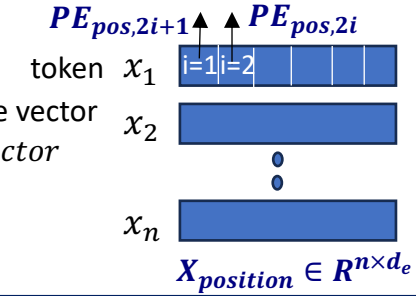
- It provides **unique positional values** for each token in the sequence.
- **Sinusoidal Positional Encoding:** $PE_{pos,2i} = sin(\frac{pos}{10000^{\frac{2i}{d_e}}})$ even indices, $PE_{pos,2i+1} = cos\left(\frac{pos}{10000^{\frac{2i}{d_e}}}\right), i = 1, \dots, d_e$ odd indices

**pos** is an integer the token's position in the sequence, $i$ is the feature index in embedding vector ranging from 1 to $d_e$

- Positional Encoding is Fixed (not learnable.)

Turns integer into a dense vector

$x_i \rightarrow pos\ embedding\ vector$



# ➢ Transformer Model Training

In Transformer training, we process the **entire sequence at once** and compute $n$ **probability distributions in parallel** (one for each token position).

- Transformers are designed to **process sequences in parallel**.
- The model predicts **all next tokens at once**, unlike RNNs that predict one token at a time.

**Input:** Full sequence $X \in R^{n \times d_e}$
**Output:** $n$ probability distributions over the vocabulary
**Loss:** Compute $n$ losses, average them, then update parameters

1. **Tokenization/Embedding/Positional Encoding:** Input text → Tokens → IDs → Embeddings + Positional Encoding : $X \in R^{n \times d_e}$
2. **Forward Pass:**
   $X \in R^{n \times d_e}$ (Input embeddings + Positional encoding) → Stacked Transformer Blocks (Multi-Head Attention/LayerNorm/Feedforward Network) $\in R^{n \times d_e}$ → Output Layer (computes logits $\in R^{n \times V}$ )

**3. Compute output Probabilities ($n$ probability distributions):** The model predicts all next tokens at once instead of generating them sequentially like an RNN. Softmax is applied to the logits to generate $n$ n probability distributions (one for each token position, output probabilities $\in R^{n \times V}$

**4. Compute Losses:**

- The predicted probability distribution is compared with the true label.
- The loss function (Cross-Entropy Loss) measures how far the prediction is from the correct token.

$$L = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

where:

- $n$ = *Number of tokens in the sentence.*
- $y_i$ **true probability (one-hot)**.
- $\hat{y}_i$ is the **predicted probability from softmax**.

**4. Backpropagation & Gradient Descent:**

- **Compute gradients** of the loss with respect to model parameters.
- Optimizer (e.g., Adam, AdamW) updates weights of:
  Token embeddings, Multi-Head Attention parameters ($W_Q, W_K, W_V$ for all heads and transformer blocks), Feedforward layers

**5.** Repeat this process for all training data across multiple epochs.

➢ Everything is done in parallel, making training efficient!

## ➢ Example:

**Training data:** "The cat sat on the mat."

| Token | Token ID | Embedding ( de=4) |
|---|---|---|
| "The" | 32 | [0.2, -0.1, 0.5, 0.3] |
| "cat" | 120 | [0.4, 0.7, -0.3, 0.8] |
| "sat" | 45 | [0.1, 0.6, -0.5, 0.9] |
| "on" | 78 | [0.5, -0.2, 0.7, 0.3] |
| "the" | 32 | [0.2, -0.1, 0.5, 0.3] |
| "mat" | 210 | [-0.4, 0.9, 0.1, -0.6] |
| "." | 10 | [0.6, 0.3, -0.1, 0.2] |

$$X = X_{token} + X_{position}$$

$X_{token} \in R^{n=7 \times d_e=4}$

| | | | |
|---|---|---|---|
| 0.2 | −0.1 | 0.5 | 0.3 |
| 0.4 | 0.7 | −0.3 | 0.8 |
| 0.1 | 0.6 | −0.5 | 0.9 |
| 0.5 | −0.2 | 0.7 | 0.3 |
| 0.2 | −0.1 | 0.5 | 0.3 |
| −0.4 | 0.9 | 0.1 | −0.6 |
| 0.6 | 0.3 | −0.1 | 0.2 |

$X_{position} \in R^{n=7 \times d_e=4}$

| | | | |
|---|---|---|---|
| 0.0 | 1.0 | 0.0 | 1.0 |
| 0.841 | 0.54 | 0.009 | 0.999 |
| 0.909 | 0.41 | 0.018 | 0.999 |
| 0.951 | 0.3 | 0.027 | 0.999 |
| 0.978 | 0.2 | 0.036 | 0.999 |
| 0.993 | 0.1 | 0.045 | 0.999 |
| 1.0 | 0.0 | 0.054 | 0.999 |

**(Masked)Attention weights** $\in R^{n=7 \times n=7}$

| | "The" | "cat" | "sat" | "on" | "the" | "mat" | "." |
|---|---|---|---|---|---|---|---|
| "The" | 1.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| "cat" | 0.462 | 0.537 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| "sat" | 0.250 | 0.337 | 0.412 | 0.000 | 0.000 | 0.000 | 0.000 |
| "on" | 0.164 | 0.221 | 0.284 | 0.330 | 0.000 | 0.000 | 0.000 |
| "the" | 0.118 | 0.160 | 0.205 | 0.251 | 0.264 | 0.000 | 0.000 |
| "mat" | 0.095 | 0.122 | 0.156 | 0.191 | 0.211 | 0.222 | 0.000 |
| "." | 0.079 | 0.097 | 0.118 | 0.152 | 0.168 | 0.186 | 0.196 |

**Computed Probabilities** $\in R^{n=7 \times V}$

| Input Token | "cat" (Target) | "sat" | "on" | "the" | "mat" | "." | Other Tokens (Rest of V) |
|---|---|---|---|---|---|---|---|
| "The" | **0.98** | 0.01 | 0.002 | 0.003 | 0.002 | 0.001 | 0.002 |
| "cat" | 0.002 | **0.95** | 0.02 | 0.01 | 0.01 | 0.002 | 0.006 |
| "sat" | 0.001 | 0.02 | **0.97** | 0.005 | 0.002 | 0.002 | 0.0005 |
| "on" | 0.001 | 0.005 | 0.002 | **0.99** | 0.001 | 0.001 | 0.0002 |
| "the" | 0.001 | 0.002 | 0.002 | 0.02 | **0.94** | 0.03 | 0.005 |
| "mat" | 0.0005 | 0.002 | 0.003 | 0.01 | 0.03 | **0.99** | 0.0005 |

**Loss function**

- **"The" → True next word: "cat"** → $Loss_1 = -1 \times \log$ (0.98)
- **"cat" → True next word: "sat"** → $Loss_2 = -1 \times \log$ (0.95)
- **"sat" → True next word: "on"** → $Loss_3 = -1 \times \log$ (0.97)
- **"on" → True next word: "the"** → $Loss_4 = -1 \times \log$ (0.99)
- **"the" → True next word: "mat"** → $Loss_5 = -1 \times \log$ (0.94)
- **"mat" → True next word: "."** → $Loss_6 = -1 \times \log$ (0.99)

$$Loss = -\sum_{i=1}^{n=6} y_i \log(\hat{y}_i) \quad = -\sum_{i=1}^{n=6} Loss_i$$

# ➢ Transformer Inference

**(GPT) Inference Process**

- **Input Processing**
  - Convert input text into tokens.
  - Convert tokens into embeddings.
  - Add positional encoding.
- **Autoregressive Decoding**
  - The first token (e.g., <BOS> or a given prompt) is passed through the Transformer.
  - The model predicts the next token using softmax probabilities.
  - The predicted token is appended to the input sequence.
  - The updated sequence is fed into the model to predict the next token.
  - Steps are repeated until a stopping condition is met (e.g., max length, end token <EOS>).

**(GPT) Inference Strategies**

- **Greedy Decoding**
  - Selects the most probable next token at each step.
  - Fast but may produce repetitive or suboptimal text.
- **Beam Search**
  - Instead of choosing the single most likely token at each step (as in greedy decoding), beam search keeps track of **multiple candidate sequences** (beams) simultaneously, and selects the best.
  - More diverse but computationally expensive.
- **Top-k Sampling**
  - Chooses from the top-k most probable next tokens, adding randomness.
- **Nucleus Sampling (Top-p)**
  - Selects from the smallest set of tokens whose cumulative probability exceeds $p$
  - More flexible and avoids low-probability words.

**Temperature parameter** to control the randomness of the sampling process
Temperature 0: choose next token with highest probability (deterministic)
Temperature 1: next token is chosen with the probability outputted by the model (random)

# ➢ Transformers

**1. Parallelization & Efficiency**

- **Unlike RNNs**, which process sequentially, Transformers **process entire sequences at once** using the self-attention mechanism.
- This allows for **faster training** on GPUs/TPUs.

**2. Long-Range Dependencies**

- RNNs and LSTMs struggle with long-term dependencies due to vanishing gradients.
- Transformers use **self-attention**, allowing them to capture dependencies between tokens that are far apart in the sequence.

**3. Contextual Understanding**

- Unlike traditional word embeddings, Transformers generate **context-dependent embeddings** (e.g., "bank" in "river bank" vs. "bank account" has different meanings).
- This improves **machine translation, question answering, and summarization**

**4. Versatility & Transfer Learning**

- **Pre-trained Transformers** (**GPT, BERT, T5**) can be fine-tuned for **various tasks** (e.g., chatbots, code generation, medical diagnosis).
- **One model → many applications**.

# ➢ Different Transformer Models

- Encoder-only models (e.g., BERT) → Best for understanding input (classification, embeddings, question answering).
- Decoder-only models (e.g., GPT) → Best for generating text (chatbots, text generation).
- Encoder-Decoder (Seq2Seq) models (e.g., T5, BART) → Best for tasks like translation and summarization.

| Feature | **Encoder-Only (BERT)** | **Decoder-Only (GPT)** | **Encoder-Decoder (T5, BART)** |
|---|---|---|---|
| **Best for** | Understanding | Generation | Sequence-to-sequence |
| **Attention Type** | Full self-attention | Masked self-attention | Encoder: Full, Decoder: Masked |
| **Examples** | BERT, RoBERTa | GPT-3, GPT-4, ChatGPT | T5, BART, Pegasus |
| **Use Cases** | Text classification, embeddings | Chatbots, code generation | Translation, summarization |

# ➢ Transformer Vs RNN

- **Architecture & Processing:**

| Feature | **Transformer** | **RNN (including LSTM, GRU)** |
|---|---|---|
| **Architecture** | Self-attention mechanism | Sequential processing with recurrent connections |
| **Parallelization** | **Highly parallelizable** (processes all tokens at once) | **Not parallelizable** (processes tokens one at a time) |
| **Processing Speed** | **Faster** due to parallelism | **Slower** since tokens are processed sequentially |
| **Scalability** | Efficient on large datasets | Struggles with long sequences |

- **Inference Speed & Efficiency:**

| **Feature** | **Transformer (GPT, BERT)** | **RNN (LSTM, GRU)** |
|---|---|---|
| **Processing Type** | **Parallel (all tokens at once)** | **Sequential (one token at a time)** |
| **Inference Speed** | **Fast** on GPUs/TPUs | **Slow** due to recursion |
| **Memory Usage** | **High (stores attention scores)** | **Lower (only stores hidden states)** |
| **Best for Long Sequences?** | **Yes** (handles long-range dependencies well) | **No** (loses context over long sequences) |
| **Best for Real-Time Applications?** | High inference latency and memory usage limit real-time deployment. | Yes (low-latency, smaller footprint) |

- **Handling Long-Range Dependencies:**

| Feature | **Transformer** | **RNN (including LSTM, GRU)** |
|---|---|---|
| **Memory of past inputs** | Uses **self-attention**, allowing direct connections between distant words | Struggles with **vanishing gradient problem** (hard to retain long-term dependencies) |
| **Context Awareness** | **Captures entire context at once** | Needs multiple time steps to understand long-term relationships |

- **Computational Complexity:**

| Feature | **Transformer** | **RNN (including LSTM, GRU)** |
|---|---|---|
| **Time Complexity** | **O($n^2$) in standard self-attention**, but optimized versions (Linformer, Longformer) reduce this | **O(n)** (scales linearly with sequence length) |
| **Memory Usage** | **High** (stores all pairwise attention scores) | **Lower** (only stores hidden states) |

- **Training Efficiency:**

| Feature | **Transformer** | **RNN (including LSTM, GRU)** |
|---|---|---|
| **Training Speed** | **Faster** (leverages parallelism and batch processing) | **Slower** (sequential updates lead to longer training times) |
| **Gradient Stability** | **More stable gradients** (avoids vanishing gradient issue) | **Prone to vanishing gradient problem** in deep RNNs |

- **Interpretability & Flexibility:**

| Feature | **Transformer** | **RNN (including LSTM, GRU)** |
|---|---|---|
| **Interpretability** | Hard to interpret ("black box") | More interpretable (hidden states carry meaning) |
| **Flexibility** | Can handle multimodal tasks (text, vision, speech) | Mostly used for sequential text/audio processing |

## ➢ Cross Attention Mechanism

- **Query (Q)** comes from the latent tokens (latent patches in ViTs, DiTs[1]):
  - $Q \in R^{n \times d_k} = X_Q W_Q$
  - Input matrix: $X_Q \in R^{n \times d_e}$(latent space).

- **Key (K) and Value (V)** come from the conditioning embeddings[2] (e.g., text embeddings):
  - $K \in R^{m \times d_k} = X_{KV} W_K$ and $V \in R^{m \times d_v} = X_{KV} W_V$
  - Input matrix: $X_{KV} \in R^{m \times d_e}$ (text encoder embeddings).

- **Cross Attention Formula:** $\text{Cross}Attenstion(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) V \in R^{n \times d_e}$

**Example: Cross-attention in text-to-image models**

Latent tokens (**Q**) "look at" the text embeddings (**K, V**) to generate relevant images. Cross-attention allows latent tokens to selectively attend to and incorporate **semantic information** from the conditioning input (e.g., text). Thus, the model learns to condition the generation process on text descriptions.

1 : See ViTs and DiTs sections.
2 : See Multimodal Models section.

# ➢ Transformer-based Gen-AI models

- To combine transformer models with other generative AI models, we first need to "tokenize each input modality appropriately.

- **Text**: Needs to be tokenized into subwords or word-pieces.
- **Images**: Split into patches and flattened into sequences.
- **Audio**: Tokenized using spectrograms or by converting spectrograms into discrete tokens (e.g., via VQ-VAE or HuBERT).
- **Multimodal**: Each modality gets preprocessed/tokenized differently, then unified in a shared embedding space.

# Vision Transformers (ViTs)

- Dosovitskiy, Alexey, et al. “An image is worth 16x16 words: Transformers for image recognition at scale”, In ICLR 2021. https://arxiv.org/pdf/2010.11929

# ➢ Vision Transformers (ViTs)

**Key Characteristics of Vision Transformers (ViT):**

**1. Fully Transformer-Based**

- Replaces CNNs entirely with Transformer blocks.
- Uses attention across patches rather than convolutions.

**2. Patch-Based Tokenization**

- The image is split into non-overlapping patches, each treated as a “word token” in NLP.

**3. Positional Embeddings**

- Since Transformers are permutation-invariant (i.e., they don’t inherently preserve order), ViT adds positional embeddings to retain spatial layout.

**4. Scales Well with Data**

- Performs better than CNNs when trained on large datasets (e.g., JFT-300M).

# ➢ Vision Transformers (ViTs)

**1) Image Tokenization**

• Input image $x \in R^{H\times W\times C}$ (e.g., 224×224×3) is split into fixed-size non-overlapping patches $P\times P$(e.g., 16×16).

• Each patch ($\in R^{P\times P\times C}$) is flattened into a vector ($R^{P^2.C}$ ) and linearly projected to a D-dimensional embedding space.

$$\text{Patches}: x = [x^{(1)}, x^{(2)}, \dots, x^{(N)}],\ \text{each flattened patch (vector)}\ x^{(i)} \in R^{P^2.C}$$

$$\text{Tokens: } z_0 = \left[x_{class}, x^{(1)}W_e, \dots, x^{(N)}W_e\right] + E_{pos} \in R^{(N+1)\times D}$$

- $W_e \in R^{(P^2.C)\times D}$: learnable weight matrix that maps each patch vector to a Transformer token embedding of size D. Hence, projected patch token is: $x^{(i)}W_e \in R^D,\quad x^{(i)} \in R^{P^2.C} \rightarrow x^{(i)}W_e \in R^D$
- $x_{class} \in R^D$: learnable token with the same dimension as the patch tokens.
- Concatenating all tokens : $z_0 = [x_{class}, x^{(1)}W_e, \dots, x^{(N)}W_e] \in R^{(N+1)\times D}$
- $E_{pos} \in R^{(N+1)\times D}$: positional embedding, same shape as $z_0$

•$H\times W$ : image resolution (e.g., 224 × 224)
•$C$ : number of image channels (e.g., 3 for RGB)
•$P\times P$: patch size (e.g., 16 × 16)
•$N = \frac{H.W}{P^2}$ : number of patches
•D: Transformer embedding dimension (e.g., 768 or 1024)

**2) Transformer Encoder**

• The sequence of patch tokens is processed by standard **Transformer encoder blocks**:
  • Multi-head Self-Attention (MSA)
  • MLP blocks with LayerNorm (LN) and residuals

$$z'_l = MSA\left(LN(z_{l-1})\right) + z_{l-1}$$
$$z_l = MLP\left(LN(z'_l)\right) + z'_l$$

  • This is repeated for **L layers** (e.g., 12 or 24 layers): $z_0 \rightarrow z_1 \rightarrow \cdots \rightarrow z_L$

**3) Classification Head**

- After Transformer encoding, the final output of the **[class] token** is passed to a **linear classifier**: $y = softmax\left(W_{head}.z_L^{(class)}\right) \in R^K$ (A vector of class probabilities after softmax.

where, $z_L^{(class)} \in R^D$ is the first row (the class token embedding) of the final output $z_L \in R^{(N+1)\times D}$

•The class token is designed to aggregate global information from all other tokens via attention. After the final layer, it is used as the representation of the entire image for classification.

# Diffusion Transformers (DiTs)

- Peebles W, Xie S. “Scalable diffusion models with transformers”, In Proceedings of the IEEE/CVF international conference on computer vision 2023 (pp. 4195-4205). https://arxiv.org/abs/2212.09748

# ➢ Diffusion Transformers (DiTs)

- Diffusion-based generative models that utilize a pure **Transformer architecture**—instead of the **conventional U-Net**—to predict noise or perform denoising in diffusion models.

- Unlike typical diffusion models (e.g., Stable Diffusion, which uses a convolutional U-Net with self-attention), **DiTs** entirely replace convolutional layers with Transformer blocks, resulting in a fully attention-based noise prediction model.

➢ **Key Characteristics of Diffusion Transformers (DiTs):**

• **Fully Transformer-Based:** Instead of a convolutional or hybrid architecture, DiT uses pure Transformer blocks (self-attention + MLP layers) for diffusion-based denoising.

• **Tokenization of Latent Space:** The latent image[1] representation is divided into a sequence of tokens (patches). Each token is a small latent patch, similar to Vision Transformers (ViT).

• **Noise Prediction via Attention:** Noise prediction at each diffusion step is handled entirely by multi-head self-attention across latent tokens, leveraging the Transformer's powerful modeling capabilities.

• **No Convolutional Layers:** DiTs rely solely on attention mechanisms and positional embeddings for spatial information, removing the need for convolutional inductive biases.


- Peebles W, Xie S. “Scalable diffusion models with transformers”, In Proceedings of the IEEE/CVF international conference on computer vision 2023 (pp. 4195-4205). https://arxiv.org/abs/2212.09748


1: See Latent-Space Diffusion Model section in Algorithm section.

# ➢ Diffusion Transformers (DiTs)

**1) Latent Space Tokenization:**

• Latent representation $z_t$ at diffusion step $t$ is split into fixed-size tokens (similar to ViT's patches):

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

In **latent diffusion models (like Stable Diffusion or DiTs)**, the latent representation $z_t$ at timestep t is typically a **3-dimensional tensor** (not just a single flat vector**):** $\boldsymbol{z_t} \in \boldsymbol{R^{C\times H\times W}}$ where **C** is the Number of latent channels (e.g., 4 for Stable Diffusion), and **H, W**: Spatial dimensions of the latent feature map (e.g., 64×64, 32×32).

**Example:**

• **Original latent tensor**: $z_t \in R^{4\times 32\times 32}$

• **Patch size** (e.g., 8×8): each patch covers an 8×8 area in spatial dimension

• **Each patch dimension before flattening:** 4×8×8**, Flattened patch size (D)=256 (embedding vector size of a token)**

• **Number of tokens**: $\frac{32}{8}\times\frac{32}{8} = 16$

• Resulting sequence: $z_t = [z_t^{(1)}, z_t^{(2)}, \dots, z_t^{(16)}]$ with each token $z_t^{(i)} \in R^{D=256}$

**2) Positional Embeddings:**

• Add positional embeddings to tokens to encode spatial structure.

**3) Forward diffusion and Transformer-Based Noise Prediction:**

• Forward diffusion process: $\left[z_0^{(1)}, z_0^{(2)}, \dots, z_0^{(N)}\right] \to \left[z_1^{(1)}, z_1^{(2)}, \dots, z_1^{(N)}\right] \to \cdots \to \left[z_T^{(1)}, z_T^{(2)}, \dots, z_T^{(N)}\right]$ where $z_t^{(i)} = \sqrt{\bar{\boldsymbol{\alpha}}_{\boldsymbol{t}}} z_0^{(i)} + \sqrt{1-\bar{\alpha}_t}\epsilon_t^{(i)}, \epsilon_t^{(i)} \in R^D \sim N(0, I)$

• Transformer blocks (self-attention + MLP layers) predict the noise added at timestep t:

$$z_t = \left[z_t^{(1)}, z_t^{(2)}, \dots, z_t^{(N)}\right] \to \left[\epsilon_t^{(1)}, \epsilon_t^{(2)}, \dots, \epsilon_t^{(N)}\right]$$ where $\epsilon_t^{(i)} \in R^D$, $D$: embedding size of each token

This process predicts noise in the latent tokens directly, as in standard diffusion.

**4) Iterative Denoising:**

• At inference, start from noisy latent space and iteratively apply DiT Transformer to remove noise, eventually decoding back to pixel space

**5) From Latent to Image:**

After the final denoised latent, we use a decoder — typically from a pre-trained autoencoder (like VAE) — to reconstruct the full image.

# Attention-Based U-Net

- Oktay, O., Schlemper, J., Le Folgoc, L., Lee, M., Heinrich, M., Misawa, K., Mori, K., & Rueckert, D., “Attention U-Net: Learning Where to Look for the Pancreas”, arXiv preprint arXiv:1804.03999, 2018, https://arxiv.org/abs/1804.03999

# ➢ Attention-based U-Net:

Standard U-net (used in diffusion models) works well, but convolution has a local receptive field and inductive bias. To improve global reasoning (especially for text-guided generation), attention or cross-attention is injected ( e.g., Text Conditioned Image Generator, cross attention between text embedding and image latent tokens ).

- **Comparison: Standard U-Net vs. Attention-Based U-Net**

| **Component** | **Standard U-Net** | **Attention-Based U-Net** |
|---|---|---|
| **Feature Extractor** | Conv layers + ResBlocks | Same, but with optional **self-attention** layers at certain depths |
| **Long-Range Dependencies** | Limited to local regions | Attention enables **global context modeling** (image-wide and text-wide) |
| **(Text) Conditioning** | Often not used or concatenated to input channels | Uses **cross-attention** between image features and text embeddings. Uses **cross-attention blocks** throughout the U-Net (especially in the mid and upsampling paths) |

- **How Transformer/Attention Is Injected**

**1. Self-Attention in U-Net Blocks**

• Applied at specific layers (e.g., bottleneck) to capture global spatial dependencies
• Instead of just convolutions, add Multi-Head Self-Attention (MHSA) on the flattened spatial tokens

**2. Cross-Attention with Text Embeddings**

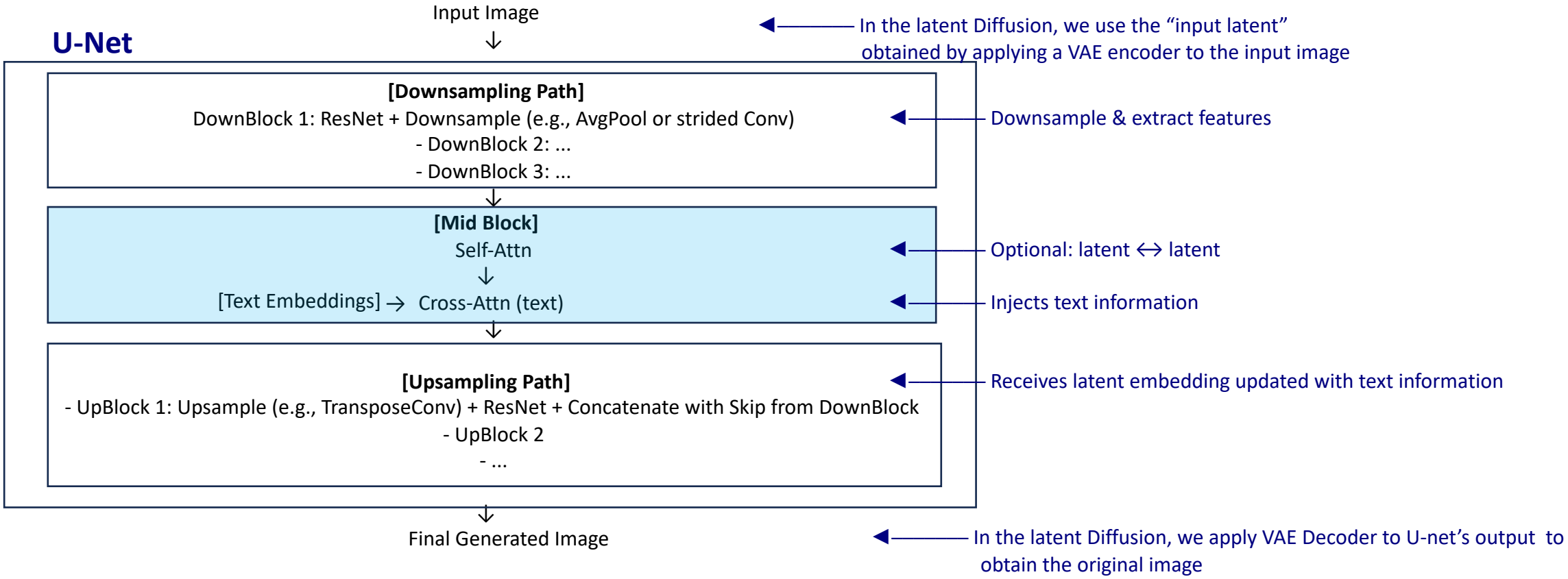
• Latent tokens (from image) serve as queries
• Text tokens (from prompt) serve as keys and values
• Implemented using standard attention[1]

- **Where Are Attention Modules Placed ?**

Usually inside or between residual blocks in the U-Net.

1: Check Cross Attention Mechanism section

**Example:**


Input Image
U-Net
In the latent Diffusion, we use the “input latent” obtained by applying a VAE encoder to the input image
[Downsampling Path]
DownBlock 1: ResNet + Downsample (e.g., AvgPool or strided Conv)
- DownBlock 2: ...
- DownBlock 3: ...
Downsample & extract features
[Mid Block]
Self-Attn
Optional: latent ↔ latent
[Text Embeddings] → Cross-Attn (text)
Injects text information
[Upsampling Path]
- UpBlock 1: Upsample (e.g., TransposeConv) + ResNet + Concatenate with Skip from DownBlock
- UpBlock 2
- ...
Receives latent embedding updated with text information
Final Generated Image
In the latent Diffusion, we apply VAE Decoder to U-net’s output to obtain the original image

## U-Net vs. DiT in Diffusion Models

| Aspect | U-Net (e.g., Stable Diffusion) | DiT (Diffusion Transformer) |
|---|---|---|
| **Core Architecture** | Convolutional U-Net with skip connections | Transformer architecture similar to Vision Transformers (ViTs) |
| **Input Format** | 3D tensor (latent: $C \times H \times W$) | 3D tensor, tokenized into 1D sequence of flattened latent patches |
| **Downsampling / Upsampling** | Explicit down and up blocks using Conv/Pooling and TransposeConv | No explicit up/downsampling; uses token embeddings across layers |
| **Feature Processing** | Convs + ResNets capture local features; attention (optional) for long-range | Global modeling via self-attention over all tokens |
| **Text Conditioning** | Cross-attention layers in mid & up path (latent ↔ text tokens) | Cross-attention layers at each transformer block (tokens ↔ text) |
| **Skip Connections** | Between down and up blocks (spatial level) | None — attention operates globally, with positionally encoded tokens preserving structure |
| **Training Target** | Predict noise $\hat{\epsilon}_t$ in latent space | Same — predict noise $\hat{\epsilon}_t$ for each latent token |
| **Spatial Awareness** | Built-in due to convolutions | Requires positional embeddings to preserve spatial structure |
| **Inference Process** | Iteratively denoise latent $z_t \rightarrow z_0$, decode with VAE | Same: denoise sequence of latent tokens $z_t \rightarrow z_0$, then decode with VAE |
| **Memory & Scale Efficiency** | Efficient for high-resolution images (due to patch-wise computation) | May require more memory due to full self-attention over large token sequences |
| **Popular Examples** | Stable Diffusion v1.x, Imagen, DALL·E 2 U-Net | DiT (Peebles & Xie), Stable Diffusion XL (partially uses DiT) |

# Multimodal Models


- Aditya Ramesh et al., "Zero-Shot Text-to-Image Generation," February 24, 2021, https://arxiv.org/abs/2102.12092.
- Aditya Ramesh et al., "Hierarchical Text-Conditional Image Generation with CLIP Latents", 2022, https://arxiv.org/abs/2204.06125.
- Alec Radford et al., "Learning Transferable Visual Models From Natural Language Supervision,", 2021, https://arxiv.org/abs/2103.00020.
- Alex Nichol et al., "GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models", 2021, https://arxiv.org/abs/2112.10741
- Chitwan Saharia et al., "Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding", 2022, https://arxiv.org/abs/2205.11487
- Foster, David. Generative deep learning. " O'Reilly Media, Inc.", 2022.

## ➢ Multimodal Models

Multimodal models involve training generative models to convert between different types of data, such as generating images from text descriptions (text-to-image), converting speech to text (speech-to-text), or generating video from a sequence of images and sound (image-and-audio to video).

Example Modalities:

- **Text-to-Image**: e.g., DALL·E generates pictures based on text prompts.
- **Speech-to-Text**: e.g., Whisper transcribes spoken language into written text.
- **Image Captioning**: e.g., CLIP-like models generate descriptive text from an image.
- **Text-to-Speech**: e.g., Tacotron synthesizes human-like speech from text.
- **Video Generation**: e.g., Generating a short video from a script or sequence of images and audio.

## ➢ Multimodal Transformer

A **Multimodal Transformer** is a model that takes inputs from **multiple modalities**—like **images + text**—and learns to process them jointly by fusing them in a shared transformer-based architecture.

**Key Components:**

- **Image Encoder:** Uses a CNN or Vision Transformer (ViT) to turn an image into a set of visual feature tokens.
- **Text Encoder:** Uses a language model (e.g., GPT) to tokenize and embed the caption text.
- **Multimodal Fusion Layer (Cross-Attention):** Combines image and text features using cross-modal attention, allowing one modality (e.g., text) to guide the other (e.g., image).
- **Output Head:** Can be a classification head (e.g., "is the caption correct for this image?") or a generation head (e.g., generate a caption).

Here's the high-level overview of training a text-to-image model:
**Text Prompt → Text Encoder → Conditioned Image Generator → Output Image**

## ➢ Common Image/Text Encoders

| Image Encoder Type | Description | Typical Output |
|---|---|---|
| **CNN-based Encoders** | Uses Convolutional Neural Networks (CNNs) to extract multi-scale spatial features from image of shape $R^{C\times H\times W}$ | Feature maps $R^{C'\times H'\times W'}$ or compressed vectors $R^{d}$ |
| **VAE Encoders** | Probabilistic encoders that map image to mean and variance for sampling latent variables which can be a vector or tensor | Mean and variance of vector/tensor |
| **Vision Transformers (ViTs)** | Split images into patches, embed them, and model global relationships using Transformer layers | Sequence of patch embeddings $R^{N_{patch}\times d_{model}}$ |
| **U-Net Encoder (in Diffusion)** | Downsampling path of a U-Net (Residual Blocks and Convolutions) that maps the input image into progressively lower-resolution feature maps. | Multi-scale feature maps $R^{C'\times H'\times W'}$ |

| Text Encoder Type | Description | Typical Output |
|---|---|---|
| **Transformer-based Encoders** | Encode text sequences using self-attention layers applied on the sequence of the text tokens | Sequence of token embeddings $R^{N_{token}\times d_{model}}$ |
| **T5 Encoder** | Text-To-Text Transfer Transformer; maps input text into a sequence of latent embeddings for generation tasks. | Sequence of embeddings $R^{N_{token}\times d_{model}}$ |
| **LSTM/GRU Encoders** | Use sequential RNNs to encode text token-by-token; historically common, now less used in GenAI. | Final hidden state vector or full sequence of hidden states |

| (Image+Text) Encoder Type | Description | Typical Output |
|---|---|---|
| **Cross-modal Models (CLIP, Flamingo)** | Dual encoders (image encoder + text encoder) that map image and text into a shared vision-language embedding space. | Aligned embedding vectors (e.g., $R^{D}$) in a joint space |

# Text Prompt → Text Encoder → Conditioned Image Generator → Output Image

## DALL·E 3

Text prompt
↓
Transformer-based text encoder (to generate the text embeddings, often similar to GPT)
↓
Diffusion model (U-Net + Classifier-Free Guidance (CFG))
↓
Iterative denoising (conditioned on text)
↓
Generated image

- The **text embeddings** are used to condition the image generation process — meaning, the model will reference the text context during each denoising step in the diffusion process.
  - Text embeddings are injected into the U-Net at each denoising step (e.g., via cross-attention or concatenation).

- **Latent Diffusion:** A U-Net-based diffusion model denoises a latent tensor (compressed image representation e.g., Input image: [B, 3, 512, 512], After VAE encoder: [B, 4, 64, 64]). This happens over many time steps.

- **Classifier-Free Guidance (CFG)** improves text-image alignment by steering the output toward the text semantics.
  - **Classifier-Free Guidance** (CFG):
    - Generates two predictions: one conditioned on the prompt and one unconditioned.
    - Combines them to **amplify alignment with the text** without needing a separate classifier.
      Linear Combination (CFG Formula) with the guidance scale.

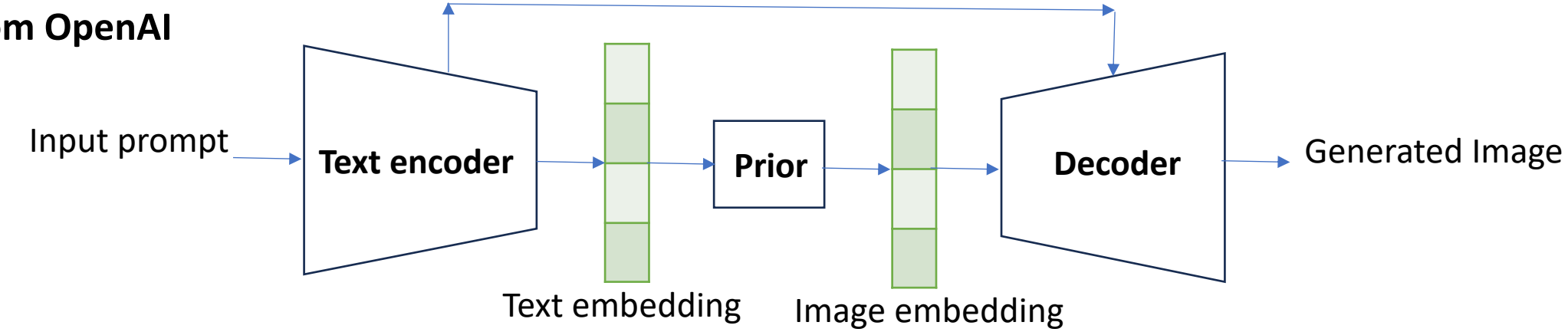
- Once the denoised latent is obtained, it's passed through a **decoder** to reconstruct the full-resolution image (Upsamples latent representation to reconstructs full-resolution image.

## ➢ Multimodal Models

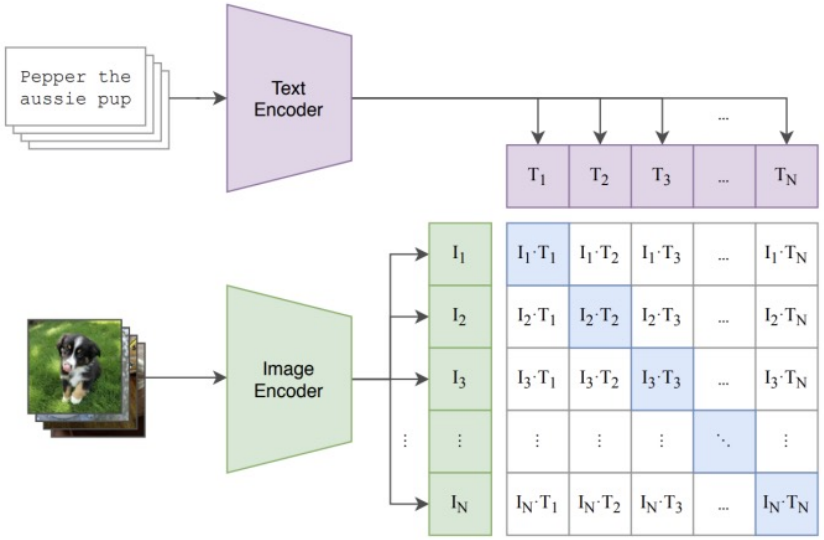
We examine the following architectures:

- DALL.E 2 and CLIP from OpenAI
- Stable Diffusion from Stability AI

## ➢ DALL.E 2 from OpenAI




- **Text encoder:** converts the text prompt into an embedding vector that represents the conceptual meaning of the text prompt within a latent space.
  - **CLIP:** Contrastive Language-Image Pre-training (CLIP) uses contrastive learning to match images with text descriptions.
    - ***Contrastive Learning***: trains 2 **transformers**, a **text encoder** to convert text to a text embedding (employed in DALL.E 2 text encoder) and an **image encoder** to convert an image to an image embedding. Given a batch of text-image pairs, we compare all text and image embedding combinations using **cosine similarity** and train the network to maximize the score between matching text-image pairs and minimize the score between incorrect text-image pairs.




• Given a batch of **N (image, text)** pairs, CLIP is trained to identify which of the **N × N** possible image-text pairings in the batch are the true matches.

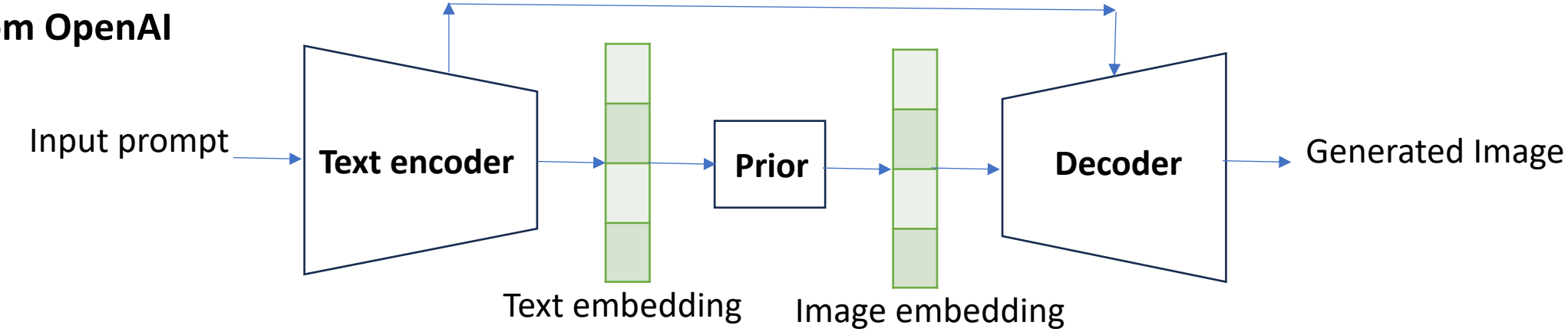
•**Text Encoder**: Each caption *i* is passed through a Transformer-based text encoder, producing a text embedding $T_i \in R^D$.

• **Image Encoder**: Each image *i* is passed through a Vision Transformer (ViT) or CNN, producing an image embedding $I_i \in R^D$.

• CLIP learns a **shared image-text embedding space** by jointly training the encoders using a **contrastive loss**:
  - The cosine similarity between the embeddings of the **N correct (image, text)** pairs is maximized
  - The similarity between all **$N^2$ – N incorrect (mismatched)** pairs is minimized .
  - This contrastive objective enables CLIP to align image and text modalities in a way that supports zero-shot transfer across a wide range of vision-language tasks.

Reference: Radford, Alec, et al. Learning Transferable Visual Models From Natural Language Supervision. ICML, 2021. https://arxiv.org/abs/2103.00020

## ➢ DALL.E 2 from OpenAI

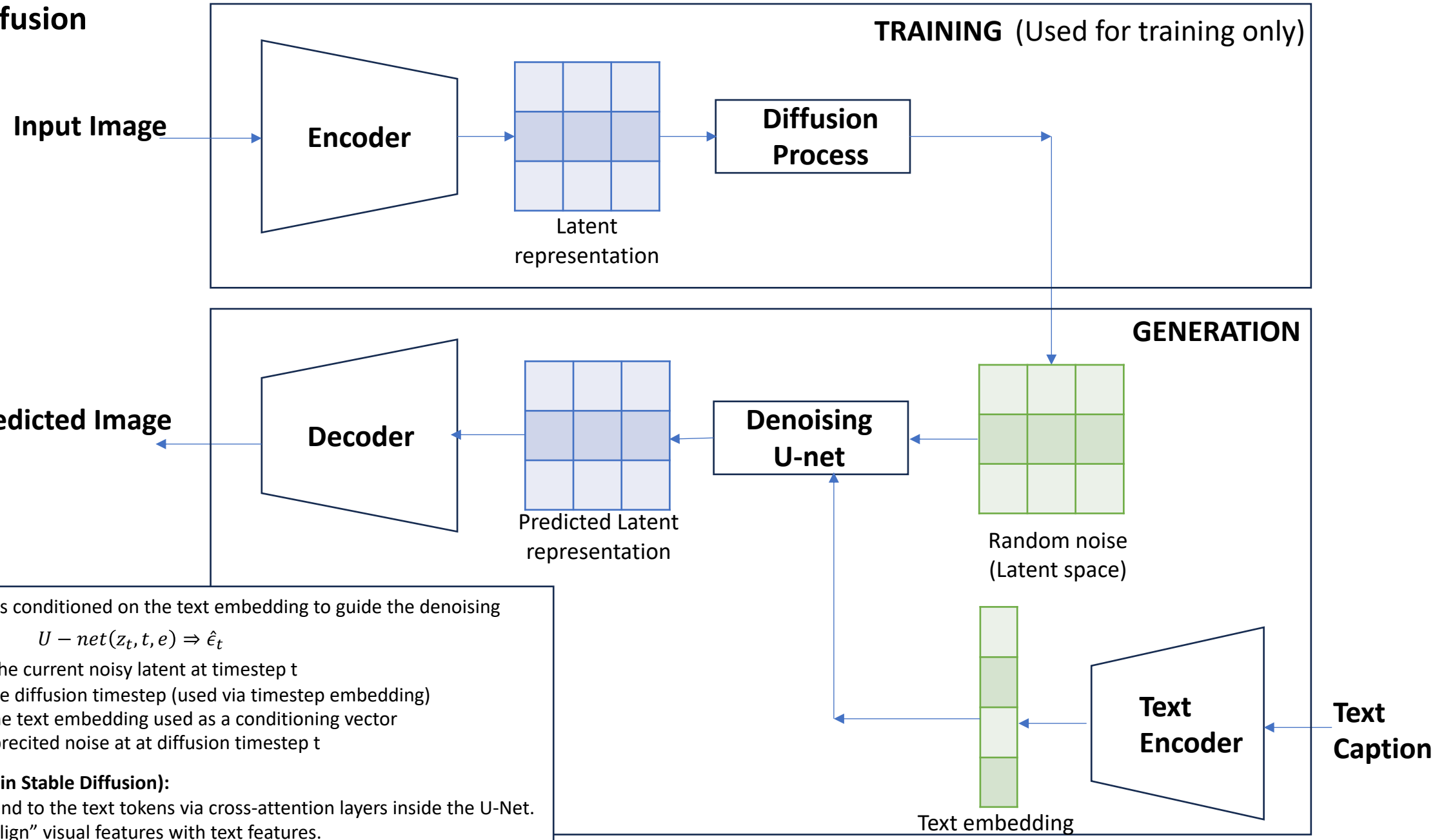



- **Text encoder:** converts the text prompt into an embedding vector that represents the conceptual meaning of the text prompt within a latent space.
  - **CLIP:** Contrastive Language-Image Pre-training (CLIP) uses contrastive learning to match images with text descriptions.
    - ***Contrastive Learning***: trains 2 **transformers**, a **text encoder** to convert text to a text embedding (employed in DALL.E 2 text encoder) and an **image encoder** to convert an image to an image embedding. Given a batch of text-image pairs, we compare all text and image embedding combinations using **cosine similarity** and train the network to maximize the score between matching text-image pairs and minimize the score between incorrect text-image pairs.

- **(Diffusion) Prior:** Converts text embedding into a CLIP image embedding.
  - Training: each CLIP text and image embedding pair → concatenated into a single vectors → image embedding is noised to turn into random noise in several steps → diffusion prior is trained to predict the denoised image embedding conditioned on the text embedding.
  - To generate a new image embedding, we sample a random vector, prepend the related text embedding and pass it through the trained diffusion prior.

- **Decoder:** Generates the final image conditioned on the text prompt and the predicted image embedding output by the prior
  - **GLIDE:** generates realistic images from text prompts working directly with the raw text prompts instead of CLIP embeddings. It is trained as diffusion model with U-net for denoiser and Transformer for the text encoder to generate text embedding used to guide the U-net through out the denoising process. It also has an Upsampler trained to scale the generated image.

# ➢ Stable Diffusion




•The Denoising U-Net is conditioned on the text embedding to guide the denoising process:

$$U - net(z_t, t, e) \Rightarrow \hat{\epsilon}_t$$

- $z_t$: the current noisy latent at timestep t
- $t$: the diffusion timestep (used via timestep embedding)
- $e$: the text embedding used as a conditioning vector
- $\hat{\epsilon}_t$ : precited noise at at diffusion timestep t

**Cross-Attention (used in Stable Diffusion):**

•The latent tokens attend to the text tokens via cross-attention layers inside the U-Net.

•This lets the model "align" visual features with text features.

# Foundation Models

## ➢ Foundation Models:

Large models trained on **broad, diverse data** using **self-supervised learning**, capable of generalizing across many tasks.

**Characteristics:**

- Trained on massive unlabeled datasets (e.g., web, books, code).
- Learn general representations useful for many downstream tasks.
- Often used in zero-shot, few-shot, or fine-tuned ways.
- Require high compute and data for training.

**Pros:**

- Can be adapted to many tasks.
- Reduce the need for task-specific model design.
- Enable transfer learning and rapid prototyping.

**Key Technologies Foundation Models:**

- Transformers (attention-based architecture)
- Large-scale parallel training
- Distributed optimization (e.g., Adam, LAMB, ZeRO)
- Prompt engineering and instruction tuning
- Reinforcement Learning from Human Feedback (RLHF)

## ➢ Foundation Models:

**Examples:**

- GPT-3, GPT-4: language tasks
- CLIP: image-text understanding
- DINO, MAE:  vision tasks
- BERT, T5, PaLM, Gemini:  multiple modalities

**Example Categories:**

| Modality | Foundation Model Examples | Based on Language Model? |
|---|---|---|
| **Language** | GPT-3/4, BERT, T5, PaLM | Yes |
| **Vision** | MAE, DINO, ViT (Vision Transformer), SAM | No |
| **Audio** | Whisper, AudioMAE, Wav2Vec 2.0 | No |
| **Code** | Codex, AlphaCode, Code LLaMA | Yes (language modeling applied to code tokens) |
| **Multimodal** | CLIP, Flamingo, Gemini, GPT-4V, Kosmos-1 | Partially (many use a language model as one component) |
| **Robotics** | RT-2 (Google), GR00T (NVIDIA) | Partially (many use a language model as one component) |

# ➢ Foundation Models

How to use foundation models for specific tasks:

➢ **Transfer Learning:**

1. Inference-Based (Prompt-Based): Zero-Shot, Few-Shot
2. Training-Based: Feature Extraction (Frozen), Partial Fine-Tuning, Full Fine-Tuning

## 1. Inference-Based Transfer Learning

*(a.k.a. Prompt-Based Learning — no model training involved)*

- The pretrained model is used as-is, with no weight updates.
- Task-specific information is provided via prompts or examples during inference.

Includes:

• **Zero-Shot Learning:** No examples are provided; only instructions.

• **Few-Shot Learning:** A few task-specific examples are included in the prompt.

## 2. Training-Based Transfer Learning

- The pretrained model is **further trained** (i.e., weights are updated) on the target task.
- This allows better task-specific performance at the cost of training time and compute.

Includes:

• **Partial Fine-Tuning:** Only a subset of layers (e.g., the top few) are updated.

• **Full Fine-Tuning:** All layers are updated on the new task. Take a pre-trained model and train it further on a specific labeled dataset, usually with a task-specific loss function(we change the weights of the model to adapt it better to the new task). Example: Fine-tune GPT-3 on customer support emails to build a virtual assistant tailored for technical support.

• **Feature Extraction (Frozen Encoder + Trainable Head):** Base model is frozen; only a small head is trained. Only use the model to extract features or representations from input data. Then use those features as input to a separate model (usually a small classifier or regressor) that you do train.

**Zero-Shot Learning**

Example:
Prompt: "Classify this email: 'Congratulations! You've won a free cruise. Click here to claim your prize.'"
Output: "Spam"

**Few-Shot Learning**

Example:
Prompt: Classify the email content:
"Your Amazon receipt for the headphones is attached." → Not Spam
"Earn $$$ working from home. No experience needed!" → Spam
"Your weekly newsletter from Stack Overflow." → ?
Output: "Not Spam"

**Fine-Tuning**

Example: Fine-tuning GPT on a legal document Q&A dataset to build a legal assistant chatbot.

## ➢ Foundation Models:

Typical **pipeline**[1] for building foundation models:

**1. Unsupervised Pretraining (Self-Supervised Learning)**

- The model is trained on a massive corpus of unlabeled data using self-supervised objectives (e.g., masked token prediction, next token prediction).
- Example: GPT is trained to predict the next word in a sentence (causal language modeling).

**2. Supervised Fine-Tuning (Optional)**

- The model is fine-tuned on smaller, labeled datasets to specialize in specific tasks (e.g., summarization, classification).
- This step injects task-specific knowledge.

**3. Reinforcement Learning from Human Feedback (RLHF)**

- The model's responses are improved using human preference data.
- A reward model is trained on human ratings, and the model is fine-tuned using reinforcement learning (e.g., PPO) to generate preferred outputs.

1: Not all foundation models go through **all three** steps. Some stop at step 1 or 2 depending on their use case.

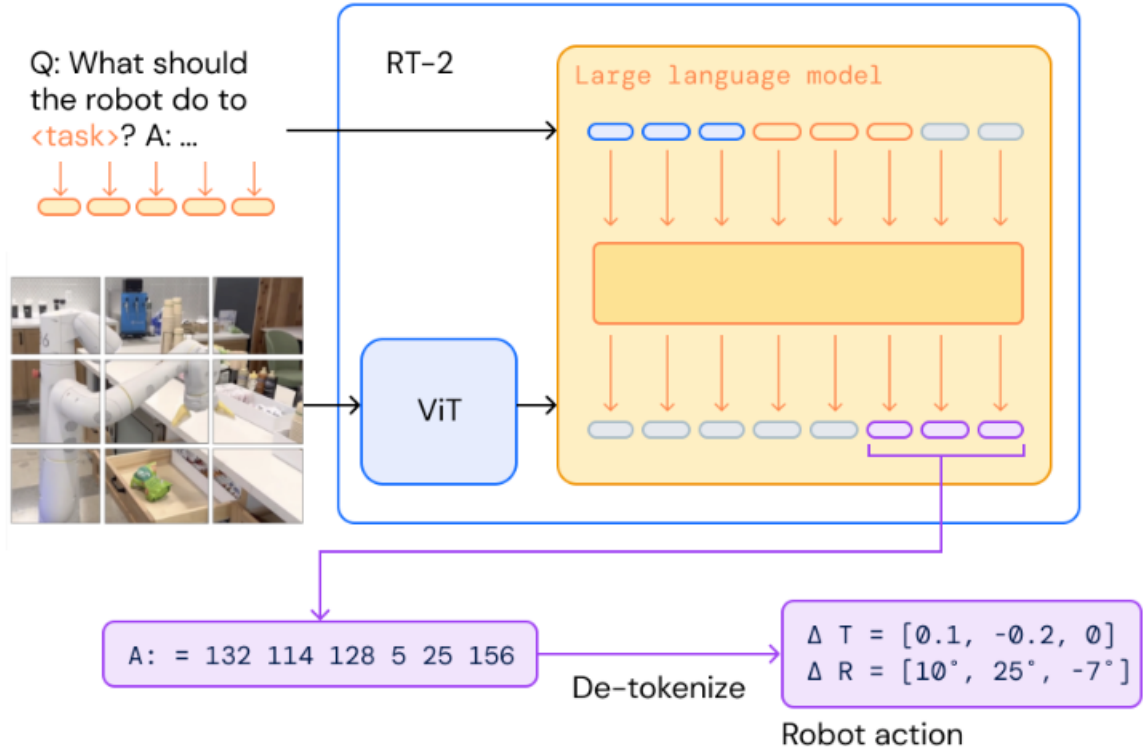
# ➢ Examples of Robotics Foundation Models

## ➢ RT-2 (Robotics Transformer 2) - Google DeepMind (2023)

- **Model Basis:**
  - Builds on Vision-Language Models (VLMs) like PaLI-X and Flamingo
  - Multimodal pretraining: leverages web-scale data (images + language) to build visual and semantic understanding
  - Then fine-tuned on robotic data to map vision + text to actions
  - Learns to execute long-horizon, language-driven robotic tasks

- **Key Idea:** Transfer web-scale vision-language knowledge to the robotics domain with minimal robotic data.

Brohan, Anthony, et al. "RT-2: Vision-language-action models transfer web knowledge to robotic control." *arXiv preprint arXiv:2307.15818*, https://arxiv.org/pdf/2307.15818 (2023).
https://robotics-transformer2.github.io/



**Text (e.g., "Pick up the blue block")**

↓

Tokenizer → e.g., 20 tokens

↓

Each token is mapped to an embedding D-dimensional vector

(Token Embedding + Positional Encoding) ∈ $R^{20\times D}$

↓

**Token embedding matrix : $R^{20\times D}$** (token + positional encodings)

1

2

**Image ∈ $R^{224 \times 224 \times 3}$**

↓

Split into non-overlapping patches (e.g., 16 × 16 pixels)

(224 / 16)^2 = 196 patches

↓

**196 patches ∈ $R^{16\times16\times3}$**

↓

Flatten each patch: 16×16×3 = 768 → patch vector ∈ $R^{768}$

↓

**Matrix of flattened patches: $R^{196\times768}$**

↓

Linear projection to embedding dim D (e.g., D = 768, 1024)

→ (Projected patch matrix + Positional encoding) ∈ $R^{196\times D}$

↓

**Patch embedding matrix : $R^{196\times D}$**

↓

ViT Processes sequence of patch embeddings through transformers

↓

ViT Output ∈ $R^{196\times D}$

↓

Concatenate Vision + Text: [196 + 20, D] = [216, 768]

↓

Transformer (LLM-based)

↓

Outputs a sequence of discrete tokens, e.g., 6 action tokens: [132, 114, 128, 5, 25, 156]

These tokens come from a learned action vocabulary (e.g., 32,000 tokens)

↓

De-tokenizer (learned mapping from tokens to continuous control)

[132, 114, 128, 5, 25, 156] → = [0.1, -0.2, 0.0, 10°, 25°, -7°]

↓

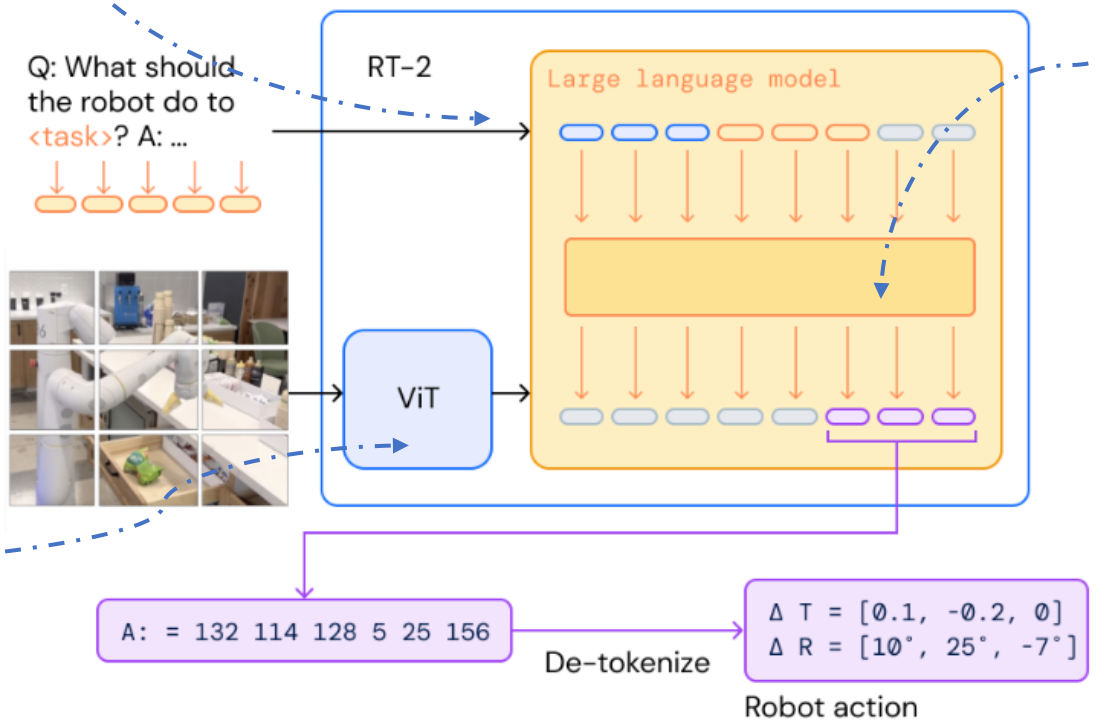
Robot action = [ΔT, ΔR] ∈ $\mathbf{R}^6$

where,

ΔT ∈ $\mathbf{R}^3$ : [Δx, Δy, Δz]= [0.1, -0.2, 0.0] (translation)

ΔR ∈ $\mathbf{R}^3$ : [Δroll, Δpitch, Δyaw]= [10°, 25°, -7°] (rotation)

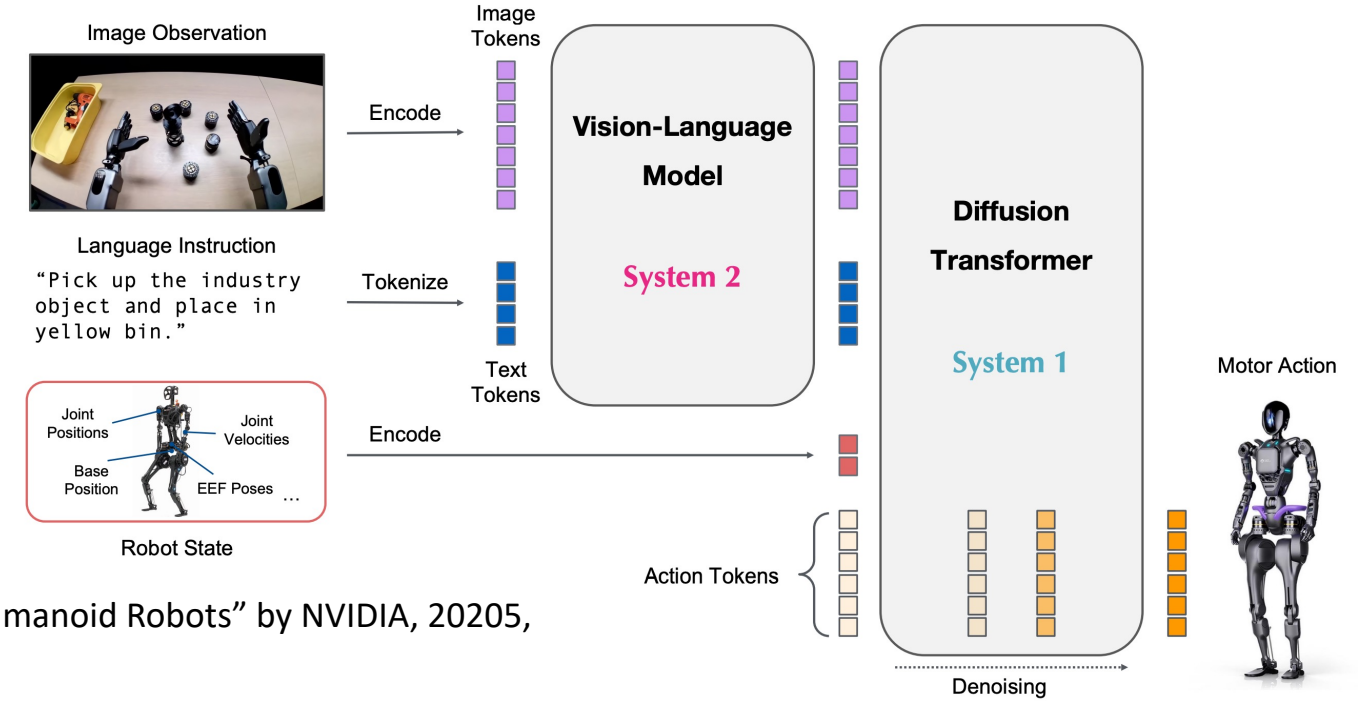
3



# ➢ Examples of Robotics Foundation Models

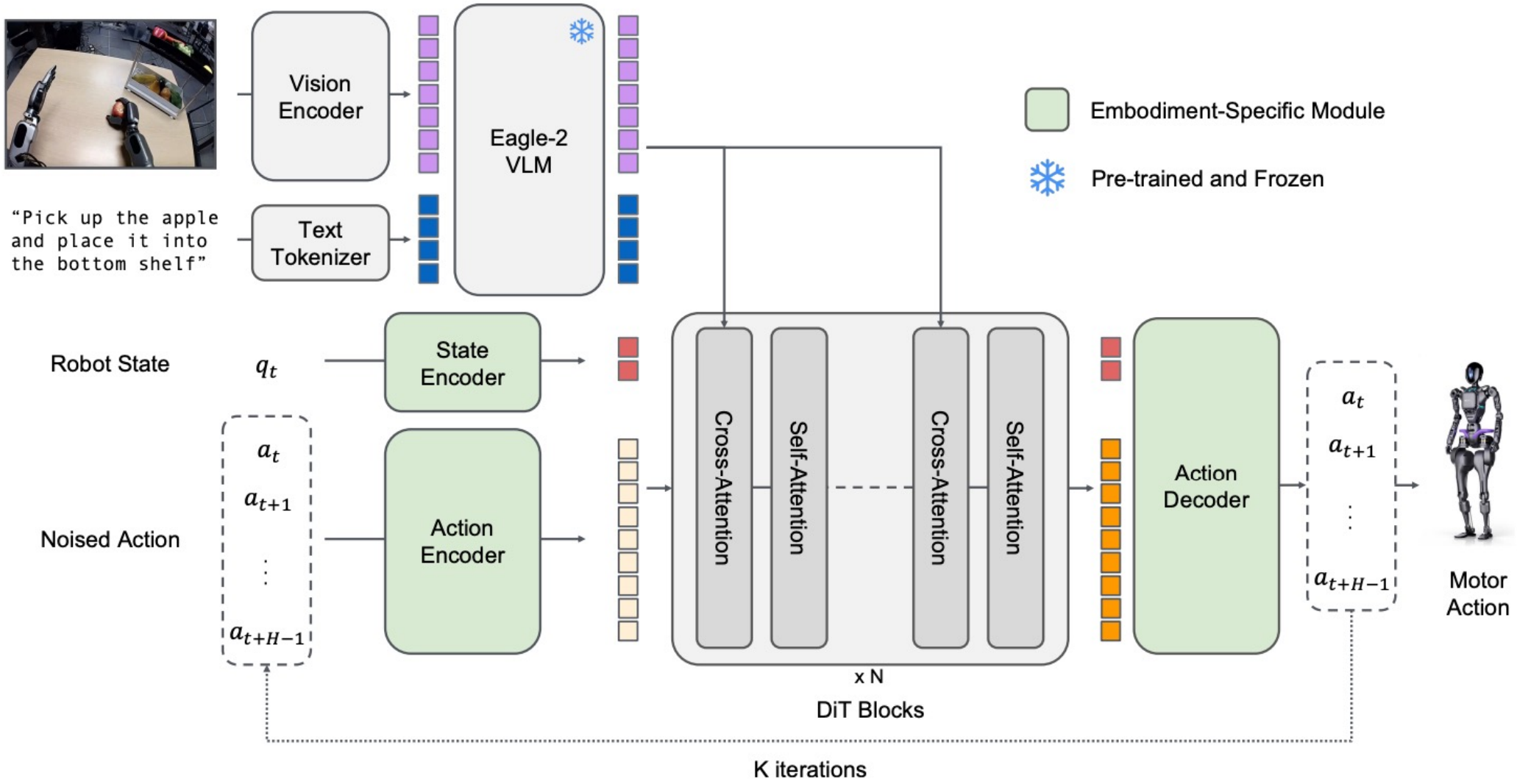
## ➢ GR00T N1: An Open Foundation Model for Generalist Humanoid Robots, NVIDIA

- Designed to enable humanoid robots to perform a wide range of tasks through a unified model.
- **Key Features:**
  - **Multimodal inputs**: Vision (images/video), proprioception (e.g., joint states), and language instructions.
  - **Transformer-based architecture**: Inspired by language models like GPT, GR00T tokenizes all input streams and processes them via a large transformer.
  - **Tokenized world representation**: Everything — vision, commands, joint angles, actions — is encoded as a sequence of tokens, enabling generalization and compositionality.
  - **Pretraining and fine-tuning**: Trained on diverse robot behaviors, simulated environments, and large-scale human demonstrations.
  - **Modularity**: Can control various types of humanoid robots (arms, hands, legs) in both simulation and real-world environments.
  - **Open Model**: NVIDIA aims to make GR00T a foundation for embodied AI research, releasing the model and code to the public.




- Paper: "GR00T N1: An Open Foundation Model for Generalist Humanoid Robots" by NVIDIA, 20205, https://arxiv.org/pdf/2503.14734

# GR00T N1 Model Architecture




- A diffusion transformer (DiT) processes the robot's proprioceptive state (e.g., joint states) and noised action tokens, and uses cross-attention to condition on image and text tokens, from the Eagle-2 VLM backbone. It outputs denoised motor actions over the time steps $t$ through $t + H - 1$.

## GR00T N1 Model Architecture

➢ **MLP-based State and Action Encoder:**

- To process states and actions of varying dimensions across different robot embodiments, GR00T uses an MLP per embodiment to project them to a shared embedding dimension (conceptually similar to CLIP's multimodal embedding space, where it maps image and text to the same embedding space) as input to the DiT.
- The Action Encoder MLP also encodes the **diffusion timestep** together with the noised action vector.

**Why Include the Timestep:** In diffusion models, the same model is used at all timesteps, but its behavior must change with the timestep. So the model needs to know what step it's on. Usually, this is done by encoding the diffusion timestep into a vector and injecting it into the model, often via addition or conditioning. This is similar to positional encoding in transformers — it gives the model a sense of "where in the process" it is.

➢ **Vision-Language Module**

- For encoding vision and language inputs, GR00T N1 uses the Pre-trained model (Eagle-2 vision-language model (VLM)) pretrained on Internet-scale data.
- Images are encoded at resolution 224 × 224 followed by pixel shuffle (Shi et al., 2016), resulting in 64 image token embeddings per frame (image patch), e.g., shape = [64 × D].
- The image tokens and the textual task description (like "Put the orange on the plate") are passed together into the Eagle-2 LLM component (a large transformer).
- The text is in "chat format", meaning it mimics conversational instructions, consistent with how Eagle-2 was trained.
- The LLM processes: i) Vision tokens (64/image), 2) Language tokens, 3) and fuses them into a **joint embedding space.**
- **Output of the Vision-Language Module:** The output is a tensor of shape: batch_size × sequence_length × hidden_dim
- This unified vision-language representation is what gets passed to the DiT for action generation.
- **Middle-Layer Representations**: Instead of using the final output layer of the LLM, GR00T uses the 12th layer output. Middle-layer features tend to be richer and more general (less overfit to LLM-specific objectives). They are also faster to extract and empirically showed better downstream policy performance.

## GR00T N1 Model Architecture

➢ **Diffusion Transformer Module:**

- GR00T learns to generate robot actions using a Diffusion Transformer (DiT), where actions are gradually "denoised" from random noise. It takes as input: 1) noisy action tokens $A\tau_t$ (sampled partway through the diffusion process), 2) the robot's proprioceptive state embeddings, 3) Vision-language tokens, and learns to predict the direction of denoising — i.e., how to update the noisy action toward the correct one.

- **Action Denoising via Flow Matching:** GR00T uses Flow Matching instead of reverse-time sampling like denoising diffusion probabilistic models (DDPMs).
- Given: 1) ground-truth action chunk $A_t$ , 2) diffusion timestep $\tau \in [0, 1]$, 3) sampled noise $\epsilon \sim \mathcal{N}(0, I)$, It constructs the **noised action**:

$$A_t^\tau = \tau A_t + (1 - \tau)\epsilon$$

- The model predicts how to "move" the noisy action toward the target using: $V_\theta(\phi_t, A_t^\tau, q_t) \approx \epsilon - A_t$
- This forms the **loss function**: $L_{fm}(\theta) = E_\tau [\|V_\theta(\phi_t, A_t^\tau, q_t) - (\epsilon - A_t)\|^2]$

- **Inference (Action Generation):** During inference, GR00T generates actions using K-step forward Euler integration: 1) Start with random noise $A_t^0 \sim \mathcal{N}(0, I)$, 2) Iterate $K$ times (e.g., $K = 4$) to denoise: $A_t^{\tau+\frac{1}{K}} = A_t^\tau + (\frac{1}{K})V_\theta(\phi_t, A_t^\tau, q_t)$

➢ **MLP-based Action Decoder:**

After DiT finishes denoising, the output token sequence is passed through an embodiment-specific MLP (Action Decoder) to produce the final robot actions (e.g., joint positions, velocities, etc.).

# Topics:

## ➢ Generative AI - Architectures

- Multilayer Perceptrons (MLPs)
- Training and Loss Functions Types
- Backpropagation Algorithm
- Stochastic Gradient Descent (SGD) and Adam Optimizer
- Common Training Issues, Regularization in Deep Learning, and Scaling Laws for Deep Learning
- Convolutional Neural Networks (CNNs)
- PixelCNN
- U-Net Denoising Model
- Recurrent Neural Networks (RNNs)
- LSTM (Long Short-Term Memory)
- GRU (Gated Recurrent Unit)
- Transformers: Self-Attention, Multi-Head Attention, and Cross-Attention
- Diffusion Transformers (DiTs)
- Vision Transformers (ViTs)
- Attention-Based U-Nets
- Multimodal Models
- Foundation Models

# Topics:

## ➢ Generative AI - Algorithms

- Flow Models
- Ordinary Differential Equation (ODE)-based Flow Models
- Denoising Diffusion Models (DDMs)
- Stochastic Differential Equation(SDE)-based Denoising Diffusion Models
- Autoencoders and Variational Autoencoders (VAEs)
- Latent Space Diffusion Models
- Autoregressive Models
- Generative Adversarial Networks (GANs)

## ➢ Generative AI - Architectures

- Multilayer Perceptrons (MLPs)
- Training and Loss Functions Types
- Backpropagation Algorithm, Stochastic Gradient Descent (SGD), and Adam Optimizer
- Common Training Issues, Regularization in Deep Learning, and Scaling Laws for Deep Learning
- Convolutional Neural Networks (CNNs) and PixelCNN
- U-Net Denoising Model
- Recurrent Neural Networks (RNNs), LSTM (Long Short-Term Memory), and GRU (Gated Recurrent Unit)
- Transformers: Self-Attention, Multi-Head Attention, and Cross-Attention
- Diffusion Transformers (DiTs), Vision Transformers (ViTs), and Attention-Based U-Nets
- Multimodal Models
- Foundation Models

# Appendices

# Appendices:

- Key Differential Equations in Generative AI
- Fine-tuning Large Language Models
- Deep Reinforcement Learning - Key Concepts and Summary
  - PG, VPG, PPO, DDPG, TD3, SAC
- Reinforcement Learning from Human Feedback (RLHF) and Imitation Learning
- Adversarial Training, Robustness in Language Models, and Language Models Evaluation
- Python Libraries for Generative AI

# ➢ Key Differential Equations in Generative AI

- **Flow Model: ODE:** $\frac{dx_t}{dt} = u_t^\theta(x_t)$, Initial state: $x_0 \sim p_{init}$ , $x_1 \sim p_{data}$

- **Continuity Equation:** $x_t \sim p_t$, $\frac{dp_t(x)}{dt} = -div(\, p_t(x)u_t(x)\,)$

PDE that governs the time evolution of probability distribution through the ODE, where div is divergence operator $div(f) = \sum_i \frac{\partial}{\partial x_i} f(x)$

- **Diffusion Model: SDE:** $dx_t = u_t^\theta(x_t)dt + \sigma_t dW_t$, Initial state: $x_0 \sim p_{init}$

- **Fokker-Planck Equation (Forward Kolmogorov Eq):** $x_t \sim p_t$, $\frac{\partial p_t(x)}{\partial t} = -div(\, p_t(x)u_t(x)\,) + \frac{\sigma_t^2}{2}\Delta p_t(x)$

PDE that governs the time-evolution of probability distribution through the SDE, where Δ is Laplacian operator $\Delta f = \sum_i \frac{\partial^2}{\partial^2 x_i} f(x)$

- **Denoising Diffusion Model:** $dx_t = \left[u_t^\theta(x_t) + \frac{\sigma_t^2}{2}\nabla_x \log p_t(x)\right]dt + \sigma_t dW_t$, Initial state: $x_0 \sim p_{init}$ , $x_1 \sim p_{data}$

- **Langevin dynamics:** special case of denoising diffusion model where $u_t^\theta(x_t) = 0$ $dx_t = \left[\frac{\sigma_t^2}{2}\nabla_x \log p(x)\right]dt + \sigma_t dW_t$

$p(x)$ is a stationary distribution of Langevin dynamics, $x_0 \sim p(x), x_t \sim p(x)\ t \geq 0$ (The probability distribution remains stationary over time $p_t(x) = p(x)$ )

If $x_0 \sim \tilde{p}_0(x) \neq p(x),\ x_t \sim \tilde{p}_t(x) \to p(x)$ : Distribution of $x_t$ converges to $p(x)$

# Fine-tuning Large Language Models

**1. Supervised Fine-Tuning (SFT)**

Supervised fine-tuning is the most straightforward approach. It involves training a pre-trained language model on a curated dataset with input-output pairs (e.g., question-answer, instruction-response). The model learns to generate desired outputs for given prompts by minimizing a supervised loss (typically cross-entropy loss).

**Use case:** Aligning the model with domain-specific tasks or instruction-following abilities.

**2. Knowledge Distillation:**

In this technique, a smaller or more efficient model (the student) is trained to mimic the behavior of a larger, more capable model (the teacher). The student learns not just from ground truth labels but also from the teacher's output (soft targets), which provide richer training signals.

**Use case:** Creating smaller, faster models by compressing LLMs, while maintaining comparable performance.

**3. Policy Optimization (Reinforcement Learning):**

After initial SFT, models can be further fine-tuned using Reinforcement Learning (RL) to optimize behavior according to human preferences or safety constraints. A popular example is RLHF (Reinforcement Learning from Human Feedback), where: i) A reward model is trained using human-labeled preferences, ii) The LLM is further fine-tuned using policy optimization algorithms (e.g., PPO) to maximize the reward.

**Use case:** Making model outputs more aligned, helpful, safe, or preference-consistent.

## 2. Knowledge Distillation:

Knowledge distillation is a model compression technique where:

- A large, high-performing model (called the teacher) is used to generate soft labels (i.e., predicted probabilities or outputs).
- A smaller model (called the student) is trained to mimic the teacher's behavior using these soft labels.

**Input → Teacher Model → Soft Output → Student Model (Trained to Match Teacher)**

How It Works:

1. Input Data Feeding:
   - A dataset (e.g., prompts or sentences) is passed into the teacher model.
2. Teacher Prediction:
   - The teacher model outputs logits or probability distributions over possible outputs.
   - These are often "**softer**" than hard labels — showing which tokens the teacher thinks are likely, not just the most likely.
   - Soft labels contain **more information** than hard labels, e.g., instead of just saying "the answer is 'cat'," the teacher might say: cat: 70%, dog: 20%, mouse: 10%
3. Student Learning:
   - The student model takes the same input and tries to generate similar outputs.
   - The student is trained using a loss function to minimize the difference between its outputs and those of the teacher.
   - Instead of (or in addition to) cross-entropy with ground-truth labels, it uses Kullback-Leibler divergence to learn the teacher's full distributions, i.e., $L = KL - Divergence(P_{teacher} \parallel P_{student})$

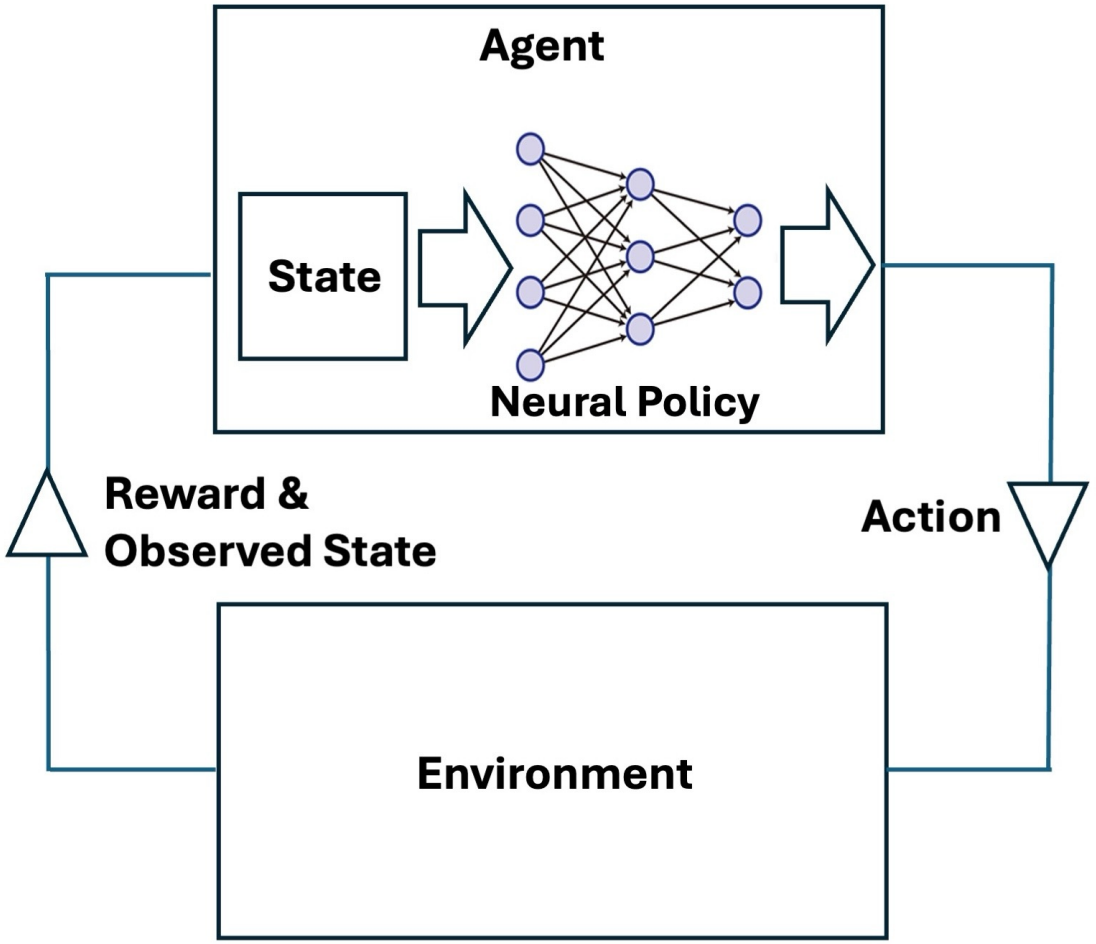
# Deep Reinforcement Learning (DRL)

## Key Concepts and Summary

- Deep Reinforcement Algorithms
- Reinforcement Learning from Human Feedback (RLHF)
- Imitation Learning

# Deep Reinforcement Algorithms

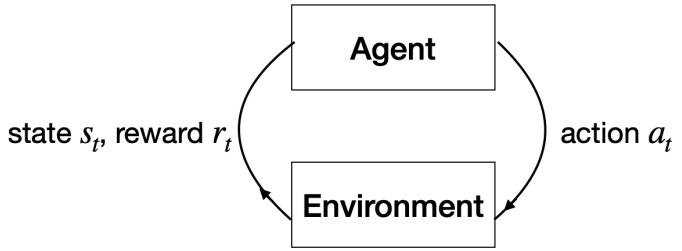
- Achiam, J., & OpenAI. *Spinning Up in Deep Reinforcement Learning*. https://spinningup.openai.com/en/latest/




**(Deep) Reinforcement Learning** is a framework where an agent learns to make decisions by interacting with an environment.

1) At each time step, the agent observes the state of the environment, 2) A neural policy (deep neural network) maps the state to an action (Deep Reinforcement Learning), 3) The environment responds with a reward and a new state, 4) The agent's objective is to maximize cumulative reward over time.

- **RL:** a type of machine learning where an agent learns to make decisions by interacting with an environment.
  - The agent takes an action in a given state.
  - The environment responds with a new state and a reward.
  - The agent's goal is to maximize cumulative reward over time.




| | |
|---|---|
| **Agent** | The learner or decision maker. |
| **Environment** | Everything the agent interacts with. |
| **State ($s_t$)** | The current situation of the agent. |
| **Action ($a_t$)** | The move the agent makes. |
| **Trajectory ($\tau$)**<br>**(episodes** or **rollouts)** | Sequence of states and actions $\tau = (s_0, a_0, s_1, a_1, ..)$, $s_0 \sim p_0(.)$ is randomly sampled from the initial state distribution<br>**State transition**: Deterministic $s_{t+1} = f(s_t, a_t)$ or Stochastic: $s_{t+1} \sim P(.|s_t, a_t)$<br>**Probability of a trajectory:** $\tau \sim \pi$, $P(\tau|\pi) = p_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$ |
| **Reward ($r_t$)** | Feedback from the environment after an action e.g., $r_t = R(s_t, a_t, s_{t+1})$ |
| **Policy ($\pi$)** | Strategy that the agent uses to choose actions. Mapping from state $s_t$ to action $a_t$.<br>**Deterministic policy** $a_t = \mu(s_t)$ , **Stochastic policy** $a_t \sim \pi(.|s_t)$ : sampled from a probability distribution over actions, given the state $s_t$, e.g., $\pi(.|s_t) = N(\mu_\theta(s_t), \Sigma_\theta(s_t))$ , e.g., Softmax over discrete action logits |
| **Value Function ($V^\pi(s)$)** | Estimates how good a state (or state-action pair) is in terms of expected future rewards, e.g.,"If I'm in state $s_t$, how much total reward can I expect to collect in the future if I keep acting well (according to the optimal policy)?"<br>There's also a **Q-value function** $Q^\pi(s, a)$, which estimates the expected return of taking action $a$ in state $s$ and then following policy $\pi$<br>**Bellman Equation:** The value of your current state is equal to the reward you get now, plus the value of wherever you end up next (It's a recursive way of bootstrapping value estimates)<br>$V^\pi(s_t) = \mathrm{E}_{a_t \sim \pi, s_{t+1} \sim P}[r_t + \gamma V^\pi(s_{t+1})]$, Optimal value function: $V^*(s_t) = \max_a \mathrm{E}_{s_{t+1} \sim P}[r_t + \gamma V^*(s_{t+1})]$<br>$Q^\pi(s_t, a_t) = \mathrm{E}_{a_{t+1} \sim \pi, s_{t+1} \sim P}[r_t + \gamma Q^\pi(s_{t+1}, a_{t+1})]$, Optimal Q function: $Q^*(s_t, a_t) = \mathrm{E}_{s_{t+1} \sim P}[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})]$ |

- **RL Optimization:** *Find a policy* $\pi^*$ to **maximize the total discounted reward**

$$\pi^* = \arg\max_\pi \mathrm{E}\left[\underbrace{\sum_{t=0}^{\infty} \gamma^t r_t}_{R(\tau)}\right] = \int_\tau P(\tau|\pi)R(\tau)$$

$R(\tau)$: cumulative reward over the trajectory $\tau$

- $\gamma$: Discount factor (how much future rewards are worth)

➢ **RL Algorithms**

**Model-Free**

**Model-Based**

Examples:
Model-Predictive Control (MPC)
Monte Carlo Tree Search

**Policy Optimization**

- Directly learns a **parameterized policy** $\pi_\theta(a|s)$
- Optimizes the expected return by adjusting $\theta$ using gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta \mathrm{E}\left[\sum_t \gamma^t r_t\right]$$

- **on-policy**: each update only uses data collected while acting according to the most recent version of the policy.
- Works for **discrete and continuous** actions
- Higher Variance(**gradient estimates** fluctuate significantly)
- Often includes learning a **value function** $V_\phi(s)$ to reduce the variance of policy gradient estimates.

**Q-Learning**

- **Learns Q-function** $Q_\theta(s,a)$
- Typically, they use an objective function based on the Bellman equation.
- The **policy is implicit**: $\pi(s) = \arg\max_a Q(s,a)$
- **off-policy:** each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained.
- The behavior policy is often deterministic, unless exploration is introduced via stochasticity (e.g., ε-greedy).
- Best for **discrete** actions
- Lower Variance(More consistent)

Policy Gradient
Synchronous Advantage Actor-Critic (A2C)
Asynchronous Advantage Actor-Critic (A3C)
Proximal Policy Optimization (PPO)
Trust Region Policy Optimization (TRPO)

Deep Deterministic Policy Gradient (DDPG)
Twin Delayed DDPG(TD3)
Soft Actor-Critic (SAC)

Deep Q-Networks(DQN)
Categorical 51-Atom DQN(C51)
Quantile Regression DQN(QR-DQN)
HER(Hindsight Experience Replay)

# Policy Optimization

- We need to describe the gradient of the policy performance ($\nabla_\theta J(\pi_\theta)$ : **policy gradient**) with respect to policy parameters: $\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)$

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) && \text{Expand expectation} \\
&= \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) && \text{Bring gradient under integral} \\
&= \int_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) && \text{Log-derivative trick} \\
&= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta) R(\tau)] && \text{Return to expectation form} \\
\therefore \nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right] && \text{Expression for grad-log-prob}
\end{aligned}
$$

| | | |
|---|---|---|
| $\pi_\theta(a\|s)$ | Scalar probability | $\in [0,1]$ |
| $\log \pi_\theta(a\|s)$ | Scalar | $\in \mathbb{R}$ |
| $\nabla \pi_\theta(a\|s)$ | **Vector** | Same size as parameters $\theta$ |

Collect a set of trajectories $D = \{\tau_i\}_{\{i=1,\dots,N\}}$ where each trajectory is obtained by letting the agent act in the environment using the policy $\pi_\theta$, the policy gradient can be estimated with

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau),$$

where |D| is the number of trajectories.

For each trajectory, i) loop over t = 0 to T, ii) at each time step, compute the gradient $\nabla_\theta log \pi_\theta(a_t \mid s_t)$, and iii) multiply each of those by **the same** scalar $R(\tau)$.

- Discrete action space: The log likelihood for an action a can then be obtained by indexing into the vector $\log \pi_\theta(a|s) = \log [P_\theta(s)]_a$
- Log-Likelihood. The log-likelihood of a k -dimensional action a, for a diagonal Gaussian with mean $\mu = \mu_\theta(s)$ and standard deviation $\sigma = \sigma_\theta(s)$ is given by

$$\log \pi_\theta(a|s) = -\frac{1}{2}\left(\sum_{i=1}^{k}\left(\frac{(a_i-\mu_i)^2}{\sigma_i^2} + 2\log\sigma_i\right) + k\log 2\pi\right)$$

- **Reward-to-go based Policy Gradient:** Actions are only reinforced based on rewards obtained after they are taken.

(Reward-to-go): $\hat{R}_t \doteq \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})$

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})\right]$$

- **Baseline based Policy Gradient:** For any function b which only depends on state $\mathop{\mathrm{E}}_{a_t\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)] = 0$

This allows us to add or subtract any number of terms to policy gradient, without changing its expectation

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)\right)\right]$$

common choice of baseline: $\boldsymbol{b(s_t) = V_\phi^\pi(s_t)}$ **(=Expected reward-to-go from state $s_t$)** to reduce the variance/ faster and more stable policy learning.

$$\phi_k = \arg\min_\phi \mathop{\mathrm{E}}_{s_t,\hat{R}_t\sim\pi_k}\left[\left(V_\phi(s_t) - \hat{R}_t\right)^2\right]$$

# **Policy Optimization**: Vanilla Policy Gradient (VPG)

**Algorithm 1** Vanilla Policy Gradient Algorithm

1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
6: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

- Advantage function: $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$
- Estimated Advantage function : $\hat{A}_t = \hat{R}_t - V_{\phi_k}(s_t)$
  Hence, Vanilla Policy Gradient : **baseline-based policy gradient** method — where the **baseline** is typically the learned **state-value function** $V_{\phi_k}(s_t)$

7: Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.
8: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$$

- Value function estimation

typically via some gradient descent algorithm.
9: **end for**

Policy Gradient Methods for Reinforcement Learning with Function Approximation, Sutton et al. 2000

Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs, Schulman 2016(a)

Benchmarking Deep Reinforcement Learning for Continuous Control, Duan et al. 2016

High Dimensional Continuous Control Using Generalized Advantage Estimation, Schulman et al. 2016(b)

# **Policy Optimization**: Trust Region Policy Optimization(TRPO)

- **Goal:** how can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so far that we accidentally cause performance collapse
- TRPO updates policies by taking the largest step possible to improve performance, while satisfying a constraint on how close the new and old policies are allowed to be (expressed in terms of KL-Divergence).
- This is different from normal policy gradient, which keeps new and old policies close in parameter space. But even seemingly small differences in parameter space can have very large differences in performance—so a single bad step can collapse the policy performance. This makes it dangerous to use large step sizes with vanilla policy gradients, thus hurting its sample efficiency. TRPO nicely avoids this kind of collapse and tends to quickly and monotonically improve performance.

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}(\theta_k, \theta)$$
$$\text{s.t. } \bar{D}_{KL}(\theta||\theta_k) \leq \delta$$

$\mathcal{L}(\theta_k, \theta)$ :surrogate advantage, a measure of how policy $\pi_\theta$ performs relative to the old policy $\pi_{\theta_k}$ using data from the old policy

$$\mathcal{L}(\theta_k, \theta) = \mathop{\mathrm{E}}_{s,a\sim\pi_{\theta_k}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a)\right]$$

$\bar{D}_{KL}(\theta||\theta_k)$ average KL-divergence between policies across states visited by the old policy:

$$\bar{D}_{KL}(\theta||\theta_k) = \mathop{\mathrm{E}}_{s\sim\pi_{\theta_k}}\left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)\right]$$

Apply a Taylor expansion to the objective:

$$\theta_{k+1} = \arg\max_{\theta} g^T(\theta - \theta_k)$$
$$\text{s.t. } \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \leq \delta$$

$$\Rightarrow \quad \theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

Due to the approximation errors introduced by the Taylor expansion, this may not satisfy the KL constraint or actually improve the surrogate advantage. TRPO adds a modification to this update rule: a backtracking line search

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

where $\alpha \in (0,1)$ is the backtracking coefficient, and $j$ is the smallest nonnegative integer such that $\pi_{\theta_k}$ satisfies the KL constraint and produces a positive surrogate advantage.

**Algorithm 1** Trust Region Policy Optimization

1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: Hyperparameters: KL-divergence limit $\delta$, backtracking coefficient $\alpha$, maximum number of backtracking steps $K$
3: **for** $k = 0, 1, 2, \ldots$ **do**
4: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
5: Compute rewards-to-go $\hat{R}_t$.
6: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
7: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

8: Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
9: Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

where $j \in \{0, 1, 2, \ldots K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.
10: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$$

typically via some gradient descent algorithm.
11: **end for**

# **Policy Optimization**: Proximal Policy Optimization(PPO)

- **Goal:** same as TRPO
- PPO is a family of first-order methods (instead of TRPO's second-order method); Hence, PPO methods are significantly simpler to implement.

**PPO-Penalty** approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately.

**PPO-Clip** doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead, it relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.

PPO-clip updates policies via

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} [L(s, a, \theta_k, \theta)]$$

where

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \quad g(\epsilon, A^{\pi_{\theta_k}}(s, a))\right)$$

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

- If the ratio $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ deviates beyond $1 \pm \epsilon$, it stops the update from going further.

**Algorithm 1** PPO-Clip

1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t))\right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

# **Policy Optimization & Q-Learning** : Deep Deterministic Policy Gradient (DDPG)

- Concurrently learns a Q-function and a policy. It uses off-policy data and the **Bellman equation** to learn the **Q-function**, and uses the Q-function to learn the policy.
- DDPG is an off-policy algorithm.
- For **continuous control tasks**. DDPG can be viewed as an extension of Q-learning for continuous action spaces.

➢ **Components: i) Actor Network:** $\mu_\theta(s)$ deterministic policy, **ii) Critic Network:** $Q_\phi(s,a)$ estimates the value of taking action $a$ in state $s$, **iii) Target Networks**: $\mu_{\theta_{targ}}(s)$ and $Q_{\phi_{targ}}(s,a)$ : smoothed copies of the actor and critic for stable training

➢ **Goal:** The **critic** learns how good actions are (Q-leaning). The **actor** is trained to select actions that maximize the Q-values estimated by the critic (via deterministic policy gradients).

➢ **Q-Learning:**

- Bellman equation for optimal Q-function: $Q^*(s_t,a_t) = \mathrm{E}_{s_{t+1}\sim P}[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1},a_{t+1})]$
- **Mean-Squared Bellman Error (MSBE)**:

Collected a set D of transitions $(s_t, a_t, r_t, s_{t+1}, d)$ (where $d \in \{0,1\}$ indicates whether state $s_{t+1}$ is terminal)

$$\mathrm{L} = \mathrm{E}_{(s_t,a_t,r_t,s_{t+1},d)\sim D}\left[\left(Q_\phi(s_t,a_t) - (r_t + \gamma(1-d)\max_{a_{t+1}} Q_\phi(s_{t+1},a_{t+1}))\right)^2\right]$$

Target Network

d=1 for terminal state, we **ignore the next Q-value** — because there is no next state to plan for.

- Target Network depends on the same parameters we are trying to train: $\phi$. This makes MSBE minimization unstable. Instead, we use target networks $Q_{\phi_{targ}}(s_{t+1}, \mu_{\theta_{targ}}(s_{t+1}))$

$$\mathrm{L} = \mathrm{E}_{(s_t,a_t,r_t,s_{t+1},d)\sim D}\left[\left(Q_\phi(s_t,a_t) - (r_t + \gamma(1-d)\, Q_{\phi_{targ}}(s_{t+1},\mu_{\theta_{targ}}(s_{t+1})))\right)^2\right]$$

➢ **Deterministic policy gradients:**

- Given optimal Q function: $a^*(s_t) = \arg\max_a Q^*(s_t,a_t)$
- Hence: $\max_\theta E_{s\sim D}[Q_\phi(s,\mu_\theta(s))]$

➢ **Targett Network Updates:** The target network parameters in DDPG can be viewed as a **time-delayed** (or exponentially smoothed) version of the original network parameters.

- Slowly update target networks

**Algorithm 1** Deep Deterministic Policy Gradient

Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
**repeat**
  Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute $a$ in the environment
  Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
  Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
  If $s'$ is terminal, reset environment state.
  **if** it's time to update **then**
    **for** however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_\phi(s,a) - y(r,s',d))^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

    **end for**
  **end if**
**until** convergence

# **Policy Optimization & Q-Learning**: Twin Delayed DDPG (TD3)

- Improves the DDPG by:

- **Clipped Double-Q Learning:** TD3 learns two Q-functions ($Q_{\phi_1}$ and $Q_{\phi_2}$)instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

$$L_1 = E_{(s_t,a_t,r_t,s_{t+1},d)\sim D}\left[\left(Q_{\phi_1}(s_t,a_t) - (r_t + \gamma(1-d)\, min_{i=1,2} Q_{\phi_{targ_i}}(s_{t+1}, a(s_{t+1}))\right)^2\right]$$

$$L_2 = E_{(s_t,a_t,r_t,s_{t+1},d)\sim D}\left[\left(Q_{\phi_2}(s_t,a_t) - (r_t + \gamma(1-d)\, min_{i=1,2} Q_{\phi_{targ_i}}(s_{t+1}, a(s_{t+1}))\right)^2\right]$$

- **Target Policy Smoothing.** TD3 adds noise to the target action, to prevent the policy from overfitting to Q-function estimation errors by smoothing out Q along changes in action.

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \quad \epsilon \sim \mathcal{N}(0,\sigma) \longrightarrow a_{Low} \le \mu_{\theta_{targ}} + \epsilon \le a_{High}$$

$$-c \le \epsilon \le c$$

- **Delayed Policy Updates.** TD3 updates the policy (and target networks) less frequently than the Q-function, e.g., one policy update for every two Q-function updates.
The policy is learned just by maximizing $Q_{\phi_1}$ : $\max_\theta E_{s\sim D}[Q_{\phi_1}(s, \mu_\theta(s))]$

**Algorithm 1** Twin Delayed DDPG

Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
**repeat**
  Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute $a$ in the environment
  Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
  Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
  If $s'$ is terminal, reset environment state.
  **if** it's time to update **then**
    **for** $j$ in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute target actions

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \quad \epsilon \sim \mathcal{N}(0,\sigma)$$

      Compute targets

$$y(r, s', d) = r + \gamma(1-d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_{\phi_i}(s,a) - y(r,s',d))^2 \qquad \text{for } i = 1, 2$$

      **if** $j$ mod `policy_delay` $= 0$ **then**
        Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_{\phi_1}(s, \mu_\theta(s))$$

        Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1-\rho)\phi_i \qquad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

      **end if**
    **end for**
  **end if**
**until** convergence

# Policy Optimization & Q-Learning: Soft Actor-Critic (SAC)

- SAC is an off-policy algorithm.
- Uses a stochastic policy
- Adds an entropy bonus to the objective
- Like in TD3, SAC uses two Q-functions and a target network, but differs by using a stochastic policy and entropy regularization to encourage exploration.

➢ **Entropy-regularized reinforcement learning:** the agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep

$$\pi^* = \arg\max_{\pi} \underset{\tau\sim\pi}{\mathrm{E}}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t,s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\Big)\right] \quad \alpha > 0$$

$$H(P) = \underset{x\sim P}{\mathrm{E}}\left[-\log P(x)\right]$$

Value function:

$$V^{\pi}(s) = \underset{\tau\sim\pi}{\mathrm{E}}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t,s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\Big)\,\Big|\, s_0 = s\right] \Rightarrow V^{\pi}(s) = \underset{a\sim\pi}{\mathrm{E}}\left[Q^{\pi}(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right)$$

Q-function:

$$Q^{\pi}(s,a) = \underset{\tau\sim\pi}{\mathrm{E}}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^t H\left(\pi(\cdot|s_t)\right)\,\Big|\, s_0 = s, a_0 = a\right]$$

Bellman Equation:

$$\begin{aligned} Q^{\pi}(s,a) &= \underset{\substack{s'\sim P\\ a'\sim\pi}}{\mathrm{E}}\left[R(s,a,s') + \gamma\left(Q^{\pi}(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)\right] \\ &= \underset{s'\sim P}{\mathrm{E}}\left[R(s,a,s') + \gamma V^{\pi}(s')\right]. \end{aligned} \Rightarrow \underset{\substack{s'\sim P\\ a'\sim\pi}}{\mathrm{E}}\left[R(s,a,s') + \gamma\left(Q^{\pi}(s',a') - \alpha\log\pi(a'|s')\right)\right]$$

➢ **Q-Learning:** The Q-functions ($Q_{\phi_1}$, $Q_{\phi_2}$) are learned in a similar way to TD3

- Like in TD3, both Q-functions are learned with MSBE minimization, by regressing to a single shared target.
- Like in TD3, the shared target is computed using target Q-networks, and the target Q-networks are obtained by polyak averaging the Q-network parameters over the course of training.
- Like in TD3, the shared target makes use of the clipped double-Q trick.
- Unlike in TD3, the target also includes a term that comes from SAC's use of entropy regularization.
- Unlike in TD3, the next-state actions used in the target come from the **current policy** instead of a target policy.
- Unlike in TD3, there is no explicit target policy smoothing. TD3 trains a deterministic policy, and so it accomplishes smoothing by adding random noise to the next-state actions. SAC trains a stochastic policy, and so the noise from that stochasticity is sufficient to get a similar effect.

➢ **Policy Learning:** The policy should, in each state, act to maximize the expected future return plus expected future entropy. That is, it should maximize

$$\begin{aligned} V^{\pi}(s) &= \underset{a\sim\pi}{\mathrm{E}}\left[Q^{\pi}(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right) \\ &= \underset{a\sim\pi}{\mathrm{E}}\left[Q^{\pi}(s,a) - \alpha\log\pi(a|s)\right] \end{aligned}$$

Unlike in TD3, which uses $Q_{\phi_1}$ (just the first Q approximator), SAC uses $\min(Q_{\phi_1}, Q_{\phi_2})$. The policy is thus optimized according to

$$\max_{\theta} \underset{\substack{s\sim\mathcal{D}\\ \xi\sim\mathcal{N}}}{\mathrm{E}}\left[\min_{j=1,2} Q_{\phi_j}(s,\tilde{a}_{\theta}(s,\xi)) - \alpha\log\pi_{\theta}(\tilde{a}_{\theta}(s,\xi)|s)\right]$$

- Gaussian policy: $\tilde{a}_{\theta}(s,\xi) = \tanh\left(\mu_{\theta}(s) + \sigma_{\theta}(s)\odot\xi\right), \quad \xi\sim\mathcal{N}(0,I)$

**Algorithm 1** Soft Actor-Critic

Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
**repeat**
  Observe state $s$ and select action $a \sim \pi_\theta(\cdot|s)$
  Execute $a$ in the environment
  Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
  Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
  If $s'$ is terminal, reset environment state.
  **if** it's time to update **then**
    **for** $j$ in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r,s',d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s',\tilde{a}') - \alpha\log\pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}'\sim\pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i}\frac{1}{|B|}\sum_{(s,a,r,s',d)\in B}\left(Q_{\phi_i}(s,a) - y(r,s',d)\right)^2 \qquad \text{for } i = 1,2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta\frac{1}{|B|}\sum_{s\in B}\left(\min_{i=1,2} Q_{\phi_i}(s,\tilde{a}_\theta(s)) - \alpha\log\pi_\theta\left(\tilde{a}_\theta(s)|s\right)\right),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt $\theta$ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1-\rho)\phi_i \qquad \text{for } i = 1,2$$

    **end for**
  **end if**
**until** convergence

| Method | Main Idea | Policy Update Equation | Drawbacks | Improvement in Successor(s) |
|---|---|---|---|---|
| **REINFORCE** (Basic PG) | Learn directly from sampled returns by computing Monte Carlo estimates of the policy gradient. | $\theta_{k+1} = \theta_k + \alpha\, \hat{g}$<br>$\hat{g} = \frac{1}{\lvert D \rvert} \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \lvert s_t) R(\tau)$ | • High variance in gradient estimates<br>• No baseline → credits all actions equally<br>• Poor sample efficiency<br>• On-policy only (can't reuse past data)<br>• Full trajectory return $R(\tau)$ used at every step, causing delayed credit assignment | • Add a **baseline** (e.g. value function $V(s)$ ) to reduce variance → **Vanilla PG (with baseline)**<br>• Use **reward-to-go** $R_t$ instead of full return $R(\tau)$ to better assign credit → **Reward-to-go PG**<br>• Add **KL constraint** or **clipping** to stabilize updates → **TRPO**, **PPO**<br>• Use **bootstrapped critics** and off-policy sampling for efficiency → **Actor-Critic methods (DDPG, TD3, SAC)** |
| **VPG** (with baseline) | Use reward-to-go $R_t$ and a value function baseline $V_\phi(s_t)$ to reduce variance without biasing the gradient. | Same as above, but replace $R(\tau)$ with advantage estimate: $A_t = R_t - V_\phi(s_t)$ | Still on-policy; small updates can lead to slow learning. | **TRPO** introduces constrained optimization for larger but safe updates. |
| **TRPO** | Maximize a surrogate objective while **constraining KL divergence** from the old policy. | $\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$ | Complex implementation (2nd-order method); expensive due to Hessian. | **PPO** simplifies with first-order clipping instead of KL constraint. |
| **PPO** (Clip) | Simplifies TRPO by using a clipped surrogate objective (to restrict updates), avoiding KL constraints. | $\theta_{k+1} = \arg\max_\theta \mathop{\mathrm{E}}_{s,a \sim \pi_{\theta_k}} [L(s, a, \theta_k, \theta)]$<br>$L(s, a, \theta_k, \theta) = \min \left( \frac{\pi_\theta(a\lvert s)}{\pi_{\theta_k}(a\lvert s)} A^{\pi_{\theta_k}}(s,a),\ \ \mathrm{clip}\left( \frac{\pi_\theta(a\lvert s)}{\pi_{\theta_k}(a\lvert s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right)$ | Still on-policy; less sample-efficient; sensitive to clip range. | Move to **off-policy** methods for better sample reuse → **DDPG/TD3/SAC.** |
| **DDPG** | Actor-critic method for **continuous actions** using deterministic policies. | Critic: Mean-Squared Bellman Error (MSBE) loss with respect to the target network described in terms of target Q function and targe policy<br>Actor: Q-function maximization | Overestimation bias in Q-learning; unstable training. | **TD3** fixes with clipped double Q-learning and delayed policy update. |
| **TD3** | Improves DDPG with: 1) **2 critics**, 2) **target smoothing**, 3) **delayed updates**. | Same as DDPG, but uses minimum of 2 target Q-functions in MSBE | Still deterministic policy; no explicit exploration via policy. | **SAC** uses stochastic policy and adds entropy regularization. |
| **SAC** | Stochastic actor-critic with **entropy bonus** for exploration + stability. | Critic: like TD3 but with entropy<br>Actor: maximizes value function with entropy | Requires tuning or learning the entropy coefficient $\alpha$; slightly higher computational cost due to two Q-networks and sampling from stochastic policy | |

| **Concept** | **Used In** | **Purpose** |
|---|---|---|
| Value Function Baseline V(s) | VPG, TRPO, PPO | Reduce gradient variance |
| Advantage Estimation A = R - V | All PG methods | Credit assignment |
| KL Constraint | TRPO | Prevent destructive updates |
| Clipping | PPO | Ensure stable updates |
| Entropy Bonus $\alpha \, log \, \pi$ | SAC | Encourage exploration |
| Target Networks | DDPG, TD3, SAC | Stabilize critic training |

## ➢ Example:

**Policy MLP Network:**

```
pi_net = nn.Sequential(
                nn.Linear(obs_dim, 64),
                nn.Tanh(),
                nn.Linear(64, 64),
                nn.Tanh(),
                nn.Linear(64, act_dim)
                )
```

**Loss:**

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right]$$

$s_t$ $a_t$ $R(\tau)$

```
def compute_loss(obs, act, weights):
    logp = get_policy(obs).log_prob(act)   log π_θ(a_t|s_t)
    return -(logp * weights).mean()
```

**Architecture:**

1) **mlp**(sizes, activation=nn.Tanh, output_activation=nn.Identity)

Gymnasium environment

2) **train**(env_name='CartPole-v0', hidden_sizes=[32], lr=1e-2, epochs=50, batch_size=5000, render=False)
   - logits_net = mlp(sizes=[obs_dim]+hidden_sizes+[n_acts]) policy network
   - get_policy(obs) action distribution (policy)
   - get_action(obs) action sampled from policy
   - compute_loss(obs, act, weights)
   - **train_one_epoch**()
     - batch_loss = compute_loss(obs=torch.as_tensor(np.array(batch_obs), dtype=torch.float32), act=torch.as_tensor(batch_acts, dtype=torch.int32), weights=torch.as_tensor(batch_weights, dtype=torch.float32))
     - batch_loss.backward()
     - optimizer.step()

3) for i in range(epochs): train loop

```
batch_loss, batch_rets, batch_lens = train_one_epoch()
torch.save(logits_net.state_dict(), 'trained_policy.pt')
```

- Achiam, J., & OpenAI. *Spinning Up in Deep Reinforcement Learning*. https://spinningup.openai.com/en/latest/

# Reinforcement Learning from Human Feedback (RLHF)

## ➢ Reinforcement Learning from Human Feedback (RLHF)

- **Reinforcement Learning (RL)** for **fine-tuning AI models**
  - Treat the **pretrained model** as a policy $\pi_\theta(a|s)$ where:
    $s$ : input context (e.g., prompt, observation), $a$ : model output (e.g., text, image, action), $\theta$ : model parameters

Then, apply **policy gradient optimization** to improve the model based on interaction or evaluation-based feedback.

**Example:** At each step:

- The model observes a **context** (previous tokens)
- Then **chooses the next token** from a fixed **vocabulary set**: $a_t \in \{token_1, token_2, \dots, token_v\}$
- The policy $\pi_\theta(a|s)$ is a **categorical distribution** over discrete actions.

## ➢ **Reinforcement Learning (RL)** for **fine-tuning AI models**

- Treat the **pretrained model** as a policy $\pi_\theta(a|s)$ where:

  $s$ : input context (prompt, observation), $a$ : model output (e.g., text, image, action), $\theta$ : model parameters

Then apply **policy gradient optimization** to improve the model via interaction or evaluation-based feedback.

**1. Trajectory Collection**

Instead of an environment simulator, we generate responses (i.e., trajectories) from the model: $D = \{(s_i, a_i, R_i)\}$

$s_i$ : user prompt or input, $a_i \sim \pi_\theta(.|s_i)$: model-generated output, $R_i$ : scalar reward (from a human, a reward model, or a metric)

**2. Policy Gradient Estimation** $\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right]$

Since most AI fine-tuning tasks don't have multi-step episodes, reduce to **one-step trajectory**: $\nabla_\theta \mathrm{J}(\pi_\theta) = \mathrm{E}_{s,a}[\nabla_\theta \log \pi_\theta(a|s)\, R(s,a)]$

**3. Add Reward-to-Go or Advantage**

- If we have multi-step outputs (e.g., token-by-token generation), we can compute reward-to-go instead of reward: $\hat{R}_t \doteq \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})$

- We can use **advantage estimation** with a baseline $b(s)$:

$$\nabla_\theta \mathrm{J}(\pi_\theta) = \mathrm{E}_{s,a}[\nabla_\theta \log \pi_\theta(a|s)\,(R(s,a) - b(s))]$$

In RLHF, $b(s)$ is typically learned from value function $V_\phi(s) \approx \mathrm{E}_a[R(s,a)]$

**4. Apply PPO or Clipping**

To ensure stable fine-tuning, we typically apply Proximal Policy Optimization (PPO).

# Imitation Learning (IL)

## ➢ Imitation Learning (IL)

- Agent learns to perform tasks by **observing and mimicking expert behavior**, rather than learning purely from trial and error like in traditional reinforcement learning (RL).
- Instead of learning from rewards, the agent is trained using **demonstrations from a human or expert policy.**

| Feature | Imitation Learning | Reinforcement Learning |
|---|---|---|
| Learning Signal | Expert demonstrations (no reward needed) | Reward feedback from environment |
| Exploration | No exploration (just mimic expert) | Actively explores actions |
| Sample Efficiency | High (if expert data is good) | Often low (needs many trials) |
| Generalization | May overfit to expert data | Can generalize well if trained properly |

**Common Imitation Learning Methods:**

**1.Behavior Cloning (BC)**:

Supervised learning from state-action pairs (like training a classifier).
The agent learns a policy $\pi(s) \approx a$ from demonstration data (s, a).
Simple but suffers from **compounding errors** due to distribution shift.

**2. Inverse Reinforcement Learning (IRL)** :

Learns the **reward function** that the expert is assumed to optimize.
Then uses RL to derive the policy under this reward.

**3. DAGGER (Dataset Aggregation)** :

An interactive approach: the agent tries to imitate, gets corrected by the expert, and the dataset is updated.
Reduces compounding errors better than Behavior Cloning.

**1. Behavior Cloning (BC)**

A **supervised learning** approach where the agent learns to imitate an expert's behavior by mapping observations to actions.

- Collect a dataset of **expert trajectories**: : $D = \{(s_1, a_1), \dots, (s_N, a_N)\}$
- Train a policy $\pi_\theta(a|s)$ to **minimize prediction error**: $\min_\theta \sum_{i=1}^{N} \|\pi_\theta(s_i) - a_i\|^2$

  Essentially: "Watch and copy the expert."

**Pros:** 1) Simple and fast to implement. 2) Works well when expert demonstrations cover the state space well.
**Cons:** 1)Fails if the agent visits unseen states not in the demo, 2) Compounding errors: small mistakes lead to drifting into unknown territory.

**2. DAGGER (Dataset Aggregation)**

- An iterative imitation learning algorithm that **fixes the distribution shift problem** in behavior cloning by aggregating more data from the agent's own policy.
- Let the agent **explore**, but always ask the expert what they would do, and use that to **update** the model.

1) Initialize policy with Behavior Cloning, 2) Roll out the current policy in the environment, 3) For each visited state, query the expert for the correct action, 4) Add these new (state, expert action) pairs to the training set, 5) Retrain the policy (like in BC)on the updated dataset, 6) Repeat.

**Pros:** 1) Reduces **compounding error** and distribution mismatch, 2) Handles unseen states by learning from them directly.
**Cons:** 1) Requires **online access to the expert** (e.g., human or simulator), 2) More expensive to run than pure behavior cloning.

- **Behavior Cloning**: One-time supervised imitation based on fixed expert demonstrations.
- **DAGGER**: Iterative imitation with expert corrections during agent exploration.

## ➢ Reinforcement Learning from Human Feedback (RLHF) VS Imitation Learning (IL)

| | **Imitation Learning (IL)** | **Reinforcement Learning from Human Feedback (RLHF)** |
|---|---|---|
| **Supervision Type** | Supervised learning (behavior cloning) | Reinforcement learning (reward-based) |
| **Human Data Type** | Human demonstrations (actions or outputs) | Human **preferences** or **rankings** over model outputs |
| **Learning Signal** | Imitate human actions/output exactly | Learn to optimize behavior by maximizing rewards from a reward model |
| **Goal** | Mimic expert behavior | Align with human preferences / values |
| **Training Objective** | Minimize prediction error on human-provided actions | Maximize expected reward via policy gradient (e.g., PPO) |
| **Exploration** | No exploration (passive learning) | Includes exploration (active learning via policy updates) |
| **Example Use** | Self-driving vehicles, robot control demonstrations | ChatGPT, InstructGPT, LLM alignment |
| **Algorithm Examples** | Behavior Cloning, DAGGER | RLHF pipeline (Reward model + PPO) |

# Adversarial Training

## ➢ Adversarial Training

These topics explore how to train models to resist adversarial inputs — subtle changes in the input meant to fool the model.

- **Key Topics:**
  1. **Adversarial Examples in NLP** (e.g., word substitutions, paraphrasing attacks)
  2. **Adversarial Training for Text**: Fine-tuning models using perturbed inputs
  3. **Contrastive Learning under Adversarial Perturbations**
  4. **Certified Robustness in Text Models**: Provable guarantees against perturbations
  5. **Gradient-based vs. Black-box Attacks**: FGSM, PGD, HotFlip, TextFooler, BERT-Attack
  6. **Robust Optimization Frameworks**: Min-max formulations

**1. Adversarial Examples in NLP**

- **Goal:** Create small, often imperceptible changes to inputs (like synonyms or typos) that cause the model to fail.
- **Examples:** i) Word Substitutions: Replace words with similar ones (e.g., "happy" → "glad"), ii) Paraphrasing Attacks: Reword the sentence while preserving its original meaning
- **Tools:** TextFooler, BERT-Attack

**2. Adversarial Training for Text**

- **Goal:** Improve model robustness by training it on both clean and adversarially perturbed inputs.
- **Method:** Generate adversarial examples during training and fine-tune the model on them.
- **Effect:** Helps the model learn to resist attacks and generalize better.

**3. Contrastive Learning under Adversarial Perturbations**

- **Goal:** Teach the model to keep similar inputs close in representation space even under perturbation.
- **Method:** Use contrastive loss to align clean and adversarial versions of the same sentence.
- **Example:** Extend SimCSE or BERT embeddings with adversarial contrastive learning.

**4. Certified Robustness in Text Models**

- **Goal:** Provide formal guarantees that the model is robust to certain changes (e.g., up to *k* word substitutions).
- **Method:** Use interval bound propagation, randomized smoothing, or convex relaxations to certify robustness.
- **Tradeoff:** Provides high reliability but often incurs significant computational cost and limited scalability.

**5. Gradient-based vs. Black-box Attacks**

- **Gradient-based (White-box):** Use the model's gradients to craft attacks.
  - Examples: FGSM (Fast Gradient Sign Method), PGD (Projected Gradient Descent), HotFlip
- **Black-box:** Don't access the model's internals; use outputs to guide attacks.
  - Examples: TextFooler, BERT-Attack

**6. Robust Optimization Frameworks**

- **Goal:** Formulate training as a **min-max optimization** problem: $\min_{\theta} \max_{\delta \in \Delta} \mathcal{L}(f_\theta(x+\delta), y)$
- The model learns parameters $\theta$ that minimize worst-case loss under allowable perturbations $\delta$.
- **Used in:** Adversarial training with PGD, DRO (Distributionally Robust Optimization), etc.

# Robustness in Language Models

➢ **Robustness in Language Models:**

Robustness here refers to the model's reliability under distribution shifts, noise, and unexpected or adversarial inputs.

- **Key Topics:**
  1. **Out-of-Distribution (OOD) Generalization**: How well LLMs perform on unseen data
  2. **Input Noise Robustness**: Typos, code-switching, slang
  3. **Semantic Robustness**: Understanding paraphrased or logically equivalent sentences
  4. **Robustness to Prompt Variations**: Sensitivity to prompt phrasing (prompt engineering)
  5. **Robustness to Distribution Shift**: E.g., test-time domain adaptation
  6. **Multi-lingual & Cross-lingual Robustness**
  7. **Robustness in Instruction Tuning and RLHF Models**

**1. Out-of-Distribution (OOD) Generalization**

- **Goal:** Ensure LLMs can perform well on data that differs from their training distribution.
- **Method:** Evaluate performance on datasets with domain shifts (e.g., legal, medical, informal text) or adversarial examples. Use domain adaptation, data augmentation, or contrastive training.

**2. Input Noise Robustness**

- **Goal:** Handle noisy inputs like typos, slang, or code-switching (mixing languages).
- **Method:** Apply noise injection during training, use denoising autoencoders, or fine-tune on noisy datasets to make the model resilient.

**3. Semantic Robustness**

- **Goal:** Understand different ways of expressing the same meaning (e.g., paraphrasing).
- **Method:** Train on paraphrase datasets or apply semantic contrastive learning to align representations of semantically equivalent inputs.

**4. Robustness to Prompt Variations**

- **Goal:** Ensure consistent performance across similar but differently phrased prompts.
- **Method:** Evaluate across prompt templates, use prompt tuning, or train with instruction variations.

**5. Robustness to Distribution Shift**

- **Goal:** Adapt to test-time data that differs from training data.
- **Method:** Use unsupervised domain adaptation, meta-learning, or distributionally robust optimization (DRO) to generalize under shifts.

**6. Multi-lingual & Cross-lingual Robustness**

- **Goal:** Perform well across multiple languages or when switching between them.
- **Method:** Pretrain on multilingual corpora and evaluate on cross-lingual benchmarks.

**7. Robustness in Instruction Tuning and RLHF Models**

- **Goal:** Maintain reliable behavior when trained with human feedback or instruction tuning.
- **Method:** Use diverse and high-quality prompts, enforce reward model consistency, and evaluate robustness via human preference tests under varied conditions.

# Language Models Evaluation

## ➢ LLM Evaluation

| Evaluation Type | What It Measures |
|---|---|
| Task-based | Task accuracy/performance |
| Perplexity | Language modeling quality |
| Human evaluation | Alignment with human judgment |
| Safety/bias | Responsible AI behavior |
| Instruction following | Ability to generalize to new instructions |

## ➢ LLM Evaluation

| Task Type | Example Benchmarks |
|---|---|
| QA | SQuAD, TriviaQA |
| Summarization | CNN/DailyMail, XSum |
| Translatioân | WMT |
| Natural Language Inference (NLI) | MNLI, ANLI |
| Code Generation | HumanEval, MBPP |

**1. Task-Based Evaluation (Objective Performance)**

Tests the model on benchmark datasets for NLP tasks.

**Metrics:** Accuracy, F1-score (Harmonic Mean of Precision & Recall), BLEU (Bilingual Evaluation Understudy), ROUGE (Recall-Oriented Understudy for Gisting Evaluation), EM (Exact Match), etc.

**2. Language Model Evaluation (Perplexity)**

- Measures how well the model predicts the next token in a sequence.
- Lower **perplexity** means better fluency and modeling of natural language. $\text{Perplexity} = e^{-\frac{1}{N}\sum_{i=1}^{N}\log P(x_i)}$

**3. Human Preference Evaluation**

- Ask human annotators to compare outputs generated by multiple models.
- Often used in **RLHF** pipelines.

Metric: Win rate, human preference scores, Likert scale ratings.

**4. Safety, Bias, and Toxicity Evaluation**

Evaluate whether the model produces **harmful, biased, or toxic content**.

Tools & Datasets: RealToxicityPrompts, BiasBench, StereoSet, ToxiGen

**5. Instruction Following / Generalization**

How well does the model follow complex or unseen instructions?

Example Benchmarks:

• HELMeval, BIG-bench, MMLU (Massive Multitask Language Understanding)

• ARC (AI2 Reasoning Challenge)

**6. Holistic Evaluation Frameworks**

Some common frameworks used to evaluate LLMs:

| Framework | Description |
|---|---|
| HELM | Holistic Evaluation of Language Models (Stanford) — evaluates accuracy, robustness, fairness, efficiency, etc. |
| ELO rating | Models are matched against each other like in chess; useful for human-vs-model or model-vs-model comparisons |
| MT-Bench | Multi-turn evaluation for chat agents |
| Arena | LLM-as-a-judge framework for comparing model outputs |

# Python Libraries for Generative AI

- These are base libraries for building and training generative models:

| Library | Purpose |
|---|---|
| torch | Core deep learning framework (for VAEs, GANs, Transformers, Diffusion) |
| torchvision | Preprocessing, datasets, pretrained vision models |
| numpy | Numerical operations and data handling |
| scikit-learn | Basic evaluation tools, clustering, PCA, etc. |
| matplotlib / seaborn | Visualization and diagnostics |

- These are libraries for building and training various types of generative models:

| Model Type | Libraries |
|---|---|
| **VAE (Variational Autoencoder)** | torch (deep learning), pyro (probabilistic layers), matplotlib (for visualizing reconstructions, latent space) |
| **GAN (Generative Adversarial Network)** | torch, torchgan (prebuilt GAN blocks and training utilities), wandb (to visualize training stability, e.g., loss curves, FID) |
| **Diffusion Model** | torch, diffusers (model & pipeline), xformers (efficient attention mechanisms for large-scale diffusion models) |
| **Transformer / LLM** | transformers (models & tokenizers), datasets (datasets like WikiText, Common Crawl), accelerate (multi-GPU training) |
| **Multimodal Model (e.g. CLIP, BLIP)** | transformers, lavis or openclip (text-image alignment models) |

## ➢ List of Libraries:

**1. transformers (by Hugging Face)**
• **Application**: Pretrained models (GPT, BERT, T5, CLIP, DALL·E Mini, etc.)
• **Used for**: Language generation (e.g., GPT-2, GPT-3, LLaMA), Text classification, summarization, translation, Multimodal models (e.g., text-to-image, image captioning)

**2. diffusers (by Hugging Face)**
• **Application**: Diffusion models like **Stable Diffusion**
• **Used for**: Image generation from text prompts, Training or customizing your own diffusion models, Supports DDPM, DDIM, etc.

**3. torch (PyTorch)**
• **Application**: General deep learning framework
• **Used for**: Building all kinds of generative models (GANs, VAEs, Transformers) from scratch, Custom training loops and fine-tuning, Widely used in academic and production Gen AI research

**4. tensorflow / keras**
• **Application**: Deep learning framework (alternative to PyTorch)
• **Used for**: Same as PyTorch: VAEs, GANs, RNNs, Transformers,  and user-friendly Keras APIs for rapid prototyping

**5. Pyro / NumPyro**
• **Application**: Probabilistic programming
• **Used for**: Variational Autoencoders (VAEs), Normalizing flows, Bayesian generative models

**6. nflows**
• **Application**: Normalizing flow models
• **Used for**: Density estimation, Generating structured samples from learned probability distributions (e.g., complex posteriors), Latent generative models

**7. Magenta (by Google)**
• **Application**: Music and art generation
• **Used for**: Music composition (MIDI), AI-generated art, Experiments with LSTM and VAE for melodies

**8. TorchGAN**
• **Application**: Generative Adversarial Networks (GANs)
• **Used for**: Prebuilt GAN architectures (DCGAN, WGAN, etc.), evaluation metrics (e.g., Inception Score, FID), Custom GAN training

**9. OpenCLIP**
• **Application**: Contrastive vision-language models (like CLIP)
• **Used for**: Image-text alignment, Text-to-image conditioning, Visual grounding and retrieval

**10. LAVIS (by Salesforce)**
• **Application**: Vision-language foundation models
• **Used for**: BLIP, ALBEF, and other vision-language models (e.g., captioning, VQA, retrieval)

**11. stable-baselines3**
• **Application**: Reinforcement learning
• **Used for**: Training agents that could be used in RLHF (Reinforcement Learning with Human Feedback), Could integrate with LLMs for aligned learning

**12. Gradio / Streamlit**
• **Application**: Interactive interfaces for Gen AI models
• **Used for**: Creating interactive web apps for model demos (e.g., text/image generation), Building UIs for ML apps without frontend code

**13. xformers**
• **Application**: Efficient attention layers
• **Used for**: Memory-efficient transformers, Essential in large models and diffusion pipelines

**14. Taming-Transformers**
• **Application**: Vector-quantized VAEs used in DALL·E and latent diffusion
• **Used for**: Discrete latent space learning, Compressing inputs before training generative models

# Generative AI Foundations

## Algorithms and Architectures

**Ashkan Jasour**

2024-2025

Course Website: jasour.github.io/generative-ai-course